Rust Programming: Systems Programming Reimagined

Module 3: Structured Data, Generics & Fearless Concurrency

Chandrashekar Babu · training@chandrashekar.info

https://www.chandrashekar.info/ | https://www.slashprog.com/

Module 3 Overview

This module covers the full spectrum of Rust's data modelling, abstraction, and concurrency primitives — from structs and enums through to fearless multi-threaded programming.

1

Part 3.1 — Data Structures & Functional Tools

Structs, Enums, Unions · String, Vec, HashMap · Pattern Matching · Closures & Iterators

2

Part 3.2 — Generics & Traits Preview

Generic Data Structures · Generic Functions · Introduction to Traits

3

Part 3.3 — Fearless Concurrency

C/C++ Problems · std::thread · Message Passing (Channels) · Shared State (Mutex, Arc)

4

Part 3.4 — Lab Session

Multi-threaded Data Processor · Shared Cache Implementation

PART 3.1

Data Structures & Functional Tools

The first part of Module 3 dives deep into Rust's core building blocks for structuring data and writing expressive, functional-style code. Master these and you will write idiomatic Rust with confidence.

Structs & Enums

Model your domain with named-field structs, tuple structs, rich enums with data, and the indispensable `Option` and `Result` types.

Collections

Work with `String`, `Vec<T>`, and `HashMap<K,V>` — the three collections you will reach for in virtually every Rust programme.

Pattern Matching

Exhaustive, compiler-checked pattern matching with `match`, `if let`, and `while let` — the most powerful control-flow tool in Rust.

Closures & Iterators

Capture the environment with closures, then compose lazy iterator pipelines for expressive, zero-cost data transformations.

Structs — Grouping Related Data

Rust provides three flavours of struct, each suited to a different use case. Named-field structs are the most common and most readable. Tuple structs give a named type to a positional grouping. Unit structs carry no data but can implement traits.

Named-Field Struct

```
struct User {
  username: String,
  email: String,
  sign_in_count: u64,
  active: bool,
}
```

Fields accessed by name — the default choice for clarity.

Tuple Struct & Unit Struct

```
// Tuple struct
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

// Unit struct
struct AlwaysEqual;
```

Tuple structs are compact named wrappers. Unit structs are useful as markers or when implementing traits with no state.

```
let user = User {
 email: String::from("user@example.com"),
 username: String::from("rustacean"),
 active: true, sign_in_count: 1,
};
let black = Color(0, 0, 0);
let subject = AlwaysEqual;
```

Struct Update Syntax & Methods

The `..other` shorthand lets you create a new struct instance by copying fields from an existing one — but be aware that fields using `String` or other non-`Copy` types will be **moved**, invalidating the source. Methods live in `impl` blocks and take `&self`, `&mut self`, or no `self` for associated functions.

```
let user2 = User {
  email: String::from("user2@example.com"),
  ..user1  // remaining fields moved from user1
};
```

Associated Function

```
fn new(email: String, username:
String) -> Self {
  User { email, username, active: true,
sign_in_count: 1 }
}
```

Called as `User::new(...)`. No `self` parameter — Rust's constructor convention.

Immutable Method

```
fn get_email(&self) -> &str {
  &self.email
}
```

Borrows `self` immutably. Can be called on both owned and borrowed instances.

Mutable Method

```
fn deactivate(&mut self) {
  self.active = false;
}
```

Requires the binding to be declared `mut`. The borrow checker ensures no aliasing.

Enums — Expressing Alternatives

Rust's enums are algebraic data types: each variant can carry **different data**. This makes them far more expressive than C-style enums, enabling rich domain modelling in a single type definition.

Simple Enum

```
enum IpAddrKind { V4, V6 }
```

Named constants — no data attached.

Data-Carrying Enum

```
enum IpAddr {
  V4(u8, u8, u8, u8),
  V6(String),
}
```

Each variant carries its own payload.

Mixed Variants

```
enum Message {
  Quit,
  Move { x: i32, y: i32 },
  Write(String),
  ChangeColor(i32, i32, i32),
}
```

Variants can be unit, tuple, or struct-like.

Option and Result — The Most Important Enums

`Option<T>` eliminates null-pointer bugs: the compiler forces you to handle both `Some(T)` and `None` before using a value. `Result<T, E>` does the same for operations that can fail. Together, they make Rust programmes remarkably reliable without sacrificing performance.

Option<T> — Value or Nothing

```
let some_number = Some(5);
let absent: Option<i32> = None;

// Compiler FORCES you to handle None
match absent {
  Some(n) => println!("{}", n),
  None    => println!("nothing"),
}
```

There is no implicit `null`. Every absence is explicit and checked.

Result<T, E> — Success or Failure

```
fn divide(a: f64, b: f64) -> Result<f64, String> {
  if b == 0.0 { Err("Division by zero".into()) }
  else        { Ok(a / b) }
}

match divide(10.0, 2.0) {
  Ok(result) => println!("Result: {}", result),
  Err(e)     => println!("Error: {}", e),
}
```

Errors are values. Unhandled errors produce compiler warnings or type errors.

Unions — Unsafe Overlapping Storage

Unions allow multiple fields to share the same memory region, mirroring C's `union`. Accessing any field is `unsafe` because Rust cannot guarantee which interpretation is valid. In practice, **99.9% of Rust code never needs unions** — they exist primarily for C FFI and extremely low-level bit manipulation.

```
#[repr(C)]
union MyUnion {
    i: i32,
    f: f32,
    b: [u8; 4],
}

let u = MyUnion { i: 42 };
unsafe {
    println!("As i32:  {}", u.i);   // 42
    println!("As f32:  {}", u.f);   // Garbage! Same bits as f32
    println!("As bytes: {:?}", u.b);
}
```

When You Might Need a Union

- FFI interop with C unions
- Extremely memory-constrained embedded systems
- Implementing low-level type punning

Rule of Thumb

Prefer `enum` over `union` in all idiomatic Rust. Enums are safe, pattern-matchable, and carry the same zero-cost guarantee.

Collections — String Deep Dive

`String` in Rust is a UTF-8 encoded, heap-allocated, growable byte sequence. Because UTF-8 code points can be 1–4 bytes, **direct indexing by integer is not allowed** — a deliberate design decision that prevents silent bugs when processing multi-byte characters.

```
let mut s = String::from("hello");
s.push('!');              // Add a char
s.push_str(" world");     // Add a &str
s += "!!!";               // Concatenate (moves s!)

let s4 = s1 + &s2 + &s3; // + operator: s1 moved, s2/s3 borrowed
let s5 = format!("{}-{}-{}", s2, s3, "rust"); // No ownership transfer
```

No Direct Indexing

```
// let c = s[0]; // ERROR!
```

Use `.chars()` for Unicode scalar values or `.bytes()` for raw bytes.

Safe Slicing

```
let slice = &s5[0..4];
```

Will panic at runtime if the range falls mid-character boundary.

Iteration

```
for c in s.chars() { }
for b in s.bytes() { }
```

Always prefer explicit iteration over ad-hoc indexing.

Collections — Vec<T> (Dynamic Arrays)

`Vec<T>` is Rust's go-to growable array. It stores elements contiguously on the heap, giving cache-friendly sequential access. Accessing by index with `[]` panics on out-of-bounds; using `.get()` returns `Option<&T>` for safe access.

Creating & Mutating

```
let mut v1: Vec<i32> = Vec::new();
let v2 = vec![1, 2, 3];  // macro

v1.push(5);
v1.push(6);
let last   = v1.pop();       // Option<T>
let second = v1.remove(1);   // panics if OOB
```

Safe vs Panic Access

```
let third = &v1[2];          // panics OOB
let third = v1.get(2);       // Option<&T>

for i in &v1     { }         // &T
for i in &mut v1 { *i += 10; } // &mut T
```

Handy Methods

```
v1.len(); v1.is_empty();
v1.contains(&5); v1.sort();
v1.dedup();
```

Collections — HashMap<K, V>

`HashMap` provides O(1) average-case lookup, insertion, and deletion. Keys and values are **moved into** the map on insertion, so the map owns them. The `.entry()` API is particularly ergonomic for update-or-insert patterns without double-lookup.

```
use std::collections::HashMap;
let mut scores: HashMap<String, i32> = HashMap::new();
scores.insert(String::from("Blue"),   10);
scores.insert(String::from("Yellow"), 50);
```

Overwrite

```
scores.insert("Blue".into(), 25);
```

Replaces existing value unconditionally.

Insert If Absent

```
scores.entry("Blue".into())
      .or_insert(50);
```

Only inserts when the key has no entry.

Update or Insert

```
scores.entry("Red".into())
      .and_modify(|e| *e += 10)
      .or_insert(1);
```

The most expressive pattern — avoids two separate lookups.

Ownership note: inserting `field_name` and `field_value` moves them — they cannot be used after the call.

Pattern Matching Deep Dive — match

Rust's `match` is exhaustive: the compiler rejects any match expression that doesn't handle all possible variants. This, combined with rich pattern syntax — multiple patterns, ranges, guards, and binding — makes it a safer and far more expressive replacement for `switch/case`.

Enum Matching with Binding

```
match coin {
  Coin::Penny          => 1,
  Coin::Nickel         => 5,
  Coin::Dime           => 10,
  Coin::Quarter(state)  => {
    println!("From {}!", state); 25
  }
}
```

Multiple Patterns, Ranges & Guards

```
match n {
  0            => "zero",
  1 | 2        => "one or two",
  3..=10       => "three to ten",
  n if n < 0   => "negative",
  _            => "something else",
}
```

The `_` wildcard is the catch-all. The guard `if n < 0` adds runtime conditions to a pattern arm.

Pattern Matching — Destructuring

Patterns can reach inside structs and enums to bind their fields directly. Partial destructuring with `..` lets you ignore fields you don't care about, keeping match arms concise and readable.

```
let point = Point { x: 10, y: 20 };

// Destructure struct fields
match point { Point { x, y } => println!("({}, {})", x, y) }

// Rename fields
match point { Point { x: a, y: b } => println!("({}, {})", a, b) }

// Ignore y with ..
match point { Point { x, .. } => println!("x is {}", x) }
```

```
let msg = Message::Move { x: 5, y: 10 };
match msg {
 Message::Quit => println!("Quit"),
 Message::Move { x, y } => println!("Move to ({}, {})", x, y),
 Message::Write(text) => println!("Text: {}", text),
 Message::ChangeColor(r,g,b) => println!("Color: {} {} {}", r, g, b),
}
```

Pattern Matching — if let and while let

When you only care about **one** pattern variant, `if let` replaces a verbose two-arm `match` with concise, readable code. `while let` extends this to loops, consuming a stack or iterator until the pattern no longer matches.

if let — Concise Single-Pattern

```
// Verbose
match optional {
  Some(x) => println!("{}", x),
  None => (),
}

// Concise
if let Some(x) = optional {
  println!("{}", x);
} else {
  println!("Nothing!");
}
```

while let — Loop Until No Match

```
let mut stack = vec![1, 2, 3];
while let Some(top) = stack.pop() {
  println!("{}", top);
}
// Prints 3, 2, 1 then stops
```

Result with if let

```
let result: Result<i32, &str> = Ok(42);
if let Ok(value) = result {
 println!("Success: {}", value);
}
```

Closures — Anonymous Functions

Closures are anonymous functions that can capture variables from their enclosing scope. Unlike regular functions, each closure has a unique, compiler-generated type. The `move` keyword forces the closure to take **ownership** of every captured variable — essential when sending a closure to another thread.

```
let x = 5;
let add_x = |y| y + x;     // Captures x by reference
println!("{}", add_x(10));   // 15

let s = String::from("hello");
let consume = move || {      // Moves s into closure
  println!("{}", s);
};
consume();
// println!("{}", s);  // ERROR: s was moved!
```

Type Inference

```
let add_one = |x: i32| -> i32 { x + 1 };
let add_one = |x| x + 1;   // Inferred
```

Higher-Order Functions

```
fn call_twice<F: Fn(i32) -> i32>(f: F, x: i32) -> i32 {
 f(f(x))
}
call_twice(|x| x * 2, 5); // 20
```

Closure Traits — Fn, FnMut, FnOnce

Rust classifies closures into three traits based on how they interact with captured variables. A function that accepts `Fn` is the most restrictive (any closure works). `FnOnce` is the most permissive (accepts even consuming closures). Choosing the right bound keeps your API flexible without sacrificing safety.

Fn

Immutably borrows captured values. Can be called any number of times. Most closures fall into this category.

```
let x = 5;
let print_x = || println!("{}", x);
print_x(); print_x(); // Both OK
```

FnMut

Mutably borrows captured values. Can be called multiple times but requires `mut` binding.

```
let mut y = 5;
let mut add_one = || { y += 1; };
add_one(); add_one(); // y = 7
```

FnOnce

Consumes captured values. Can only be called **once** — calling again is a compile error.

```
let s = String::from("hi");
let consume = || drop(s);
consume(); // OK
// consume(); // ERROR
```

Iterators — Lazy Processing

Iterators in Rust are **lazy**: creating an iterator chain performs no computation. Work only happens when a *consuming adapter* such as `.collect()`, `.sum()`, or `.count()` is called. This design enables expressive pipeline-style code with zero runtime overhead compared to hand-written loops.

```
let numbers = vec![1, 2, 3, 4, 5];

let result: i32 = numbers
    .iter()
    .map(|&x| x * 2)      // Iterator adapter — lazy
    .filter(|&x| x > 5)   // Iterator adapter — lazy
    .sum();               // Consuming adapter — triggers work
```

1

.iter()

Yields `&T`. Non-consuming; original collection remains valid.

2

Adapters

`.map()`, `.filter()`, `.take()`, `.skip()` — each returns a new lazy iterator.

3

Consumers

`.collect()`, `.sum()`, `.fold()`, `.count()` — trigger evaluation of the chain.

Common Iterator Methods

The standard library's iterator API is vast, but a core set of methods covers the overwhelming majority of use cases. Chaining these adapters replaces verbose imperative loops with concise, readable data-processing pipelines.

Transformation & Filtering

```
let squares: Vec<_> = v.iter()
    .map(|&x| x * x).collect();

let evens: Vec<_> = v.iter()
    .filter(|&&x| x % 2 == 0).collect();
```

Reducing

```
let sum = v.iter().fold(0, |acc, &x| acc + x);
let product = v.iter().product::<i32>();
```

Searching

```
v.iter().any(|&x| x == 3)
v.iter().all(|&x| x > 0)
v.iter().find(|&&x| x % 2 == 0)
v.iter().position(|&x| x == 3)
```

Combining & Slicing

```
v.iter().take(2).collect::<Vec<_>>()
v.iter().skip(2).collect::<Vec<_>>()
v.iter().zip(other.iter()).collect()
v.chunks(2).collect()
```

PART 3.2

Generics & Traits Preview

Generics allow you to write a single implementation that works for many concrete types, eliminating code duplication without any runtime cost. Traits define shared behaviour — Rust's answer to interfaces. Together they form the backbone of the entire standard library.

Why Generics?

Without generics, you write `largest_i32`, `largest_char`, and so on. With generics, one function handles every comparable type — same logic, zero duplication.

Generic Structs & Enums

`Option<T>` and `Result<T, E>` in the standard library are themselves generic enums — you define the pattern once and the compiler generates code for each type you use.

Traits as Contracts

A trait defines the interface a type must fulfil. Trait bounds on generic parameters constrain which types are accepted, keeping generics type-safe and expressive.

Why Generics? Eliminating Duplication

The classic motivating example: you write `largest_i32` for integers and then copy-paste the identical logic for `largest_char`. Any future bug fix must be applied twice. Generics collapse these duplicates into a single, correct implementation that the compiler specialises at compile time — with **zero runtime overhead**.

Without Generics — Duplicated Code

```
fn largest_i32(list: &[i32]) -> i32 {
  let mut largest = list[0];
  for &item in list {
    if item > largest { largest = item; }
  }
  largest
}
// ... identical for char, f64, etc.
```

With Generics — One Implementation

```
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
 let mut largest = list[0];
 for &item in list {
 if item > largest { largest = item; }
 }
 largest
}

// Works for any T that is PartialOrd + Copy
largest(&[34, 50, 25, 100]);
largest(&['y', 'm', 'a', 'q']);
```

Generic Structs

Structs can be parameterised over one or more type variables. The `Point<T>` pattern means a single struct definition serves both integer and floating-point coordinates — the compiler generates the concrete type you actually use.

```
struct Point<T>     { x: T, y: T }
struct Pair<X, Y>  { first: X, second: Y }
```

Same-Type Fields

```
let int_pt  = Point { x: 5,   y: 10 };
let flt_pt  = Point { x: 1.0, y: 4.0 };
```

Both are `Point<T>` — different `T` at each call site.

Different-Type Fields

```
let pair = Pair { first: 5, second: "hi" };
// Pair<i32, &str>
```

Multiple type params let fields differ in type.

Enums with Generics

```
enum MyOption<T>     { Some(T),
None }
enum MyResult<T, E> { Ok(T), Err(E) }
```

This is exactly how `Option` and `Result` are defined in `std`.

Generic Methods

Methods on generic structs are written inside `impl<T> Point<T>` blocks. You can also write **specialised** implementations that only apply to a specific concrete type, such as `impl Point<f64>` — those methods are not available on `Point<i32>`.

General Method — Any T

```
impl<T> Point<T> {
  fn x(&self) -> &T { &self.x }

  fn mixup<U>(self, other: Point<U>) -> Point<U> {
    Point { x: other.x, y: other.y }
  }
}
```

Available on every `Point<T>` regardless of what `T` is.

Specialised Method — Only f64

```
impl Point<f64> {
  fn distance_from_origin(&self) -> f64 {
    (self.x.powi(2) + self.y.powi(2)).sqrt()
  }
}

let p = Point { x: 3.0, y: 4.0 };
p.distance_from_origin(); // 5.0
```

Calling `distance_from_origin()` on a `Point<i32>` is a compile error.

Generic Functions

Generic functions are parameterised over types at the call site. Trait bounds — either inline with `T: Trait` or in a `where` clause — constrain which types are accepted, giving the function body access to the operations that trait defines.

```
fn identity<T>(value: T) -> T { value }

fn print_slice<T: std::fmt::Debug>(slice: &[T]) {
  for item in slice { println!("{:?}", item); }
}
```

Multiple Bounds — Inline

```
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T
```

Concise for one or two bounds on a single type parameter.

Complex Bounds — where Clause

```
fn complex<T, U>(t: T, u: U) -> String
where
  T: std::fmt::Display + Clone,
  U: std::fmt::Debug  + PartialEq,
{ ... }
```

The `where` clause keeps the signature readable when bounds grow complex.

Introduction to Traits

A trait declares a set of methods a type must implement. Default implementations let you provide fallback behaviour — implementors can choose to override or accept the default. This is Rust's primary mechanism for polymorphism and code sharing across unrelated types.

Defining a Trait

```
trait Summary {
  fn summarize(&self) -> String;

  // Default implementation
  fn summarize_author(&self) -> String {
    String::from("(Unknown author)")
  }
}
```

Implementing for Two Types

```
impl Summary for NewsArticle {
 fn summarize(&self) -> String {
 format!("{}, by {} ({})",
 self.headline, self.author, self.location)
 }
}

impl Summary for Tweet {
 fn summarize(&self) -> String {
 format!("{}: {}", self.username, self.content)
 }
 fn summarize_author(&self) -> String {
 format!("@{}", self.username)
 }
}
```

Using Traits as Parameters & Return Types

Traits can appear as function parameter types — either via the concise `impl Trait` syntax or the equivalent generic trait-bound form. You can also return `impl Trait` to hide the concrete type, useful for returning closures or complex iterator chains from public APIs.

1 impl Trait Syntax

```
fn notify(item: &impl Summary) {
  println!("{}", item.summarize());
}
```

2 Trait Bound Syntax (Equivalent)

```
fn notify<T: Summary>(item: &T) { ... }
```

3 Multiple Bounds

```
fn notify_debug<T: Summary + std::fmt::Debug>(item: &T) {
... }
```

4 Return impl Trait

```
fn returns_summarizable() -> impl Summary {
  Tweet { username: "horse_ebooks".into(), ... }
}
```

The caller only sees `Summary`; the concrete type is hidden.

Trait Objects vs. Generics

Generics use **static dispatch**: the compiler monomorphises each call site, generating specialised code per type. Trait objects (`&dyn Trait`) use **dynamic dispatch** via a vtable at runtime. Generics are faster; trait objects enable heterogeneous collections and runtime polymorphism.

Static Dispatch — Generics

```
fn static_dispatch<T: Animal>(animal: T) {
  animal.make_sound();
}
// Compiler creates one version for Dog,
// one version for Cat — zero overhead.
```

Dynamic Dispatch — dyn Trait

```
fn dynamic_dispatch(animal: &dyn Animal) {
  animal.make_sound();
}
// Mix types in a collection:
let animals: Vec<&dyn Animal> = vec![&Dog, &Cat];
for a in animals { a.make_sound(); }
```

Choose Generics When

All types are known at compile time. Performance is critical. You want zero-cost abstraction.

Choose Trait Objects When

You need a heterogeneous collection. Types are determined at runtime. You're building a plugin system.

PART 3.3

Fearless Concurrency

Concurrency is notoriously hard to get right in C and C++. Rust's ownership model turns an entire class of concurrency bugs — data races — into compile errors, letting you write multi-threaded code with genuine confidence.

Concurrency in C/C++ — The Problems

In C and C++, nothing stops two threads from simultaneously reading and writing the same memory. The compiler happily compiles the code, the problem only surfaces at runtime — sometimes only under load in production. This is undefined behaviour, meaning the programme's output is literally unpredictable.

```
// C++ — DATA RACE, compiles fine!
int counter = 0;
void increment() {
  for (int i = 0; i < 1000000; i++)
    counter++;  // NOT ATOMIC — undefined behaviour!
}
// Two threads: result is NOT reliably 2,000,000
```

Data Races

Undefined behaviour — silently corrupt state.

Forgotten Locks

Easy to omit; no compiler warning.

Deadlocks

Runtime bugs, hard to reproduce.

No Compiler Help

"Works on my machine" syndrome.

Rust's Ownership-Based Approach

Rust's ownership and borrow rules apply equally to threads. The borrow checker ensures that at any instant, shared data is either accessed by **many readers** or **one writer** — never both. This guarantee is enforced entirely at compile time, with no runtime overhead.

```
use std::thread;

fn main() {
  let counter = 0;
  let handle = thread::spawn(move || {
    // counter is moved here — main thread can't touch it
  });
  // println!("{}", counter); // ERROR: counter moved!
}
```

Multiple Readers

Many threads may hold `&T` simultaneously — no mutation possible, no race.

One Writer

Exactly one thread holds `&mut T` — all other access is forbidden at compile time.

Compile-Time Guarantee

No runtime checks, no performance cost. If it compiles, it is free of data races.

Creating Threads with std::thread

`thread::spawn` takes a closure and runs it on a new OS thread, returning a `JoinHandle`. Calling `.join()` blocks the current thread until the spawned thread completes. The `move` keyword transfers ownership of captured variables into the thread closure — preventing dangling references across thread boundaries.

Basic Spawn & Join

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

let handle = thread::spawn(|| {
  for i in 1..10 {
    println!("Thread: {}", i);
    thread::sleep(Duration::from_millis(1));
  }
});

for i in 1..5 { println!("Main: {}", i); }
handle.join().unwrap(); // Wait for thread
```

move Closure — Transfer Ownership

```
let v = vec![1, 2, 3];

let handle = thread::spawn(move || {
  println!("Vector: {:?}", v); // v moved in
});

handle.join().unwrap();
// println!("{:?}", v); // ERROR: v moved!
```

Without `move`, Rust can't guarantee `v` outlives the thread — a compile error that saves you from a dangling pointer.

Thread Builder and Thread Control

`thread::Builder` gives fine-grained control over a thread before it starts: you can set a human-readable name (visible in panic messages and debuggers) and a custom stack size. The handle's return value from `.join()` lets you detect whether the thread panicked.

```
let builder = thread::Builder::new()
    .name("worker".into())
    .stack_size(1024 * 1024); // 1 MB stack

let handle = builder.spawn(|| {
  println!("Thread name: {:?}", thread::current().name());
}).unwrap();
handle.join().unwrap();
```

Detecting Panics

```
let handle = thread::spawn(|| panic!("Oh no!"));
match handle.join() {
  Ok(_)  => println!("Succeeded"),
  Err(_) => println!("Thread panicked"),
}
```

CPU Hints

```
thread::sleep(Duration::from_secs(1));
thread::yield_now(); // Voluntarily give up CPU
```

Useful for cooperative scheduling in tight loops.

Message Passing with Channels

Rust's standard library provides multi-producer, single-consumer (mpsc) channels. Sending a value **transfers ownership** into the channel — the sender can no longer use the value. This eliminates shared state entirely: threads communicate by passing owned data, making data races structurally impossible.

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

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

thread::spawn(move || {
  let messages = vec!["hi", "from", "the", "thread"];
  for msg in messages {
    tx.send(msg).unwrap(); // msg MOVED into channel
  }
});

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

Key insight: `send()` transfers ownership. No shared state means no data race — the channel IS the synchronisation mechanism.

Multiple Producers with Clone

The `tx` (transmitter) end of a channel can be `.clone()`d to give multiple producers, all sending to the same single receiver. Each clone is an independent sender; the channel stays open until *all* transmitter handles are dropped, at which point the receiver's iterator ends naturally.

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

let tx1 = tx.clone();
thread::spawn(move || { tx1.send("Message from thread 1").unwrap(); });

let tx2 = tx.clone();
thread::spawn(move || { tx2.send("Message from thread 2").unwrap(); });

thread::spawn(move || { tx.send("Message from thread 3").unwrap(); });

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

Multiple tx Clones

Each thread gets its own `tx` clone — no sharing required.

Single rx

One receiver collects all messages; channel closes when last sender drops.

Shared-State Concurrency — Mutex

A `Mutex<T>` wraps data with a mutual-exclusion lock. `.lock()` blocks until the lock is acquired and returns a `MutexGuard` smart pointer. When the guard goes out of scope, the lock is released automatically — you **cannot** forget to unlock it.

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

let m = Mutex::new(5);
{
  let mut num = m.lock().unwrap(); // Acquire lock
  *num = 6;
} // Lock released here — guard dropped

println!("m = {:?}", m); // Mutex { data: 6 }
```

Can't Access Without Locking

The data is hidden inside `Mutex<T>`. The only way to get a reference is to call `.lock()`, making accidental unsynchronised reads impossible.

Automatic Release

The RAII pattern means the lock is always released when the guard's scope ends — even in the presence of early returns or panics. No forgetting to unlock.

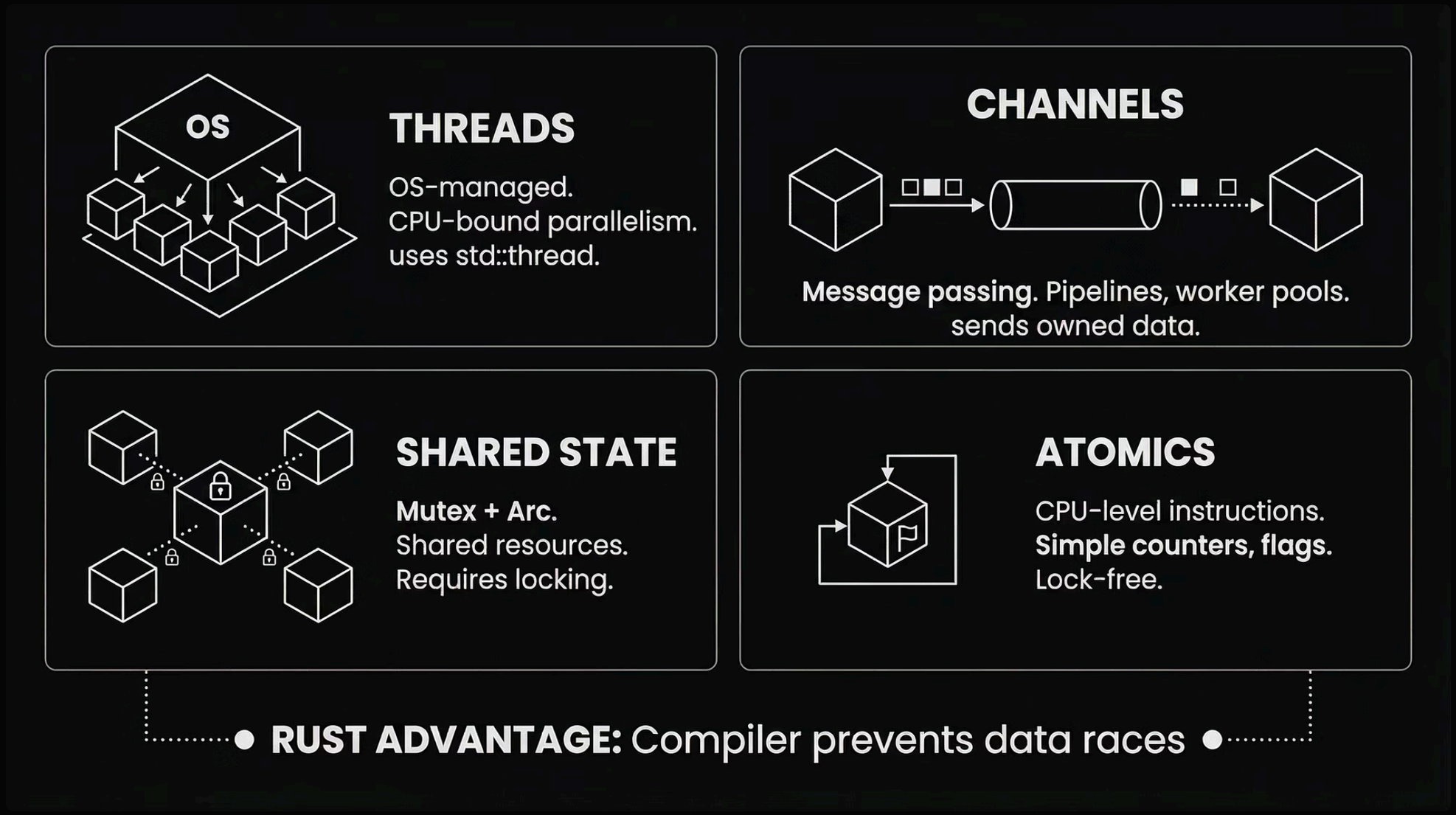
Sharing Mutex Across Threads with Arc

`Rc<T>` is not thread-safe because its reference counting uses non-atomic operations. `Arc<T>` (Atomic Reference Counting) uses CPU atomic instructions to manage the count safely across threads, at a small performance cost. The pattern `Arc<Mutex<T>>` is the idiomatic way to share mutable state across threads.

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

let counter = Arc::new(Mutex::new(0));
let mut handles = vec![];

for _ in 0..10 {
 let counter = Arc::clone(&counter); // Cheap clone — increments refcount
 let handle = thread::spawn(move || {
 let mut num = counter.lock().unwrap();
 *num += 1;
 }); // Lock released here
 handles.push(handle);
}

for handle in handles { handle.join().unwrap(); }
println!("Result: {}", *counter.lock().unwrap()); // Always 10
```

Comparing Concurrency Models

Rust doesn't prescribe a single concurrency style. You choose the model that fits your problem — and the compiler enforces safety in all of them.



Rust's advantage across all models: the `Send` and `Sync` marker traits ensure the compiler rejects any design that would allow a data race, regardless of which concurrency model you choose.

Send and Sync Traits

`Send` and `Sync` are **marker traits** — they have no methods. The compiler automatically derives them for most types and rejects any code that would transfer a non-`Send` type across a thread boundary or share a non-`Sync` reference. This makes the type system itself the concurrency safety enforcer.

Send — Transfer Across Threads

```
// i32, String, Mutex<T>, Arc<T> are Send
let x = 5;
thread::spawn(move || println!("{}", x)); // OK

// Rc<T> is NOT Send!
let rc = std::rc::Rc::new(5);
// thread::spawn(move || println!("{}", rc)); // ERROR
```

Sync — Share References Across Threads

```
// &T is Send when T: Sync
// Mutex<T> is Sync
// Cell / RefCell are NOT Sync
use std::cell::RefCell;
let cell = RefCell::new(5);
// thread::spawn(move || {
//   cell.borrow(); // ERROR: RefCell not Sync
// });
```

Auto-traits: `Send` and `Sync` are derived automatically by the compiler when all fields of a type are themselves `Send`/`Sync`. You almost never implement them manually.

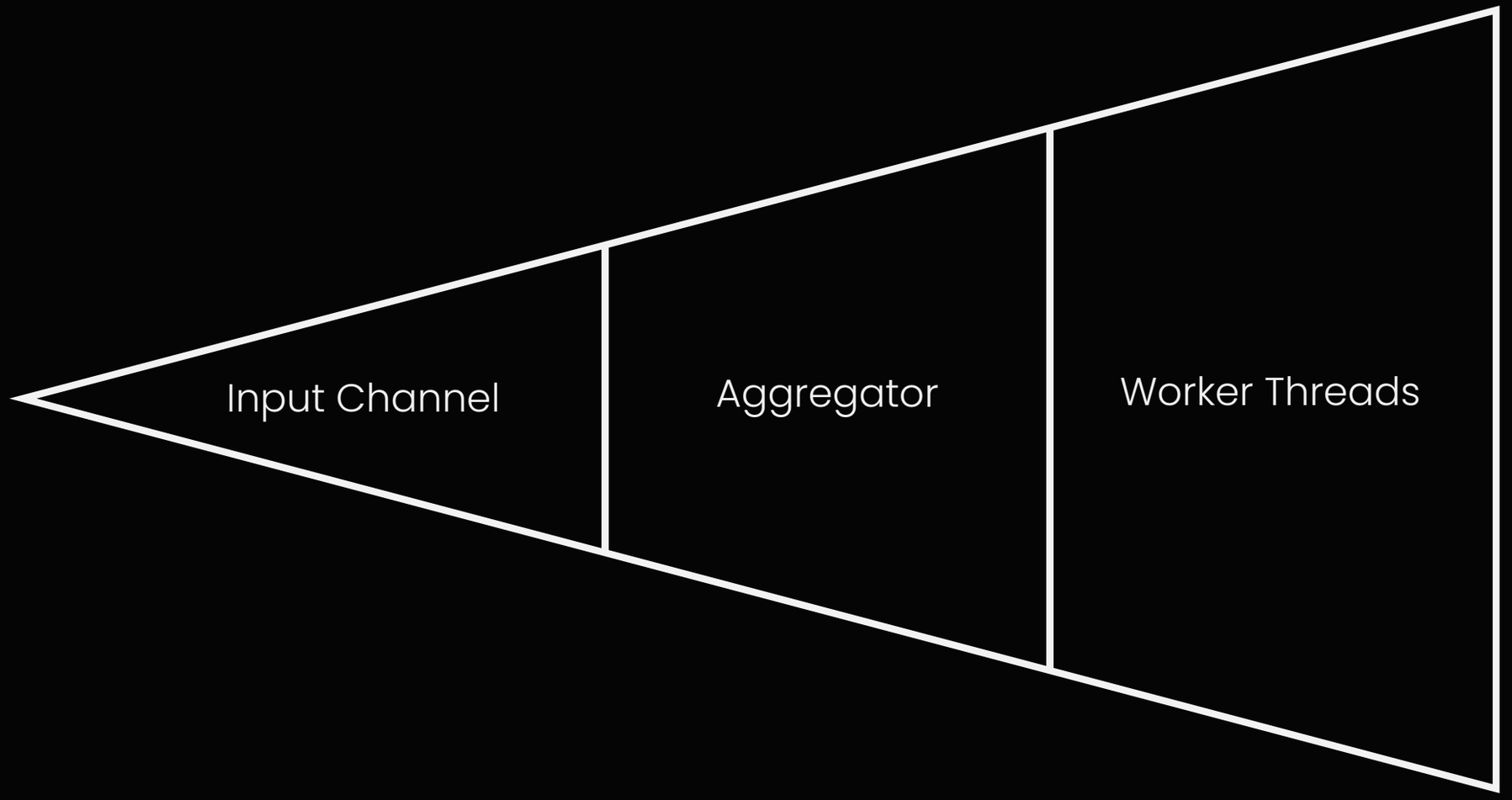
PART 3.4

Lab Session — Multi-threaded Data Processor

In this lab you will build a fully concurrent word-count processor. Multiple worker threads pull text lines from a shared channel, compute word counts (with a shared cache to avoid duplicate work), and forward results to an aggregator thread.

Lab Overview — Architecture

The system is composed of four logical parts that communicate via channels and share state through `Arc<Mutex<T>>`. Understanding this architecture before writing code helps you reason about ownership flow and synchronisation points.



The cache is wrapped in `Arc<Mutex<Cache>>` and shared by all workers. Lines already seen return a cached count without recomputation — saving redundant work when duplicate lines appear in the input.

Lab Setup — Data Structures

Before writing thread logic, define the message type and cache struct. Using an `enum` for worker messages is idiomatic: it lets you send both data payloads and control signals (`Shutdown`) through the same channel, keeping the protocol explicit and exhaustively matched.

Worker Message Enum

```
enum WorkerMessage {
  ProcessLine(String),
  Shutdown,
}
```

Sending `Shutdown` once per worker signals a clean exit without extra synchronisation primitives.

Cache Implementation

```
struct Cache {
 store: HashMap<String, usize>,
}
impl Cache {
 fn get_or_compute(&mut self, line: &str) -> usize {
 if let Some(&count) = self.store.get(line) {
 return count; // Cache hit!
 }
 let count = line.split_whitespace().count();
 self.store.insert(line.to_string(), count);
 count // Cache miss — computed & stored
 }
}
```

Worker Thread Implementation

Each worker shares the same locked receiver via `Arc<Mutex<Receiver>>`. It acquires the lock briefly to dequeue a message, then releases it before doing any work — a critical pattern to avoid blocking other workers during computation.

```
fn worker(
  id: usize,
  receiver: Arc<Mutex<mpsc::Receiver<WorkerMessage>>>,
  cache: Arc<Mutex<Cache>>,
  result_sender: mpsc::Sender<usize>,
) {
   loop {
     let message = {
       let rx = receiver.lock().unwrap();
       match rx.try_recv() {
         Ok(msg) => msg,
         Err(_) => { thread::sleep(Duration::from_millis(10)); continue; }
       }
     }; // Lock released here — before processing!

     match message {
       WorkerMessage::ProcessLine(line) => {
          let count = { cache.lock().unwrap().get_or_compute(&line) };
          result_sender.send(count).unwrap();
     }
     WorkerMessage::Shutdown => break,
   }
  }
}
```

Main Controller — Spawning Workers

The controller creates the shared state, spawns the worker pool, distributes work, and sends shutdown signals. Notice how `Arc::clone` is called *before* the `move` closure — each thread gets its own handle to the shared counter without copying the underlying data.

```
fn main() {
 let (work_tx, work_rx) = mpsc::channel();
 let (result_tx, result_rx) = mpsc::channel();

 let work_rx = Arc::new(Mutex::new(work_rx));
 let cache = Arc::new(Mutex::new(Cache::new()));

 let mut handles = vec![];
 for i in 0..3 {
 let work_rx = Arc::clone(&work_rx);
 let cache = Arc::clone(&cache);
 let result_tx = result_tx.clone();
 handles.push(thread::spawn(move || {
 worker(i, work_rx, cache, result_tx);
 }));
 }

 let lines = vec!["hello world", "rust is awesome",
 "hello world", "concurrency is fun"];
 for line in lines {
 work_tx.send(WorkerMessage::ProcessLine(line.into())).unwrap();
 }
 for _ in 0..3 { work_tx.send(WorkerMessage::Shutdown).unwrap(); }
}
```

Main Controller — Collecting Results

The aggregator runs on its own thread, receiving word counts until the channel closes. Dropping the original `result_tx` in the main thread is essential: the channel only closes when *all* senders are dropped. Without this, the aggregator thread would block forever waiting for more messages.

```
let result_handle = thread::spawn(move || {
let mut total = 0;
let mut received = 0;
while received < 4 {
match result_rx.recv_timeout(Duration::from_secs(1)) {
Ok(count) => {
total += count;
received += 1;
println!("Received: {} (running total: {})", count, total);
}
Err(_) => break,
}
}
total
});

for handle in handles { handle.join().unwrap(); }
drop(result_tx); // Close channel — signals aggregator to finish

let final_total = result_handle.join().unwrap();
println!("\nFinal word count total: {}", final_total);
```

Observing Cache Behaviour

After the run completes, inspecting the cache reveals exactly how many unique lines were processed and how many redundant computations were avoided. With two duplicate lines in the input, the cache should contain only unique entries — demonstrating the efficiency benefit clearly.

```
let cache = cache.lock().unwrap();
println!("\nCache contents:");
for (line, count) in cache.store.iter() {
  println!("  '{}': {} words", line, count);
}
// Expected output:
// Cache contents:
//   'hello world':      2 words  (computed once, reused once)
//   'rust is awesome':  3 words
//   'concurrency is fun': 3 words
```

Lines Processed

Total work units sent through the pipeline.

Unique Cache Entries

Unique lines stored — duplicates returned instantly from cache.

Cache Hits

Lines served from cache without recomputation.

Lab Challenge Tasks

Once the basic implementation is working, push further with structured challenge levels. Each level builds on the previous — work through them in order or jump straight to the level that matches your comfort with Rust concurrency.

🟢 LEVEL 1

Basic Implementation

- Get the complete system compiling and running
- Add proper error handling for all channel operations
- Print final statistics: total words, cache hit rate

🟡 LEVEL 2

Enhancements

- Add per-worker timeouts so stalled workers don't block the pool
- Implement dynamic worker pool sizing based on queue depth
- Report progress every N items processed

🔴 LEVEL 3

Advanced Features

- Implement an LRU cache eviction policy
- Add a priority queue for high-priority messages
- Graceful shutdown that drains pending work before exit
- Metrics collection: items/second, average latency

Common Pitfalls to Avoid

Even with Rust's safety guarantees, logical concurrency mistakes are still possible at runtime. The compiler prevents data races, but it cannot prevent deadlocks, lock contention, or poor performance caused by holding locks longer than necessary.

→ Pitfall 1 — Holding Locks Too Long

```
// BAD: Lock held during slow operation
let guard = data.lock().unwrap();
thread::sleep(Duration::from_secs(1)); // All other threads blocked!

// GOOD: Compute first, then lock briefly
let new_data = compute();
*data.lock().unwrap() = new_data;
```

→ Pitfall 2 — Deadlock via Lock Ordering

If thread A locks `mutex_a` then `mutex_b`, and thread B locks `mutex_b` then `mutex_a`, they can deadlock. **Always acquire multiple locks in the same order** across all threads.

→ Pitfall 3 — Forgetting to Clone Arc

Each thread needs its own `Arc` handle cloned *before* the `move` closure. Cloning inside the closure moves the original, leaving nothing for subsequent threads.

→ Pitfall 4 — Ignoring Channel Errors

Calling `.unwrap()` on `send()` is fine in examples, but production code should handle `SendError` — it signals the receiver has been dropped.

Pitfall Deep Dive — Deadlock Prevention

Deadlocks are particularly insidious: they only manifest under specific thread interleaving, making them hard to reproduce in tests. Rust cannot prevent them at compile time, but a consistent lock-ordering discipline eliminates the most common cause entirely.

Rule 1

Establish a global lock ordering and document it. All threads acquire locks in that order.

Rule 2

Keep critical sections as small as possible. Release locks before calling external code.

Rule 3

Consider `try_lock()` with a timeout for cases where strict ordering is impractical.

Module 3 — Key Takeaways

Module 3 covered a large surface area. Here is a consolidated view of the most important ideas to carry forward into your everyday Rust work.

1

Structs & Enums

Structs group related data; enums express mutually exclusive alternatives — often with data attached. `Option` and `Result` are just enums, yet they eliminate whole classes of bugs.

2

Collections

`String`, `Vec<T>`, and `HashMap<K,V>` cover 90% of everyday data-structure needs. Each enforces ownership — be deliberate about moves vs borrows.

3

Pattern Matching

Exhaustive `match`, concise `if let`, and looping `while let` form a complete, safe control-flow system. The compiler rejects non-exhaustive matches.

4

Closures & Iterators

Closures capture the environment; iterators are lazy. Chain adapters freely — the compiler optimises them to the equivalent hand-written loop.

5

Generics & Traits

Generics eliminate duplication at zero runtime cost. Traits define shared behaviour. Together they are the backbone of every Rust library.

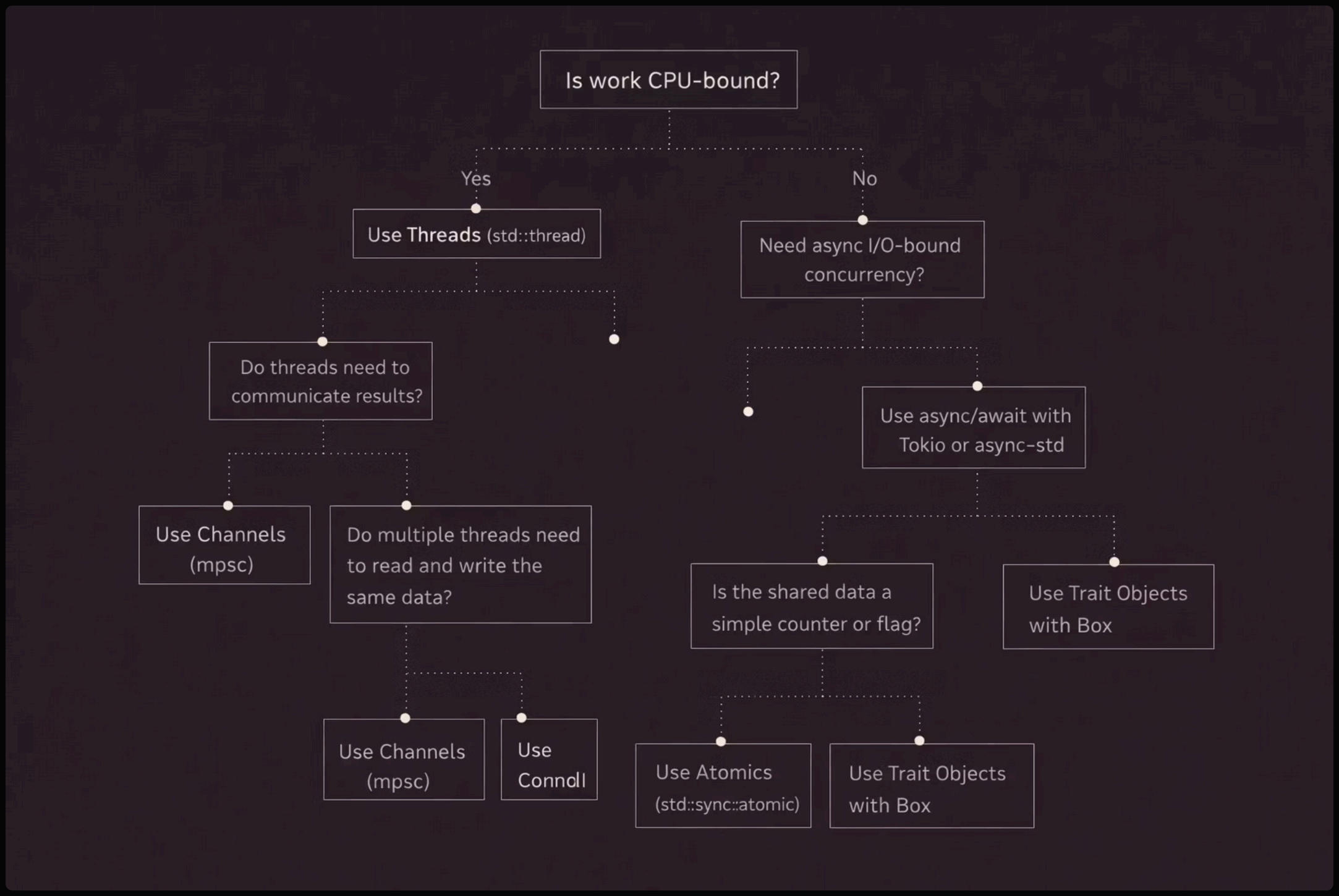
6

Fearless Concurrency

Ownership prevents data races. Channels pass ownership between threads. `Arc<Mutex<T>>` shares mutable state safely. `Send` and `Sync` enforce this in the type system.

Concurrency Model Decision Guide

Choosing the right concurrency primitive upfront avoids expensive refactoring later. Use this guide as a quick reference when starting a new concurrent component.



Note: async/await is covered in a later module. The primitives in this module — threads, channels, and `Arc<Mutex<T>>` — are the foundation on which async runtimes are built.

Preview of Module 4 — Traits & Error Handling

Module 4 builds directly on what you have learned here. Advanced trait features enable operator overloading and flexible library design. Lifetimes complete the ownership picture. Robust error handling with custom types and the `?` operator makes production-grade Rust genuinely ergonomic.

Advanced Traits

- Associated types
- Default generic type parameters
- Operator overloading via `std::ops`
- Newtype pattern

Lifetimes

- Validating references with lifetime annotations
- Lifetime elision rules
- Lifetime bounds on generic parameters
- Static lifetime

Error Handling

- Recoverable vs unrecoverable errors
- Custom error types
- Propagating errors with `?`
- The `thiserror` and `anyhow` crates

> "Building robust, production-ready applications" — Module 4 theme

Homework — Reinforce Your Learning

The best way to consolidate these concepts is to write code that breaks, then fix it by listening to the compiler. Each task below is designed to provoke a specific ownership or concurrency insight.

01

Complete the Lab

Finish all three challenge levels of the multi-threaded data processor. Reach Level 3 if you can — LRU cache implementation is a valuable exercise in combining `HashMap` with a doubly-linked structure.

02

Experiment with Concurrency Patterns

Re-implement the word-count processor using channels only (no `Mutex`) and compare the design. Then try the inverse: shared state only. Which feels more natural for this problem?

03

Build a Thread Pool

Implement a simple thread pool from scratch: a fixed set of worker threads that pull jobs from a shared queue. This is the canonical exercise for internalising `Arc<Mutex<T>>` and channel shutdown patterns.

04

Read Chapters 8, 13 & 16

Chapters 8 (Collections), 13 (Closures & Iterators), and 16 (Fearless Concurrency) of *The Rust Book* align directly with this module. Reading them cements compiler error intuitions.

Additional Resources

These resources are hand-picked for intermediate Rust learners focused on systems programming. Prioritise the official documentation and Jon Gjengset's content — both are exceptionally accurate and up to date.

📘 Official Documentation

The Rust Book

Chapters 8, 10, 13, 16 — collections, generics, closures, concurrency. Free online at *doc.rust-lang.org/book*.

Rust by Example

Executable examples covering generics, traits, and concurrency. Great for quick lookups alongside the Book.

std::sync docs

Full API reference for `Mutex`, `Arc`, `mpsc`, `RwLock`, and atomics at *doc.rust-lang.org/std/sync*.

📹 Talks & Practice

Jon Gjengset — Rust Concurrency Patterns

Long-form live coding session covering real-world concurrency patterns. Highly recommended for cementing channel and mutex intuitions.

Learn Rust With Too Many Linked Lists

A unique deep-dive that forces you to confront ownership at its most painful — building linked lists. Enlightening for understanding `Rc`, `RefCell`, and `Arc`.

Rustlings

Small, focused exercises covering every topic in this module. Run `cargo install rustlings` to get started immediately.

Quick Reference — Ownership in Concurrent Contexts

This reference card summarises the ownership rules that govern every concurrent pattern covered in Module 3. Keep it handy whilst working through the lab and homework exercises.

Situation	Tool	Ownership Rule
Spawn a thread with data	`move` closure	Data is **moved** into the thread. Cannot use original binding afterwards.
Send value to another thread	`mpsc::channel`	`send()` **moves** the value. Sender can no longer use it.
Share read-only data	`Arc<T>`	Clone the `Arc` — increments reference count. Data is immutable.
Share mutable data	`Arc<Mutex<T>>`	Lock to get exclusive access. Lock released when guard drops.
Simple counter / flag	`AtomicUsize` etc.	CPU atomic operations. No lock required. `Ordering` must be specified.
Multiple senders	`tx.clone()`	Each sender holds an independent clone. Channel closes when all senders drop.

Q&A — Frequently Asked Questions

These are the questions that come up most often at this point in the course. If you have others, bring them to the session or post in the course forum.

Struct vs Enum — when do I choose?

Use a **struct** when all fields always exist together. Use an **enum** when a value can be one of several distinct shapes — especially when different variants carry different data.

Channels vs Arc<Mutex<T>>?

Channels model *communication* — transfer ownership of data between threads. `Arc<Mutex<T>>` models *shared state* — multiple threads operate on the same data. When in doubt, start with channels; they are easier to reason about.

Closures vs named functions?

Prefer closures for short, inline callbacks and when you need to capture the environment. Use named functions for reusable logic, recursive calls, or when the type needs to appear in a public API.

How do I prevent deadlocks in practice?

Always acquire multiple locks in the same global order. Keep critical sections short. Prefer one lock per resource. Use `try_lock()` with timeouts when strict ordering is difficult to guarantee.

Thank You

Module 3 has taken you from foundational data modelling with structs and enums, through expressive functional tools, all the way to writing safe, concurrent systems code that the compiler guarantees is free of data races.

Next Session

Module 4 — Advanced Traits, Lifetimes & Error Handling

Contact

Chandrashekar Babu
training@chandrashekar.info

Motto

"Fearless concurrency means the compiler has your back."