Rust Intermediate

A Complete 7-Chapter Programming Course

Topics covered:

Lifetimes & Traits · Generics & Monomorphization

Smart Pointers (Box, Rc, RefCell) · Concurrency

Modules & Crates · Testing in Rust

Exercises: 21 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · framed throughout against Go

Course 2 of 3 · Advanced course follows

Table of Contents

CHAPTER 1 OF 7

Lifetimes

COURSE 2 · CH 1

Lifetimes

Giving the borrow checker a hint when a reference's relationship to its data isn't obvious from the code alone

Chapter 4 of Course 1 established that the borrow checker verifies references never outlive their data. Most of the time it does this invisibly. Occasionally — when a function's signature involves multiple references whose relationship to each other isn't clear from the code alone — the compiler needs an explicit hint. That hint is a **lifetime annotation**.

Why the Borrow Checker Sometimes Needs Help

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() { x } else { y }
} // compile error: missing lifetime specifier
```

This won't compile as written. The compiler can't determine which input's lifetime the returned reference should be tied to — it depends on which branch actually runs, and that's a runtime decision the compiler can't resolve at compile time without more information.

Lifetime Annotation Syntax

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}

let s1 = String::from("long string");
let s2 = String::from("short");
let result = longest(s1.as_str(), s2.as_str());
```

`'a` — an apostrophe plus a name, conventionally a single lowercase letter — is a **lifetime parameter**. This is a common early misconception worth stating plainly: lifetime annotations don't *change* how long anything

lives. They *describe* a relationship that already exists in the code — here, that the returned reference is valid for at most as long as the **shorter** of `x` and `y`'s actual lifetimes.

Lifetime Elision Rules

Most functions with references never need explicit lifetime annotations at all — the compiler applies three **elision rules** automatically:

1. Each reference parameter gets its own lifetime.
2. If there's exactly **one** input lifetime, it's assigned to every output lifetime.
3. If one parameter is `&self` or `&mut self` (a method), `self`'s lifetime is assigned to every output.

```
impl Sentence {
    fn first_word(&self) -> &str {  // no explicit lifetime needed — rule 3 applies
        self.text.split_whitespace().next().unwrap_or("")
    }
}
```

This is exactly why every `&self` method returning a reference back in Course 1 (Chapter 5) never needed an explicit lifetime — elision was silently handling it via rule 3 the entire time.

Struct Lifetimes

A struct that **holds** a reference, rather than an owned value, needs its own lifetime parameter:

```
struct Excerpt<'a> {
    part: &'a str,
}
```

This ties the struct instance's own validity to the reference it holds — if the referenced data were dropped before the struct itself, that's a compile error, exactly Course 1 Chapter 4's dangling-reference protection, now extended to cover structs.

The 'static Lifetime

`'static` is a special, reserved lifetime meaning "valid for the entire duration of the program." String literals are `&'static str`, since they're baked directly into the compiled binary and never get dropped.

SITUATION	ANNOTATION NEEDED?
One input reference, one output reference	`No — elision rule 2 handles it`
&self method returning a reference	`No — elision rule 3 handles it`
Multiple input references feeding one output	`Yes — the compiler can't infer the relationship`
A struct holding a reference	`Yes — the struct needs its own lifetime parameter`

Lifetime Annotation ('a)

Describes an existing relationship between references' validity — never changes how long anything actually lives.

Elision Rules

Three automatic rules that cover the vast majority of real functions with no annotation required at all.

Struct Lifetimes

A struct storing a reference needs a lifetime parameter, tying the struct's validity to the data it borrows.

'static

Valid for the whole program's duration — string literals are the most common example.

ZERO RUNTIME COST — AGAIN

Lifetimes are erased entirely at compile time, exactly like the generic types Course 3 will cover — they exist purely for the compiler's own analysis and leave nothing behind in the compiled binary. This is another concrete piece of the same story Course 1 Chapter 4 told about the borrow checker itself: real safety analysis, zero runtime cost.

'STATIC IS ALMOST NEVER THE RIGHT QUICK FIX

It's tempting, when a lifetime error appears, to slap `'static` on everything until the compiler stops complaining. This almost never actually solves the underlying ownership problem — it just forces data to live forever, which is often not what's actually wanted, and sometimes isn't even possible for the data in question. A `'static` requirement appearing where it doesn't obviously belong is usually a sign the real fix lies elsewhere — often in how ownership is structured, not in the lifetime annotation itself.

Coding Challenges

CHALLENGE 1

Write a function shortest<'a>(x: &'a str, y: &'a str) -> &'a str returning whichever string slice is shorter, following this chapter's longest example as a template.

View solution

CHALLENGE 2

Define a struct FirstLine<'a> holding a single field line: &'a str, and a method fn announce(&self) -> &str that returns self.line. Explain, citing the specific elision rule, why announce needs no explicit lifetime annotation even though the struct itself does.

View solution

CHALLENGE 3

Explain why changing a function's return type from &'a str to &'static str to silence a lifetime error is usually the wrong fix — describe what actually goes wrong for the caller if the underlying data genuinely isn't valid for the whole program's lifetime.

View solution

CHAPTER 1 QUICK REFERENCE

- **'a** — a lifetime parameter; describes an existing relationship between references, never changes actual lifetimes
- **Elision rule 1:** each reference parameter gets its own lifetime
- **Elision rule 2:** exactly one input lifetime → assigned to all outputs
- **Elision rule 3:** a &self/&mut self parameter's lifetime → assigned to all outputs
- **struct Name<'a> { field: &'a T }** — a struct holding a reference needs its own lifetime parameter
- **'static** — valid for the whole program's duration; string literals are &'static str
- Reaching for 'static to silence an error usually hides a real ownership-structure problem instead of fixing it
- Lifetimes are erased at compile time — zero runtime cost, purely a compiler-analysis tool
- **Next chapter:** Traits — trait definitions, default implementations, trait bounds, and a contrast with Go's interfaces

CHAPTER 2 OF 7

Traits

COURSE 2 · CH 2

Traits

Describing shared behavior across different types — Rust's answer to Go's interfaces

Course 1 built individual types with their own methods. This chapter introduces the tool for describing **shared behavior across different types** — a **trait**, directly comparable to Go's own interfaces (`go2-2`).

Defining and Implementing a Trait

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

impl Summary for Article {
    fn summarize(&self) -> String {
        format!("{} by {}", self.title, self.author)
    }
}

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

A trait declares a method signature; a type opts in by writing its own `impl Trait for Type` block — `Article` and `Tweet` implement `summarize` completely differently.

Default Implementations

A trait method can include a default body — implementing types may use it as-is, or override it:

```rust
trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}

impl Summary for Notice {} // empty impl — uses the default
impl Summary for Article {
    fn summarize(&self) -> String { /* overrides the default */ }
}
```

Traits as Parameters (impl Trait)

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

`notify` accepts *any* type implementing `Summary` — the exact same "program to an interface, not an implementation" idea Go's interfaces exist for.

Trait Bounds

`impl Trait` is sugar for a more general, more powerful syntax — **trait bounds** — needed when a constraint can't be expressed with the sugar alone. Two parameters that must be the *same* concrete type is exactly that case:

```rust
fn compare<T: Summary>(a: &T, b: &T) -> bool {
    a.summarize() == b.summarize()
}

// Multiple bounds with +:
fn notify_and_log<T: Summary + std::fmt::Display>(item: &T) { /* ... */ }

// A where clause for readability when bounds get long:
fn process<T, U>(t: &T, u: &U)
where
    T: Summary,
    U: Clone,
{ /* ... */ }
```

`impl Trait` can't express "these two parameters must be the same type" — only a shared generic parameter with a trait bound can.

Contrast With Go's Interfaces

Go interfaces are satisfied **implicitly** — structural typing means if a type happens to have the right methods, it satisfies the interface automatically, with no explicit declaration anywhere. Rust traits are **explicit**: a type must write `impl Trait for Type` even if it already happens to have a method with a matching signature.

	GO	RUST
Interface/trait satisfaction	`Implicit — structural typing`	`Explicit — impl Trait for Type required`
Accidental satisfaction	`Possible — matching methods are enough`	`Impossible — must be declared on purpose`

Rust's explicitness prevents a type from accidentally satisfying a trait it was never intended to, at the cost of one extra `impl` block.

trait Definition

Declares method signatures a type can opt into implementing.

Default Implementation

A body a trait method already has; implementers may use it or override it.

impl Trait Parameter

Accept any type implementing a trait — sugar for the more general trait-bound syntax.

Trait Bound (<T: Trait>)

Needed when multiple parameters must share the exact same concrete type.

DEFAULT IMPLEMENTATIONS LET A TRAIT EVOLVE SAFELY

Adding a new method to a trait, with a default implementation, doesn't break any existing implementer — they simply inherit the default. Adding a new method with **no** default would instead be a compile error for every existing `impl` block, demanding they implement it too — the same "compiler tells you everywhere that needs updating" theme from Course 1's enum-variant gotcha, just showing up here as the case a default deliberately avoids.

THE ORPHAN RULE: YOU CAN'T IMPL JUST ANYTHING FOR ANYTHING

Rust requires that **either** the trait **or** the type being implemented is defined in your own crate — you can't write `impl SomeExternalTrait for SomeExternalType` when neither belongs to you. This exists specifically to prevent two different crates from providing conflicting implementations of the same trait for the same type, which would make it genuinely ambiguous which one applies. It's a real, sometimes-surprising restriction for newcomers coming from more permissive languages — but it's protecting a real guarantee (coherence), not an arbitrary limitation.

Coding Challenges

CHALLENGE 1

Define a trait Describable with a method describe(&self) -> String that has a default implementation returning "No description available.". Implement it for a struct Book (overriding the default) and a struct Widget (using the default, via an empty impl block).

View solution

CHALLENGE 2

Write a function print_description(item: &impl Describable) using Challenge 1's trait, then explain why this same function could NOT be used to compare two Describable items of potentially different concrete types for equality, and what syntax change would be needed to require them to be the same type.

View solution

CHALLENGE 3

Explain, in your own words, why Go's implicit interface satisfaction could lead to a type accidentally satisfying an interface it wasn't designed for, and why Rust's explicit impl Trait for Type requirement makes that specific mistake impossible.

View solution

CHAPTER 2 QUICK REFERENCE

- **trait Name { fn method(&self) -> T; }** — declares shared behavior; types opt in via impl Trait for Type
- **Default implementations** — a trait method can have a body; implementers may use it or override it
- **fn f(x: &impl Trait)** — accepts any type implementing Trait; sugar for a trait bound
- **fn f<T: Trait>(a: &T, b: &T)** — a trait bound; needed when parameters must share the same concrete type
- **T: TraitA + TraitB** — multiple bounds; **where** clauses improve readability for longer bound lists
- Go interfaces are satisfied implicitly (structural); Rust traits require an explicit impl block — no accidental satisfaction
- Adding a defaulted trait method doesn't break existing implementers; adding one with no default does
- The orphan rule: either the trait or the type must be local to your own crate
- **Next chapter:** Generics — generic functions/structs, trait bounds on generics, and a contrast with Go's own generics

CHAPTER 3 OF 7

Generics

COURSE 2 · CH 3

Generics

Writing code once, over many types — and why you've already been using generics since Course 1

Chapter 2 introduced trait bounds as a way to constrain a generic parameter. This chapter is fully about generics themselves — writing logic once instead of duplicating it per type, with trait bounds as the way to say what that type must be able to *do*.

The Problem Generics Solve

Without generics, comparing the largest value in a list would need a separate, identical function per type — `largest_i32`, `largest_f64`, `largest_char`. Generics let that logic exist once:

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

The trait bounds aren't decoration — they're required by what the function's body actually does: `PartialOrd` because `>` needs to be defined for `T`; `Copy` because a value can't be moved out of a slice reference, only copied.

Generic Structs

```rust
struct Point<T> {
    x: T,
    y: T,
}

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

Both fields share type `T` here. When fields genuinely need to differ, use multiple type parameters: `struct Point<T, U> { x: T, y: U }`.

Generic Enums: You've Already Been Using These

Course 1's `Option<T>` (Chapter 6) and `Result<T, E>` (Chapter 7) are themselves generic enums — you've been using generics since Chapter 6, just without the formal vocabulary for it until now.

Monomorphization: How Generics Compile

At compile time, Rust generates a separate, fully concrete version of a generic function or struct for *every* distinct type it's actually used with — calling `largest` with a slice of `i32` and again with a slice of `f64` produces two entirely separate compiled functions internally. This means generics carry **zero runtime cost** — no dynamic dispatch, no boxing — the exact same "real analysis, zero runtime cost" story Course 1 told about the borrow checker and lifetimes, now extended to generics.

Contrast With Go's Generics (go3-1)

Go added generics far later than Rust (Go 1.18, 2022), using comparable constraint syntax (`[T any]`). The real engineering difference: Rust **always** fully monomorphizes every generic instantiation. Go's compiler, to control binary size and compile time, has used a mix of monomorphization and shared/dictionary-based code generation for certain cases — meaning not every Go generic instantiation gets its own fully specialized compiled copy the way Rust's always does. This is a genuine, deliberate trade-off both languages made — Go leaning toward smaller binaries and faster compiles in some cases, Rust always prioritizing runtime performance — not a case of one being straightforwardly better.

	GO	RUST
Generics added	2022 (Go 1.18)	Present since Rust 1.0

	GO	RUST
Compilation strategy	`Mix of monomorphization and shared/dictionary code`	`Always fully monomorphized`
Optimizes for	`Binary size / compile speed in some cases`	`Runtime performance, always`

Generic Function

One implementation, parameterized over T, with trait bounds specifying what T must support.

Generic Struct

Fields typed by one or more type parameters, e.g. Point<T> or Point<T, U>.

Option<T> / Result<T, E>

Generic enums you've already used since Course 1 — now named and understood properly.

Monomorphization

A separate compiled version generated per concrete type — zero runtime cost, larger binary.

YOU ALREADY KNOW GENERICS

Every time `Some(value)`, `None`, `Ok(value)`, or `Err(error)` appeared back in Course 1, that was a generic type already at work. This chapter didn't introduce a new concept from scratch — it gave a name and a formal mechanism to something already familiar.

FORGETTING THE BOUND AN OPERATION ACTUALLY NEEDS

`fn largest<T>(list: &[T]) -> T` with a body that uses `>` internally, but **no** `T: PartialOrd` bound, produces a compile error saying `>` can't be applied to type `T`. The function looks like it should "just work for any T" — but the compiler has no way to know that unless the bound says so explicitly. Whatever an operation inside a generic function actually needs (comparison, cloning, formatting), the corresponding trait has to be named in the bound.

Coding Challenges

CHALLENGE 1

Write a generic function smallest(list: &[T]) -> T, following this chapter's largest as a template, and call it with both a slice of i32 and a slice of char.

View solution

CHALLENGE 2

Define a generic struct Pair with fields first: T and second: U, and an impl block with a method describe(&self) -> String requiring T: std::fmt::Display and U: std::fmt::Display. Explain why the bound has to go on the impl block (or the method) rather than the struct definition itself.

View solution

CHALLENGE 3

Explain what "zero runtime cost" actually means for monomorphized generics, referencing what the compiler produces for two different calls to the same generic function with different types — and contrast this briefly with what a dynamic-dispatch-based alternative would cost at runtime instead.

View solution

CHAPTER 3 QUICK REFERENCE

- **fn f<T: Bound>(x: T)** — one implementation over many types, constrained by what the body actually needs
- **struct Name<T> { field: T }** — a generic struct; use multiple parameters (T, U) when fields differ in type
- **Option<T>** and **Result<T, E>** are themselves generic enums, in use since Course 1
- **Monomorphization** — the compiler generates a fully separate version per concrete type used; zero runtime cost, larger binary
- Go's generics (added 2022) sometimes use shared/dictionary code instead of always monomorphizing — a deliberate binary-size/compile-speed trade-off, not a worse design
- A generic function's trait bounds must name whatever the body's operations actually require — the compiler won't infer this from "it should just work"
- **Next chapter:** Smart Pointers — Box<T>, Rc<T>, RefCell<T>, and interior mutability

CHAPTER 4 OF 7

Smart Pointers

COURSE 2 · CH 4

Smart Pointers

Relocating, not abandoning, Course 1's ownership rules for the cases that need more flexibility

Course 1's ownership rules — exactly one owner, borrow checking enforced entirely at compile time — are strict by design. Some real, legitimate patterns genuinely don't fit that shape. Smart pointers are Rust's controlled, still-safe way to get flexibility where it's actually needed, without abandoning safety altogether.

Box<T>: Heap Allocation With a Single Owner

The simplest smart pointer: puts a value on the heap instead of the stack, while still following ordinary single-ownership rules exactly — `Box<T>` itself *is* the single owner, moving per Course 1 Chapter 3's rules like anything else. The classic use case is a **recursive type**:

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}
```

Without `Box`, `Cons(i32, List)` fails to compile — the compiler needs to know a fixed size for `List` at compile time, but a `List` containing a `List` containing a `List`... has no fixed size at all. `Box`'s size is always just one pointer, regardless of what it points to, breaking the infinite recursion.

Rc<T>: Multiple Owners via Reference Counting

Course 1's rule was exactly *one* owner. `Rc<T>` ("Reference Counted") relaxes this specifically for cases where multiple parts of a program genuinely need to jointly own the same data:

```
use std::rc::Rc;

let a = Rc::new(5);
let b = Rc::clone(&a);  // increments the count — no deep copy
println!("count: {}", Rc::strong_count(&a));  // 2
```

`Rc::clone` doesn't copy the underlying data — it increments a reference count. The data is only actually dropped once the *last* `Rc` pointing to it goes out of scope, bringing the count to zero.

RefCell<T> and Interior Mutability

The borrow checker's rules (Course 1, Chapter 4) are enforced at **compile time** by default. `RefCell<T>` moves that same rule — one mutable OR many immutable borrows — to **runtime** instead, allowing mutation even through an immutable reference to the `RefCell` itself:

```
use std::cell::RefCell;

let value = RefCell::new(5);
*value.borrow_mut() += 1;
println!("{}", value.borrow());  // 6

// Violating the rule at runtime — this PANICS instead of failing to compile:
// let _b1 = value.borrow_mut();
// let _b2 = value.borrow_mut(); // thread panicked: already borrowed
```

A genuine trade-off: more flexibility, but a violated rule moves from a compile error to a runtime panic.

Combining Rc<T> and RefCell<T>

The common, idiomatic combination — `Rc<RefCell<T>>` — gives multiple owners that can each mutate the shared data. This is Rust's deliberate, opt-in answer to a pattern that's simply the default in a garbage-collected language (shared mutable state) — not worse, just something you have to explicitly ask for rather than get for free:

```
let shared = Rc::new(RefCell::new(0));

let owner_a = Rc::clone(&shared);
let owner_b = Rc::clone(&shared);

*owner_a.borrow_mut() += 10;
*owner_b.borrow_mut() += 5;

println!("{}", shared.borrow());  // 15 — both owners mutated the SAME underlying data
```

	COMPILE-TIME BORROW CHECKING (CH.4, COURSE 1)	REFCELL<T> (RUNTIME)
When rules are checked	**At compile time**	**At runtime, on each borrow()/borrow_mut() call**

	COMPILE-TIME BORROW CHECKING (CH.4, COURSE 1)	REFCELL<T> (RUNTIME)
Violating the rule	`Compile error`	`Panic`
Flexibility	`Less — some valid patterns rejected`	`More — patterns the compiler can't verify statically still work`

Box<T>

Single ownership, heap-allocated — no rule relaxation, just placement. Needed for recursive types.

Rc<T>

Relaxes single ownership via reference counting — multiple joint owners of the same data.

RefCell<T>

Relaxes compile-time borrow checking to runtime — mutation through an immutable reference.

Rc<RefCell<T>>

Both relaxations combined — multiple owners, each able to mutate the shared data.

NOT ABANDONING THE RULES — RELOCATING THEM

Course 1 established that ownership and borrowing are checked strictly, at compile time. Smart pointers don't throw that away: `Box` changes nothing about the rules, only where data lives; `Rc` relaxes single-ownership via counting; `RefCell` relaxes borrow-checking from compile time to runtime. Every one of them is a deliberate, scoped exception — not a general escape hatch from Rust's safety model.

RC<T> ALONE DOES NOT ALLOW MUTATION

A common early mix-up: assuming `Rc<T>` by itself is "the mutable shared pointer." It isn't — `Rc<T>`'s data is immutable by default, which is exactly why it's so often paired with `RefCell<T>` specifically for the mutable case. Also worth flagging ahead: `Rc<T>` is **not** thread-safe — `Arc<T>` (the atomic version) is what Chapter 5's concurrency material uses instead.

Coding Challenges

CHALLENGE 1

Explain why enum Tree { Leaf(i32), Node(Box, Box) } compiles, but enum Tree { Leaf(i32), Node(Tree, Tree) } (without Box) does not — referencing this chapter's List example.

View solution

CHALLENGE 2

Write code creating an Rc, cloning it twice (three total owners), printing the strong count after each clone, then dropping one clone explicitly with drop() and printing the count once more.

View solution

CHALLENGE 3

Using Rc>>, write code where two separate "owner" variables each push a different number onto the same shared vector, then print the vector's final contents from a third owner to prove all three see the same underlying data.

View solution

CHAPTER 4 QUICK REFERENCE

- **Box<T>** — single-owner heap allocation; needed for recursive types (fixed-size pointer breaks infinite size)
- **Rc<T>** — multiple owners via reference counting; Rc::clone increments the count, no deep copy
- **Rc::strong_count(&x)** — inspect how many owners currently exist
- **RefCell<T>** — moves borrow checking to runtime; violating the rule panics instead of failing to compile
- **Rc<RefCell<T>>** — the idiomatic combo: multiple owners, each able to mutate shared data
- Rc<T> alone is immutable — RefCell is what actually enables mutation through it
- Rc<T> is not thread-safe — Arc<T> is the concurrency-safe equivalent, coming next chapter
- **Next chapter:** Concurrency — threads, Arc<Mutex<T>>, mpsc channels, and "fearless concurrency" contrasted with Go's goroutines

CHAPTER 5 OF 7

Concurrency

COURSE 2 · CH 5

Concurrency

Where Chapter 4's forward pointer to Arc<T> pays off — "fearless concurrency," made concrete

Chapter 4 closed with a forward pointer to `Arc<T>` for thread-safety. This chapter is where that payoff lands: the same ownership and borrowing rules that prevented aliasing bugs in single-threaded code (Course 1, Chapter 4) also prevent data races across threads — enforced at compile time, not discovered at runtime.

Spawning Threads

```
use std::thread;

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

println!("Hello from main!");
handle.join().unwrap();  // waits for the spawned thread to finish
```

`thread::spawn` returns a `JoinHandle`; `.join()` blocks until that thread completes.

The move Keyword and Ownership Across Threads

A spawned thread might genuinely outlive the function that spawned it — the compiler can't assume it won't. A closure that only *borrows* data risks that data being dropped before the thread finishes using it, so `thread::spawn` typically requires a `move` closure, forcing ownership of captured variables into the thread itself:

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

// Without `move`, this often fails to compile — the closure only
// borrows `data`, and the compiler can't guarantee it outlives the thread.
let handle = thread::spawn(move || {
    println!("{:?}", data);  // data is now OWNED by this closure
});
handle.join().unwrap();
```

This is Course 1's move semantics (Chapter 3) and borrow-checker lifetime reasoning (Chapter 4), both directly enforced here — applied to a genuinely new context where a thread's actual lifetime is unpredictable.

Sharing Data Safely with Arc<Mutex<T>>

Chapter 4's `Rc<RefCell<T>>` pattern has a thread-safe sibling: `Arc<T>` ("Atomic Reference Counted") replaces `Rc`, and `Mutex<T>` replaces `RefCell` — using a real OS-level lock, where `.lock()` returns a guard that unlocks automatically when dropped:

```
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);
    handles.push(thread::spawn(move || {
        let mut num = counter.lock().unwrap();
        *num += 1;
    }));
}
for h in handles { h.join().unwrap(); }
println!("Result: {}", *counter.lock().unwrap());  // 10
```

mpsc Channels: Message Passing

`std::sync::mpsc::channel()` ("multiple producer, single consumer") returns a `(Sender, Receiver)` pair — threads communicate by sending **owned values** rather than sharing memory directly:

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

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

thread::spawn(move || {
    tx.send(String::from("hello from the thread")).unwrap();
});

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

This is a direct expression of a proverb closely associated with Go's own community: "*Do not communicate by sharing memory; instead, share memory by communicating.*" Rust and Go arrive at genuinely similar channel-based thinking here.

Contrast With Go's Goroutines & Channels (go2-3)

Go's goroutines are lightweight, green-thread-style tasks managed by Go's own runtime scheduler — many goroutines multiplex onto relatively few real OS threads. Rust's `thread::spawn` creates real OS threads directly — heavier per-thread, with no separate runtime scheduler required (Rust's own lightweight async tasks are Course 3's territory, a different mechanism entirely). Go's channels are first-class language syntax (`ch <- value`, `<-ch`); Rust's `mpsc` channels are an ordinary standard-library type, no special syntax needed.

The more important contrast: Go's concurrency safety net is **best-effort** — its race detector catches *some* data races, opt-in, typically during testing. Rust's guarantee is enforced by the **compiler itself**, for every concurrent program, as a hard requirement just to compile at all.

	GO	RUST
Thread model	Lightweight goroutines, runtime-scheduled	Real OS threads (async tasks are separate, Course 3)
Channels	First-class language syntax	A standard-library type (mpsc)
Data-race safety	Best-effort — race detector, opt-in/testing-time	Compiler-enforced — required to compile at all

thread::spawn / join

Creates a real OS thread; join() blocks until it finishes.

move Closures

Forces ownership transfer into the thread — required since a thread's lifetime is unpredictable.

Arc<Mutex<T>>

The thread-safe sibling of Rc<RefCell<T>> — shared, lockable mutable state.

mpsc Channels

Send owned values between threads instead of sharing memory directly.

THE ACTUAL MECHANISM BEHIND "FEARLESS CONCURRENCY"

Two marker traits, `Send` (safe to move to another thread) and `Sync` (safe to share a reference across threads), are what the compiler actually checks. Trying to share a plain `Rc<T>` — not thread-safe, per Chapter 4's warning — across threads is a **compile error**, not a data race waiting to happen at runtime. This is the concrete mechanism behind Rust's "fearless concurrency" claim: it isn't a slogan, it's `Send` / `Sync` doing real, checked work.

MUTEX<T> PREVENTS DATA RACES, NOT DEADLOCKS

Two threads, each holding one lock and waiting for the other's lock, will deadlock — and Rust's compiler does **not** catch this. The compile-time guarantee is specifically about memory safety and data races, not general concurrency correctness. A deadlock is a real, possible runtime bug that the type system has no visibility into at all — an honest limit on what "fearless concurrency" actually promises.

Coding Challenges

CHALLENGE 1

Write code that spawns a thread using a move closure to print a Vec created in main, then explain what compiler error would occur (in general terms) if move were removed.

View solution

CHALLENGE 2

Using Arc>>, spawn 5 threads that each push their own thread number (0-4) onto a shared vector, join all threads, then print the vector's final length and confirm it's 5.

View solution

CHALLENGE 3

Explain why Rust's compile-time thread safety is a stronger guarantee than Go's race detector, but is still not a complete guarantee of concurrency correctness — referencing deadlocks specifically as an example of what it doesn't cover.

View solution

CHAPTER 5 QUICK REFERENCE

- **thread::spawn(|| {...})** — creates a real OS thread; returns a JoinHandle; **.join()** waits for completion
- **move closures** — required to transfer ownership into a thread, since its lifetime is unpredictable
- **Arc<T>** — thread-safe Rc; **Mutex<T>** — thread-safe RefCell, using a real OS lock
- **Arc<Mutex<T>>** — the multi-threaded analog of Rc<RefCell<T>>
- **mpsc::channel()** — a (Sender, Receiver) pair; send owned values instead of sharing memory
- **Send/Sync** — the marker traits the compiler checks to reject unsafe cross-thread sharing at compile time
- Go's race safety is best-effort tooling; Rust's is a hard, compiler-enforced compile requirement
- Mutex<T> prevents data races, not deadlocks — a real, uncaught category of concurrency bug remains possible
- **Next chapter:** Modules & Crates — the module system, Cargo.toml, workspaces, and publishing to crates.io

CHAPTER 6 OF 7

Modules & Crates

COURSE 2 · CH 6

Modules & Crates

Organizing a real, growing project — from a single file to a published crate

Course 1 stayed within single small files. This chapter covers organizing a real Rust project as it grows: modules within a crate, `Cargo.toml` in depth, workspaces for multi-crate projects, and finally publishing a crate to crates.io.

The Module System

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

front_of_house::hosting::add_to_waitlist();
```

`mod` organizes code into a tree of namespaces within a single crate. Everything is **private by default** — a genuinely different default from many languages. `pub` makes an item visible to its *parent* module — not globally, a common early misconception. `pub(crate)` exposes an item crate-wide, but not to external code.

Splitting Modules Across Files

A module can live in its own file: `mod front_of_house;` (no body) tells Rust to look for the module's contents elsewhere. The modern convention (2018 edition onward) needs no `mod.rs` file:

```
src/
  lib.rs                    // mod front_of_house;
  front_of_house.rs          // pub mod hosting;
  front_of_house/
    hosting.rs              // pub fn add_to_waitlist() {}
```

use and Bringing Paths Into Scope

```
use crate::front_of_house::hosting;
use std::collections::HashMap as Map;  // renaming to avoid a clash

pub use crate::front_of_house::hosting::add_to_waitlist;  // re-export at the crate root
```

`use` shortens repeated long paths — already in constant use since Course 1's `use std::collections::HashMap` (Chapter 8). `pub use` re-exports an item, making something defined deep in the module tree part of the crate's own public API surface at a shallower, more convenient path.

Cargo.toml in Depth

```
[package]
name = "my_crate"
version = "0.1.0"
edition = "2021"

[dependencies]
serde = "1.0"   # caret by default: allows 1.x, never 2.0
```

`Cargo.lock` pins the exact, fully-resolved dependency versions actually used — directly analogous to npm's `package-lock.json`. Committing it for a binary/application ensures reproducible builds across machines; libraries conventionally don't commit it, since consumers resolve their own compatible versions.

Workspaces: Multiple Crates, One Project

```
# root Cargo.toml
[workspace]
members = ["app", "core-lib"]
```

A workspace shares one `Cargo.lock` and one `target/` build directory across every member crate — the standard pattern once a project grows past a single crate, e.g. splitting a binary crate from the library crate(s) it depends on.

Publishing to crates.io

`cargo publish` requires a globally unique crate name (crates.io names can't be reused once claimed), required metadata (license, description), and an API token. Publishing is **permanent** for a given version

number — a published version can be *yanked* (hidden from new dependents) but never overwritten or deleted outright.

	NPM (JAVASCRIPT)	CARGO (RUST)
Lock file	`package-lock.json`	`Cargo.lock`
Purpose	`Pin exact resolved versions for reproducibility`	`Identical purpose`
Committed for libraries?	`Typically not`	`Typically not (unlike applications, which do)`

mod / pub

Organizes code into namespaces; private by default, pub exposes to the parent module.

File-Based Modules

mod name; points at name.rs, with name/submodule.rs for its own children.

Cargo.toml / Cargo.lock

Package metadata and dependencies; the lock file pins exact resolved versions.

Workspaces

Multiple crates sharing one lock file and build directory under one root Cargo.toml.

PRIVATE-BY-DEFAULT IS THE SAME PHILOSOPHY AGAIN

Rust's modules being private unless explicitly marked `pub` is the same "safety by default" instinct behind Course 1's immutable-by-default variables — an accidentally-exposed item requires deliberately opting in with `pub`, rather than requiring someone to remember to lock it down after the fact.

PUB ALONE DOESN'T GUARANTEE EXTERNAL VISIBILITY

Marking a deeply-nested item `pub` only exposes it to its immediate parent module — it does **not** automatically make it reachable from outside the crate. Every module *along the path* to that item must also be `pub` (or the item must be re-exported via `pub use` at a more accessible path). Visibility has to be "pub all the way up," not just at the final leaf item — a genuinely common early confusion.

Coding Challenges

CHALLENGE 1

Write a module tree: a top-level module shop containing a submodule inventory with a public function check_stock(). Show the full path needed to call check_stock() from outside the shop module, and explain what pub keywords are required at each level.

View solution

CHALLENGE 2

Write a Cargo.toml for a binary crate named "task_tracker" at version 0.1.0, using the 2021 edition, depending on serde version "1.0" and clap version "4.0". Explain what Cargo.lock would add on top of this file once the project is built.

View solution

CHALLENGE 3

Explain the specific bug in this module tree: mod a { pub mod b { pub fn f() {} } } where a itself is NOT marked pub — why can code outside the crate still not call a::b::f() from outside, even though both b and f are marked pub?

View solution

CHAPTER 6 QUICK REFERENCE

- **mod name { ... }** — declares a module; items are private by default
- **pub** — exposes an item to its parent module; **pub(crate)** — exposes crate-wide only
- **mod name;** (no body) — loads the module from name.rs, with name/ for its own submodules
- **use path::to::item;** — shortens repeated paths; **pub use** — re-exports at a shallower path
- **Cargo.toml** — package metadata + dependencies; **Cargo.lock** — exact resolved versions, like npm's package-lock.json
- **[workspace] members = [...]** — multiple crates sharing one lock file and build directory
- **cargo publish** — permanent per version; a bad publish can only be yanked, never overwritten
- pub must apply all the way up the module path, not just at the final item, for external code to actually reach it
- **Next chapter:** Testing in Rust — #[test], cargo test, unit vs. integration tests, contrasted with Go's built-in testing package

CHAPTER 7 OF 7

Testing in Rust

COURSE 2 · CH 7

Testing in Rust

Verifying everything Course 2 built actually works — the fitting close to this course

Course 2 has built a real project's worth of Rust knowledge — lifetimes, traits, generics, smart pointers, concurrency, modules. This final chapter closes with the tool that lets you trust all of it actually behaves as intended: Rust's built-in testing framework.

The #[test] Attribute

```
#[test]
fn it_adds_two_numbers() {
    assert_eq!(add(2, 3), 5);
}
```

A function annotated `#[test]` becomes a test case, run with `cargo test`. A panic inside a test function **is** a failed test — no separate failure mechanism exists. This is a genuinely elegant reuse of Course 1 Chapter 7's `panic!` material: testing failure is just triggering a panic, nothing new to learn.

Organizing Unit Tests

The conventional Rust pattern: unit tests live in the **same file** as the code they test, inside a `#[cfg(test)]` submodule — a real departure from many languages' separate-test-directory convention:

```
pub fn add(a: i32, b: i32) -> i32 { a + b }

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_adds_two_numbers() {
        assert_eq!(add(2, 3), 5);
    }
}
```

`#[cfg(test)]` means this module compiles *only* when running tests, never in a normal build. `use super::*` brings the parent module's items — including private ones — into the test module's scope, so unit tests can exercise private implementation details directly, not just the public API.

Integration Tests

A separate top-level `tests/` directory, sibling to `src/`, holds integration tests. Each file inside it compiles as its **own separate crate**, testing only the library's **public** API — exactly as an external consumer would use it:

```
// tests/integration_test.rs
use my_crate::add;

#[test]
fn public_add_works() {
    assert_eq!(add(2, 2), 4);
}
```

	UNIT TESTS	INTEGRATION TESTS
Location	`Same file, #[cfg(test)] mod`	`tests/ directory`
Compiled as	`Part of the same crate`	`A separate crate per file`
Can access	`Private and public code`	`Public API only`

Useful cargo test Options

- `cargo test add` — runs only tests whose name contains "add"
- `cargo test -- --nocapture` — shows `println!` output even for passing tests (captured/hidden by default)
- `#[should_panic]` — a test that **passes** only if the function panics, useful for testing error conditions
- `#[ignore]` — skips a slow test by default; run explicitly with `cargo test -- --ignored`

```
#[test]
#[should_panic]
fn rejects_a_negative_age() {
    Age::new(-1);  // this test PASSES because new() is expected to panic here
}
```

Contrast With Go's Testing Package (go3-4)

Go's built-in testing uses a naming convention instead of an attribute — a function named `TestXxx(t *testing.T)` in a `_test.go` file is automatically discovered, no annotation required. Go failures are reported explicitly via `t.Error()` / `t.Fatal()` calls rather than Rust's "a panic is a failure" approach. Both languages ship testing built-in with no external framework needed — the actual mechanics of what counts as a test, and what counts as a failure, genuinely differ.

	GO	RUST
Test discovery	Naming convention (TestXxx in _test.go)	#[test] attribute
Reporting failure	Explicit t.Error()/t.Fatal() calls	A panic — no separate reporting call
Unit vs. integration separation	File naming/build tags convention	Same-file #[cfg(test)] vs. a separate tests/ directory

#[test]

Marks a function as a test case; a panic inside it means the test failed.

#[cfg(test)] mod tests

Same-file unit tests, compiled only when testing, with access to private code.

tests/ Directory

Integration tests, each file its own crate, exercising only the public API.

#[should_panic]

A test that passes specifically because the tested code panics — for verifying error conditions.

A FITTING CLOSE TO COURSE 2

Testing is the right note to end this course on — it's the tool that lets you actually trust that lifetimes, traits, generics, smart pointers, and concurrent code all behave as intended, especially once Chapter 6's workspaces come into play with multiple crates each needing their own test coverage.

FORGETTING #[CFG(TEST)] BLOATS EVERY REGULAR BUILD

Without `#[cfg(test)]` on the test module, its code — and anything it pulls in via `use super::*` — gets compiled into **every** normal build, not just test runs. This inflates binary size and can unintentionally require test-only dependencies in production builds. Not catastrophic, but a real, common oversight worth double-checking.

Coding Challenges

CHALLENGE 1

Write a function subtract(a: i32, b: i32) -> i32 alongside a #[cfg(test)] mod tests block containing two #[test] functions: one verifying a normal case, one verifying subtracting a number from itself yields zero.

View solution

CHALLENGE 2

Write a function divide(a: f64, b: f64) -> f64 that panics if b is 0.0, and a #[test] #[should_panic] test verifying that dividing by zero actually panics.

View solution

CHALLENGE 3

Explain why a unit test inside a #[cfg(test)] mod tests block can call a private function directly, while a test inside the tests/ directory cannot — referencing what "use super::*" actually brings into scope and what a tests/ file compiles as instead.

View solution

CHAPTER 7 QUICK REFERENCE

- **#[test]** — marks a function as a test case, run by cargo test; a panic = a failure
- **assert!/assert_eq!/assert_ne!** — the standard assertion macros used inside tests
- **#[cfg(test)] mod tests { use super::*; }** — same-file unit tests, compiled only when testing, private+public access
- **tests/ directory** — integration tests, each file its own crate, public API only
- **#[should_panic]** — a test that passes only if the code panics; **#[ignore]** — skip unless run explicitly
- Go discovers tests by naming convention (TestXxx) and reports failure via explicit t.Error(); Rust uses an attribute and treats any panic as failure
- Forgetting #[cfg(test)] compiles test code into every regular build, not just test runs

★ Rust Intermediate Complete — 7 / 7 chapters

From lifetimes through traits, generics, smart pointers, concurrency, modules, and finally testing — Course 2 has built the tools needed for real, growing Rust projects. Next up: **Rust Advanced**, covering advanced traits, unsafe Rust, async Rust, macros, and performance — closing with a capstone CLI tool.