RUST / OWNERSHIP / TYPES / CONCURRENCY

Rust by Evidence

Learn ownership, types, and concurrency
by reading what the compiler proves

WINTERLOO

EDITION 1.0 · 2026

How to read this book

Rust is easiest to learn when you stop treating compiler errors as interruptions. They are observations about a program: a value moved here, a reference must remain valid there, or a generic function asked for an operation it never required.

This book teaches those relationships through short predictions. Each chapter follows the same loop:

1. Read a small program.
2. Predict whether it compiles or what it prints.
3. Inspect the actual result.
4. Build a working mental model.
5. Refine that model into the precise rule.
6. Compare two valid fixes.
7. Solve one variation.

Do not merely read the snippets. Put them in `src/main.rs` and run `cargo check` or `cargo run`. For deliberate failures, read the first relevant diagnostic before looking at the answer.

The six questions

Most of the book can be navigated with six questions:

- Who owns this value?
- Is this operation a move, copy, borrow, or reborrow?
- How long must this reference remain valid?
- Is mutation happening while another path can observe the value?
- Does the compiler know the concrete type and required capabilities?
- Is this state absence (`Option`) or failure (`Result`)?

These are more useful than memorizing individual error codes. They transfer from tiny exercises to real programs.

Setup

Install stable Rust with `rustup`, then verify:

```
$ rustc --version
rustc 1.92.0 (ded5c06cf 2025-12-08)
$ cargo --version
cargo 1.92.0 (344c4567c 2025-10-21)
```

Create a scratch program:

```
$ cargo new rust-scratch
$ cd rust-scratch
```

```
$ cargo run
```

Use these commands throughout:

- `cargo check`: type-check quickly without producing the final executable.
- `cargo run`: build and execute a binary.
- `cargo test`: run checks marked with `#[test]` and documentation tests.
- `cargo fmt`: apply standard formatting.
- `cargo clippy --all-targets -- -D warnings`: reject suspicious code and warnings.

The companion crate at the root of this repository contains corrected examples and one integrated test. It deliberately has no dependencies. The standard library is enough to learn the language rules.

What this book does not do

It does not survey every standard-library API, teach a web framework, or prescribe an application architecture. Those subjects change and are better learned when a project requires them. The goal here is durable: make Rust's ownership, type, and concurrency rules predictable.

Start with Part I even if you already know another systems language. Rust references are not merely pointers with friendlier syntax; their types encode aliasing and lifetime constraints that shape APIs throughout the rest of the language.

CONTENTS

Fifteen steps, one mental model.

Plus four practice projects and a compiler field guide.

Chapter 1 — Values, Bindings, and Mutability

GOAL

Understand the difference between a value, the name bound to it, and permission to replace that value.

? PREDICTION

```
fn main() {
    let score = 10;
    score = 11;
    println!("{score}");
}
```

What happens?

A. It prints `11` because variables are mutable by default.
B. It prints `10` because assignment is ignored.
C. Compilation fails because `score` was not declared mutable.
D. Compilation fails because integers cannot change.

REVEAL: READ THE EVIDENCE

Answer: **C**.

The useful part of `rustc`'s report is:

```
error[E0384]: cannot assign twice to immutable variable `score`
  |
2 |     let score = 10;
  |         ----- first assignment to `score`
3 |     score = 11;
  |     ^^^^^^^^^^ cannot assign twice to immutable variable
help: consider making this binding mutable
  |
2 |     let mut score = 10;
  |         +++
```

The integer is not inherently frozen. The binding named `score` lacks permission to be assigned another value.

BUILD THE MODEL

Working model: `let` creates a name for a value. The name is immutable unless you write `mut`.

```
let mut score = 10;
score = 11;
```

This says more than "Rust likes extra keywords." A reader can see which bindings are expected to change. Accidental reassignment elsewhere becomes a compiler error.

A binding is not the value itself. That distinction appears clearly with shadowing:

```
fn main() {
    let input = " 42 ";
    let input = input.trim();
    let input: u32 = input.parse().expect("a whole number");
    println!("{input}");
}
```

Each `let input = ...` creates a new binding. The old binding is hidden, or *shadowed*. Because this is a new binding, its type may change from `&str` to `u32`. Reassignment cannot do that:

```
let mut input = "42";
input = 42; // expected `&str`, found integer
```

Precise model: a binding associates a pattern with a value for some scope. Without `mut`, you cannot assign through that binding. Shadowing introduces a distinct binding, even when it reuses the same spelling. Mutability is also not deep or universal: later you will see that a mutable binding does not override borrowing rules.

Values whose size is known at compile time often live directly in the current stack frame, but do not use "binding equals stack slot" as a language rule. The compiler may optimize storage away. Ownership and permitted operations are the reliable model; physical addresses usually are not.

SIMPLEST FIX

Declare expected reassignment:

```
fn main() {
    let mut score = 10;
    score += 1;
    println!("{score}");
}
```

This prints `11`. Use `mut` when one logical thing changes over time: a counter, accumulator, or buffer.

TRADEOFF ALTERNATIVE: SHADOWING

```
fn main() {
    let score = 10;
    let score = score + 1;
    println!("{score}");
}
```

Shadowing keeps each binding immutable and permits a type change. It is useful for staged transformation. Its cost is that too many same-named stages can make debugging and error messages harder to follow. Do not shadow merely to avoid an honest `mut`.

MINI CHALLENGE

What prints?

```
fn main() {
    let spaces = "   ";
    let spaces = spaces.len();
    println!("{spaces}");
}
```

A. Three spaces
B. `3`
C. It fails because `spaces` changes type.
D. It fails because `spaces` is immutable.

ANSWER

B. The second `let` creates a new `usize` binding. No immutable binding is reassigned. The program prints `3`.

Chapter 2 — Ownership and Moves

GOAL

Predict when using a value transfers ownership and makes its previous binding unusable.

? PREDICTION

```
fn main() {
    let label = String::from("draft");
    let saved = label;
    println!("{label} -> {saved}");
}
```

What happens?

A. Both names print because assignment aliases the same string.
B. Both names print because every value is copied.
C. Compilation fails because ownership moved from `label` to `saved`.
D. Compilation fails because `String` cannot be printed.

REVEAL: READ THE EVIDENCE

Answer: **C**.

```
error[E0382]: borrow of moved value: `label`
  |
2 |     let label = String::from("draft");
  |         ----- move occurs because `label` has type `String`, which does not implement the `Copy`
trait
3 |     let saved = label;
  |                 ----- value moved here
4 |     println!("{label} -> {saved}");
  |                ^^^^^ value borrowed here after move
```

The last line says “borrowed” because formatting temporarily reads `label`. That read is rejected: `label` no longer owns a valid `String`.

BUILD THE MODEL

A `String` is a small control value containing information such as a pointer, length, and capacity. Its character bytes are held in heap storage. When the owning `String` goes out of scope, Rust runs its destructor and releases that storage.

If assignment merely copied the control value, two `String`s would later try to release the same allocation. Rust instead moves ownership:

```
let label = String::from("draft");
let saved = label;
// `saved` is now responsible for cleanup.
```

Working model: a value has one owner. Assigning or passing an owned value transfers it unless the type is cheaply copyable. After a move, stop using the old binding.

Function calls follow the same rule:

```
fn archive(text: String) {
    println!("archived: {text}");
}

fn main() {
    let note = String::from("ship it");
    archive(note);
    // `note` was moved into `archive`.
}
```

Ownership is returned when a function returns the value:

```
fn inspect(text: String) -> String {
    println!("{} bytes", text.len());
    text
}
```

That works, but passing ownership out and back is noisy when the function only needs to inspect data. Borrowing, introduced next, expresses that intent directly.

Precise model: move behavior applies to types that do not implement `Copy`. Types such as `u32`, `bool`, `char`, and many tuples of `Copy` values are copied on assignment:

```
let first = 7;
let second = first;
println!("{first} {second}");
```

Both integers remain usable. `Copy` is a trait with restrictions; it is not chosen dynamically by value size. `String` owns a resource and implements `Drop`, so it is not `Copy`.

A move is a language-level transfer, not necessarily a physical byte-by-byte operation you should try to visualize. Optimizations may remove the copy entirely. The dependable fact is which binding may still be used and which value will be cleaned up.

SIMPLEST FIX

Use the new owner:

```
fn main() {
    let label = String::from("draft");
    let saved = label;
    println!("saved as {saved}");
}
```

Often the compiler found a real design truth: the old name was unnecessary.

TRADEOFF ALTERNATIVE: CLONE

```
fn main() {
    let label = String::from("draft");
    let saved = label.clone();
    println!("{label} -> {saved}");
}
```

`clone` creates an independent owned `String`, including another allocation and copied bytes. Use it when both owners genuinely need independent lifetimes or mutation. Do not add `clone()` automatically to silence E0382; a borrow is usually cheaper when code only reads the value.

✦ MINI CHALLENGE

Which line fails?

```
fn consume(text: String) {
    println!("{text}");
}

fn main() {
    let message = String::from("hello");
    let count = message.len();
    consume(message);
    println!("{count}");
}
```

A. `message.len()`
B. `consume(message)`
C. `println!("{count}")`
D. No line fails.

✓ ANSWER

D. `len()` only reads `message`. `consume` then moves it, but `message` is never used afterward. `count` is a `usize`, which is a separate `Copy` value.

Chapter 3 — Shared and Mutable Borrowing

GOAL

Use `&T` and `&mut T` to access a value without taking ownership, and predict when those accesses conflict.

? PREDICTION

```
fn main() {
    let mut task = String::from("write");
    let view = &task;
    task.push_str(" tests");
    println!("{view}");
}
```

What happens?

A. It prints `write tests`; references always observe changes.
B. It prints `write`; references contain snapshots.
C. Compilation fails because mutation conflicts with the later use of `view`.
D. Compilation fails because `push_str` consumes the string.

REVEAL: READ THE EVIDENCE

Answer: **C**.

```
error[E0502]: cannot borrow `task` as mutable because it is also borrowed as immutable
  |
3 |     let view = &task;
  |                ----- immutable borrow occurs here
4 |     task.push_str(" tests");
  |     ^^^^^^^^^^^^^^^^^^^^^^ mutable borrow occurs here
5 |     println!("{view}");
  |                ---- immutable borrow later used here
```

`push_str` needs mutable access to `task`. The compiler sees shared access still needed afterward, so the accesses overlap.

BUILD THE MODEL

Borrowing grants temporary access while leaving ownership where it is.

```
fn byte_count(text: &String) -> usize {
    text.len()
}

fn main() {
    let task = String::from("write");
    let count = byte_count(&task);
    println!("{task}: {count}");
}
```

`&task` creates a shared reference. `byte_count` may read the `String`, but cannot replace or mutate it. The owner remains `task`, so it is usable after the call.

Mutable borrowing grants temporary exclusive access:

```
fn finish(text: &mut String) {
    text.push_str(": done");
}

fn main() {
    let mut task = String::from("write");
    finish(&mut task);
    println!("{task}");
}
```

Working model: at a given moment you may have either many shared references or one mutable reference, not both. Shared means "read without exclusive control." Mutable means "exclusive access, including permission to write."

This rule prevents two broad classes of bugs: observing a collection while an operation relocates its storage, and unsynchronized read/write or write/write access to the same value. For example, appending to a `String` may allocate a larger buffer. A reference into the old buffer must not remain usable across that append.

The owner is also restricted while its value is borrowed. `task.push_str(...)` behaves as a mutable borrow of `task`, so it cannot overlap the active shared borrow in the prediction.

Precise model: Rust checks places and uses, not merely variable names. References may be *reborrowed*, and the compiler can shorten a borrow to its last use. Types with interior mutability can permit mutation through shared references by moving checks to runtime or using synchronization, but ordinary `&T` does not permit mutation. Start with the ordinary rule; reach for interior-mutability types only when the design truly requires them.

SIMPLEST FIX

Finish reading before mutation:

```
fn main() {
    let mut task = String::from("write");
    let view = &task;
    println!("before: {view}");
    task.push_str(" tests");
    println!("after: {task}");
}
```

Because `view` is not used after the first `println!`, its borrow ends there. The mutation is then exclusive.

TRADEOFF ALTERNATIVE: CLONE THE SNAPSHOT

```
fn main() {
    let mut task = String::from("write");
    let before = task.clone();
    task.push_str(" tests");
    println!("{before} -> {task}");
}
```

Cloning is appropriate when you need an owned historical snapshot. It costs allocation and copying; shortening the borrow costs nothing and should be the default when the old view is not independently needed.

✦ MINI CHALLENGE

Will this compile?

```
fn main() {
    let mut total = 0;
    let left = &mut total;
    let right = &mut total;
    *left += 1;
    *right += 1;
}
```

A. Yes; both references came from a mutable binding.
B. Yes; integers are `Copy`.
C. No; two mutable borrows are active at once.
D. No; references cannot modify integers.

✓ ANSWER

C. `left` is used after `right` is created, so their exclusive borrows overlap. A simple sequential version works:

```
let mut total = 0;
*(&mut total) += 1;
*(&mut total) += 1;
```

Usually write the clearer `total += 1` twice; explicit references add nothing here.

Chapter 4 — References, Scopes, and Non-Lexical Lifetimes

GOAL

Predict how long a borrow lasts and distinguish a reference's scope from the lifetime required for it to remain valid.

? PREDICTION

```
fn main() {
    let mut name = String::from("Ada");
    let first = &name;
    println!("{first}");
    name.push_str(" Lovelace");
    println!("{name}");
}
```

What happens on modern Rust?

A. It fails because `first` remains in scope until the closing brace.
B. It compiles because the borrow through `first` ends after its last use.
C. It fails because referenced strings can never be mutated.
D. It compiles only in release mode.

REVEAL: READ THE EVIDENCE

Answer: **B**. On Rust 1.92, `rustc` accepts it, and it prints:

```
Ada
Ada Lovelace
```

The binding `first` remains in lexical scope, but the borrow does not need to remain active after `println!("{first}")`. This is a non-lexical lifetime: borrow checking follows relevant uses rather than blindly extending every borrow to the end of its enclosing block.

BUILD THE MODEL

A reference is only valid while the value it points to is alive. Rust checks that relationship at compile time:

```
fn main() {
    let outside: &String;
    {
        let inside = String::from("temporary");
        outside = &inside;
    }
    println!("{outside}");
}
```

The representative evidence is:

```
error[E0597]: `inside` does not live long enough
  |
4 |         let inside = String::from("temporary");
5 |         outside = &inside;
  |                   ^^^^^^^ borrowed value does not live long enough
6 |     }
  |     - `inside` dropped here while still borrowed
7 |     println!("{outside}");
  |                -------- borrow later used here
```

The problem is not that references dislike nested blocks. The owner `inside` is destroyed at the inner closing brace, but `outside` would be used later.

Working model: a borrow lasts from its creation through its last relevant use. The referenced value must stay alive for that entire period.

Two different ideas are easy to mix up:

- A **scope** is a region of source code where a binding can be named.
- A **lifetime** is the region during which a reference must be valid.

In the prediction, `first` is nameable until the end of `main`, but no later code uses it. Its required lifetime can therefore end earlier. In the failing nested example, `outside` is used after `inside` is dropped, so no shortening can make the reference valid.

A reference never extends an owner's life. Rust does not quietly keep `inside` allocated because `outside` points to it. Instead, it rejects the program.

Precise model: the borrow checker reasons over control flow and inferred regions. “Ends at last use” is a strong working shortcut, but branches and returned references can require a borrow along multiple paths. Lifetime annotations, when needed in later APIs, describe relationships among reference lifetimes; they do not prolong values or perform runtime cleanup.

SIMPLEST FIX

Move the owner into a scope at least as wide as every use of the reference:

```
fn main() {
    let inside = String::from("lasting");
    let outside = &inside;
    println!("{outside}");
}
```

Ownership now outlives the borrow.

TRADEOFF ALTERNATIVE: RETURN OWNERSHIP

If a helper creates the data, return the value rather than a reference to its local variable:

```
fn make_label() -> String {
    String::from("temporary")
}

fn main() {
    let label = make_label();
    println!("{label}");
}
```

Returning `String` transfers ownership safely and is usually cheap because moving a `String` does not copy all its characters. The tradeoff is that the caller now owns cleanup. Trying to return `&String` here would require pointing at a local value that is destroyed when the function returns.

✦ MINI CHALLENGE

Will this compile?

```
fn main() {
    let mut items = vec![1, 2, 3];
    let first = &items[0];
    println!("{first}");
    items.push(4);
}
```

A. Yes, because `first` is not used after `push`.
B. No, because vectors cannot be borrowed.
C. No, because any reference lasts to the end of the block.
D. It depends on vector capacity at runtime.

✓ ANSWER

A. The shared borrow's last use precedes `push`, so non-lexical lifetime analysis ends it in time. Runtime capacity does not affect whether the program type-checks. If you printed `first` after `push`, compilation would fail because `push` may relocate the vector's elements.

Chapter 5 — `String` and `&str`

GOAL

Choose between owned, growable string data and a borrowed view of UTF-8 text.

? PREDICTION

```
fn announce(text: &str) {
    println!("{text}");
}

fn main() {
    let message = String::from("ready");
    announce(message);
}
```

What happens?

A. It prints because `String` and `&str` are identical types.
B. Compilation fails; the function expects `&str` but receives `String`.
C. Compilation fails because string literals are required.
D. It prints and consumes `message`.

REVEAL: READ THE EVIDENCE

Answer: **B**.

```
error[E0308]: mismatched types
  |
7 |     announce(message);
  |     -------- ^^^^^^^ expected `&str`, found `String`
  |     |
  |     arguments to this function are incorrect
help: consider borrowing here
  |
7 |     announce(&message);
  |              +
```

The compiler does not invent a borrow at this call site. Add `&` to pass temporary shared access.

BUILD THE MODEL

`String` is an owned, growable UTF-8 string. It manages heap storage and can change when its binding and borrow state permit mutation:

```
let mut owned = String::from("red");
owned.push_str(" fox");
```

`&str`, pronounced "string slice," is a borrowed view into valid UTF-8 bytes owned elsewhere. A string literal has type `&'static str` because its bytes are embedded in the program:

```
let literal: &str = "red fox";
```

A slice may also view part or all of a `String`:

```
let owned = String::from("red fox");
let all: &str = &owned;
let first: &str = &owned[..3];
println!("{first}, {all}");
```

Here deref coercion lets `&String` become `&str` where needed. The slice does not own the bytes, so it cannot outlive `owned`.

Working model: use `String` when code must own or build text. Use `&str` when code only needs to read text owned somewhere else.

That makes `&str` a strong default for read-only function parameters:

```
fn initials(name: &str) -> Option<char> {
    name.chars().next()
}
```

Both callers work without allocation:

```
fn initials(name: &str) -> Option<char> {
    name.chars().next()
}

let owned = String::from("Grace");
assert_eq!(initials(&owned), Some('G'));
assert_eq!(initials("Linus"), Some('L'));
```

A parameter of `&String` is less flexible: it asks specifically for a borrowed `String`, even though the function may only need text. Prefer `&str` unless `String`-specific capacity or mutation behavior is part of the contract.

String slicing uses byte indices, not character positions. This is valid because the boundaries surround complete UTF-8 characters:

```
let word = "café";
let cafe = &word[..5];
assert_eq!(cafe, "café");
```

But an index that cuts through the two-byte é will panic at runtime. For text processing by Unicode scalar values, use `.chars()`; for raw bytes, use `.bytes()`. Neither is the same as user-perceived grapheme clusters, which the standard library does not segment for you.

Precise model: `str` is a dynamically sized sequence of valid UTF-8 bytes. It is usually handled behind a pointer such as `&str`, which carries a data pointer and a length. `String` owns a UTF-8 buffer and can yield a `&str` view. `&str` is not limited to literals and does not necessarily cover an entire allocation.

SIMPLEST FIX

Borrow the owned string:

```
fn announce(text: &str) {
    println!("{text}");
}

fn main() {
    let message = String::from("ready");
    announce(&message);
    println!("still owned here: {message}");
}
```

No character data is copied.

TRADEOFF ALTERNATIVE: ACCEPT OWNERSHIP

```
fn queue(text: String) {
    // The queue could store `text` after this call.
    println!("queued: {text}");
}

fn main() {
    let message = String::from("ready");
    queue(message);
}
```

Accept `String` when the function needs to retain, mutate independently, or transfer the text. The tradeoff is that callers with `&str` must create an owned value, usually with `.to_owned()` or `String::from`, which allocates. Ownership should reflect a real need, not be the default for convenience.

✦ MINI CHALLENGE

Which signature accepts both a string literal and a borrowed `String` without allocating or taking ownership?

A. `fn show(text: String)`
B. `fn show(text: &String)`
C. `fn show(text: &str)`
D. `fn show(text: str)`

✓ ANSWER

C. Call it as `show("literal")` or `show(&owned)`. `String` would take ownership and require allocation for the literal. `&String` does not directly accept a literal. Bare `str` is dynamically sized and cannot be passed by value as an ordinary parameter.

You now have the core questions for reading early Rust code: Which binding may change? Who owns each value? Did this operation move, copy, or borrow it? If borrowed, is access shared or exclusive, and how long is the reference actually used? For text, does this code need ownership, or only a `&str` view? Keep asking those questions; many intimidating compiler errors become precise answers.

Chapter 6 — Structs and Methods

GOAL

Model one coherent thing, then attach operations that belong to that thing.

A struct names a collection of fields. Unlike a tuple, it records what each position means:

```
struct Rectangle {
    width: u32,
    height: u32,
}

fn area(rect: Rectangle) -> u32 {
    rect.width * rect.height
}

fn main() {
    let poster = Rectangle { width: 30, height: 40 };
    println!("{}", area(poster));
    println!("{}", poster.width);
}
```

? PREDICTION

What happens?

- A. It prints `1200` and then `30`.
- B. It fails because struct fields are private.
- C. It fails because `area` takes ownership of `poster`.
- D. It prints `1200` twice.

→ REVEALC

Representative compiler evidence:

```
error[E0382]: use of moved value: `poster`
  |
  |     println!("{}", area(poster));
  |                         ------ value moved here
  |     println!("{}", poster.width);
  |                     ^^^^^^^^^^^^ value used here after move
```

The fields are accessible because this code is in the same module. The problem is the function parameter: `rect: Rectangle` consumes its argument.

MENTAL MODEL

Working model: a struct is one value with named parts. Moving the struct moves the whole value unless the type implements `Copy`.

Precise model: methods are functions declared in an `impl` block. Their first parameter controls access. `self` consumes the value, `&self` borrows it for reading, and `&mut self` borrows it for mutation. Method-call syntax supplies that first argument and performs ordinary borrowing adjustments where possible.

The operation here only observes a rectangle, so make that visible in its signature:

```
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }

    fn scale(&mut self, factor: u32) {
        self.width *= factor;
        self.height *= factor;
    }

    fn square(size: u32) -> Self {
        Self { width: size, height: size }
    }
}

fn main() {
    let mut poster = Rectangle::square(30);
    poster.scale(2);
    println!("{} by {}; area {}", poster.width, poster.height, poster.area());
}
```

Output:

```
60 by 60; area 3600
```

`area` and `scale` are methods because they receive `self`. `square` is an associated function: call it with `Rectangle::square`, not with a value.

SIMPLEST FIX

Change a read-only parameter from `Rectangle` to `&Rectangle`, or make it an `&self` method. That avoids a move and states the actual need.

Tradeoff alternative: derive `Copy` and `Clone` when every field is cheaply copyable:

```
#[derive(Clone, Copy)]
struct Rectangle {
    width: u32,
    height: u32,
}
```

Then passing a rectangle copies it. This is convenient for small plain-data types, but it changes assignment semantics for the entire type. Do not add `Copy` merely to silence one move error; borrowing scales to structs that later contain a `String` or other owned resource.

MINI CHALLENGE

Make this compile without cloning and without changing `label`:

```
struct Package {
    label: String,
    delivered: bool,
}

fn mark_delivered(package: Package) {
    package.delivered = true;
}

fn main() {
    let mut parcel = Package {
        label: String::from("BX-17"),
        delivered: false,
    };
    mark_delivered(parcel);
    println!("{}: {}", parcel.label, parcel.delivered);
}
```

ANSWER

Borrow the package mutably, and make the parameter binding operate through that borrow:

```
struct Package {
    label: String,
    delivered: bool,
}

fn mark_delivered(package: &mut Package) {
    package.delivered = true;
}

fn main() {
    let mut parcel = Package { label: String::from("BX-17"), delivered: false };
    mark_delivered(&mut parcel);
    println!("{}: {}", parcel.label, parcel.delivered);
}
```

The exclusive borrow lasts for the call; afterward `parcel` is usable again.

Chapter 7 — Enums and Pattern Matching

GOAL

Represent a value that is exactly one of several meaningful alternatives.

A boolean can say yes or no, but it cannot explain *which kind* of yes. An enum gives each case a name and may attach different data to each case.

```
enum Command {
    Quit,
    Move { x: i32, y: i32 },
    Say(String),
}

fn describe(command: Command) -> &'static str {
    match command {
        Command::Quit => "quit",
        Command::Move { x, y } => "move",
    }
}
```

? PREDICTION

Will `describe` compile?

- A. Yes; `match` may ignore uncommon variants.
- B. No; `Say` is not covered.
- C. No; enum variants cannot hold fields.
- D. Yes; unmatched variants return an empty string.

→ REVEALB

Representative rustc evidence:

```
error[E0004]: non-exhaustive patterns: `Command::Say(_)` not covered
  |
  |     match command {
  |           ^^^^^^^ pattern `Command::Say(_)` not covered
```

Rust requires an exhaustive `match`. If a `Command` exists, one arm must handle its variant.

MENTAL MODEL

Working model: an enum value carries a tag saying which variant it is, plus the data for that variant. A `match` reads the tag, selects one arm, and can unpack the attached data.

Precise model: every match arm is a pattern followed by an expression. Patterns can test structure, bind names, ignore parts with `_`, and include alternatives with `|`. All arms must produce compatible types because the whole `match` is an expression. Exhaustiveness is checked at compile time.

```
enum Command { Quit, Move { x: i32, y: i32 }, Say(String) }

fn describe(command: &Command) -> String {
    match command {
        Command::Quit => String::from("quit"),
        Command::Move { x, y } => format!("move to ({x}, {y})"),
        Command::Say(text) => format!("say {text:?}"),
    }
}

fn main() {
    let command = Command::Say(String::from("steady"));
    println!("{}", describe(&command));
}
```

Output:

```
say "steady"
```

Matching `&Command` means bindings such as `text` are references to data inside the borrowed enum. The command remains available after the call.

Use `if let` when only one pattern matters:

```
if let Command::Move { x, y } = command {
    println!("destination: {x}, {y}");
}
```

This deliberately ignores every other variant. That is concise, but it gives up the compiler's reminder when a new variant deserves handling.

SIMPLEST FIX

Add the missing explicit arm:

```
Command::Say(text) => "say",
```

Use the bound `text` if the result needs its contents; otherwise write `Command::Say(_)`.

Tradeoff alternative: add a wildcard arm, `_ => "other"`. It is appropriate when all remaining and future variants truly share behavior. The cost is silence: adding a new variant will not force this match to be revisited.

MINI CHALLENGE

Return the payload for `Reading` and zero for the other states:

```
enum Meter {
    Offline,
    Reading(i64),
    Fault { code: u16 },
}

fn value(meter: Meter) -> i64 {
    todo!()
}
```

Which implementation is both exhaustive and able to compile?

- A. `match meter { Meter::Reading(n) => n }`
- B. `match meter { Meter::Reading(n) => n, _ => 0 }`
- C. `if meter == Meter::Offline { 0 } else { meter.0 }`
- D. `match meter { _ => "0" }`

✓ **ANSWER**

B:

```
enum Meter { Offline, Reading(i64), Fault { code: u16 } }

fn value(meter: Meter) -> i64 {
    match meter {
        Meter::Reading(n) => n,
        _ => 0,
    }
}
```

The wildcard is reasonable here because the requirement explicitly groups every non-reading state together. If faults later need logging, replace `_` with named arms so the distinction becomes visible.

Chapter 8 — `Option` and `Result`

GOAL

Separate ordinary absence from operations that can fail, without sentinel values or unchecked exceptions.

`Option<T>` means a `T` may be absent. `Result<T, E>` means an operation either produced `T` or explains failure with `E`. Both are ordinary enums in the standard library.

```
fn first_char(text: &str) -> Option<char> {
    text.chars().next()
}

fn main() {
    let initial: char = first_char("");
    println!("{initial}");
}
```

? PREDICTION

What does the compiler do?

- A. Stores the null character in `initial`.
- B. Panics because the string is empty.
- C. Rejects `Option<char>` where `char` is required.
- D. Converts `None` to a space.

→ REVEALC

Representative evidence:

```
error[E0308]: mismatched types
  |
  |     let initial: char = first_char("");
  |                 ----   ^^^^^^^^^^^^^^ expected `char`, found `Option<char>`
```

The type forces the caller to choose what absence means.

MENTAL MODEL

Working model: `Some(value)` is present and `None` is absent. `Ok(value)` is success and `Err(error)` is failure. You cannot use the inner value until you handle the outer enum.

Precise model: `Option<T>` and `Result<T, E>` compose because their methods transform only the success/present branch. `map` changes an inner value; `and_then` chains an operation that already returns

the same kind of wrapper. The `?` operator returns early on `None` or `Err` and unwraps the continuing branch. Its enclosing function must return a compatible type.

```
fn first_upper(text: &str) -> Option<char> {
    let first = text.chars().next()?;
    Some(first.to_ascii_uppercase())
}

fn parse_port(text: &str) -> Result<u16, std::num::ParseIntError> {
    let port = text.parse::<u16>()?;
    Ok(port)
}

fn main() {
    println!("{:?}", first_upper("rust"));
    println!("{:?}", first_upper(""));
    println!("{:?}", parse_port("8080"));
    println!("{:?}", parse_port("many"));
}
```

Representative output:

```
Some('R')
None
Ok(8080)
Err(ParseIntError { kind: InvalidDigit })
```

The exact debug text of library errors is not a stable interface; match on error values or display them rather than depending on that formatting.

Choose between the wrappers by asking whether the caller needs a reason. Searching a list may naturally return `None`. Parsing user input should return `Err` because malformed input differs from successful absence.

SIMPLEST FIX

For a genuine default, unwrap explicitly with `unwrap_or`:

```
fn first_char(text: &str) -> Option<char> { text.chars().next() }

let initial = first_char("").unwrap_or('?');
```

For branching behavior, use `match` or `if let`. In reusable code, prefer returning the `Option` or `Result` so the caller keeps the choice.

Tradeoff alternative: `expect("non-empty username")` extracts the value and panics on absence. It is fine when absence proves a programmer invariant was broken, and useful in small examples. It is a poor response to routine user input or network failure because it terminates the current thread instead of reporting a recoverable outcome.

MINI CHALLENGE

Complete the function without `unwrap`, `expect`, or a manual `match`:

```
fn doubled(text: &str) -> Result<i32, std::num::ParseIntError> {
    // parse an i32, then double it
    todo!()
}
```

✓ **ANSWER**

```
fn doubled(text: &str) -> Result<i32, std::num::ParseIntError> {
    let number = text.parse::<i32>()?;
    Ok(number * 2)
}
```

A shorter alternative is `text.parse::<i32>().map(|number| number * 2)`. The `?` version is often easier to extend with another fallible step; `map` is compact for one transformation.

Chapter 9 — Traits

GOAL

Name a capability that different concrete types can provide.

A trait is a behavioral contract. Implementations supply the behavior for particular types.

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

struct Note {
    text: String,
}

fn print_summary(item: &impl Summary) {
    println!("{}", item.summary());
}

fn main() {
    let note = Note { text: String::from("deploy complete") };
    print_summary(&note);
}
```

? PREDICTION

What happens?

- A. It prints the note text automatically.
- B. It fails because `Note` does not implement `Summary`.
- C. It fails because traits cannot return `String`.
- D. It prints `Note` using debug formatting.

→ REVEALB

Representative compiler evidence:

```
error[E0277]: the trait bound `Note: Summary` is not satisfied
  |
  |     print_summary(&note);
  |     ------------- ^^^^^ the trait `Summary` is not implemented for `Note`
```

Defining a trait does not opt types into it. Rust requires an explicit implementation.

MENTAL MODEL

Working model: a trait says, "any type used here must provide these operations." A trait bound lets the compiler reject callers that lack them.

Precise model: trait dispatch has two common forms. Generic parameters and `impl Trait` normally use static dispatch: the compiler knows the concrete type and can generate specialized machine code. A trait object such as `&dyn Summary` uses dynamic dispatch through runtime metadata and permits different concrete types behind one interface.

Implement the smallest contract:

```
trait Summary { fn summary(&self) -> String; }
struct Note { text: String }

impl Summary for Note {
    fn summary(&self) -> String {
        format!("Note: {}", self.text)
    }
}
```

A trait may provide a default method:

```
struct Note { text: String }

trait Summary {
    fn title(&self) -> &str;

    fn summary(&self) -> String {
        format!("Summary: {}", self.title())
    }
}

impl Summary for Note {
    fn title(&self) -> &str {
        &self.text
    }
}
```

Defaults are useful when the shared behavior is genuinely correct. They are not a substitute for choosing a coherent contract.

Rust's coherence rule prevents arbitrary overlapping implementations. In practical terms, your crate may implement your trait for any suitable type, or an external trait for your local type. It generally cannot implement an external trait for an external type. This keeps trait selection globally predictable.

SIMPLEST FIX

Write `impl Summary for Note` and implement every required method. The compiler then checks the contract at the call site and the implementation site.

Tradeoff alternative: avoid a trait and write an inherent `Note::summary` method if only `Note` needs the behavior. That is less machinery and often the right first version. Introduce a trait when code genuinely needs to accept multiple implementations, or when the trait itself carries useful meaning.

MINI CHALLENGE

Make both calls compile without changing `announce`:

```
trait Named {
    fn name(&self) -> &str;
}

struct User(String);
struct Service {
    name: String,
}

fn announce(value: &impl Named) {
    println!("ready: {}", value.name());
}

fn main() {
    announce(&User(String::from("Mira")));
    announce(&Service { name: String::from("search") });
}
```

✓ **ANSWER**

```
trait Named { fn name(&self) -> &str; }
struct User(String);
struct Service { name: String }

impl Named for User {
    fn name(&self) -> &str {
        &self.0
    }
}

impl Named for Service {
    fn name(&self) -> &str {
        &self.name
    }
}
```

The implementations may retrieve names differently. Callers depend only on the promised behavior.

Chapter 10 — Generics and Trait Bounds

GOAL

Reuse one implementation across types while stating exactly which operations it needs.

A generic parameter is a type placeholder chosen by the caller. The function body must work for every type allowed by its bounds.

```
fn larger<T>(left: T, right: T) -> T {
    if left > right { left } else { right }
}

fn main() {
    println!("{}", larger(4, 9));
}
```

? PREDICTION

Does this compile?

- A. Yes; every Rust value supports >.
- B. No; `T` has no declared ordering capability.
- C. No; generic functions cannot return `T`.
- D. Yes, but only for integers.

→ REVEALB

Representative evidence:

```
error[E0369]: binary operation `>` cannot be applied to type `T`
  |
  |     if left > right { left } else { right }
  |        ---- ^ ----- T
  |
help: consider restricting type parameter `T` with trait `PartialOrd`
```

The call happens to use integers, but the function definition claims to accept every `T`. The body is checked against that claim.

MENTAL MODEL

Working model: `<T>` says the type may vary. A bound such as `T: PartialOrd` states the capability the implementation requires.

Precise model: bounds participate in type checking and trait selection; they are not runtime tests. Rust usually monomorphizes generic code, producing concrete versions for the types used. `PartialOrd` is enough for `>` but does not promise that every pair is meaningfully ordered—for example, floating-point `NaN` is unordered. The bound should match the operation, not an imagined category such as "number."

The smallest corrected function is:

```
fn larger<T: PartialOrd>(left: T, right: T) -> T {
    if left > right { left } else { right }
}
```

This consumes both arguments and returns one. If callers must keep both values, borrowing better matches the requirement:

```
fn larger_ref<'a, T: PartialOrd>(left: &'a T, right: &'a T) -> &'a T {
    if left > right { left } else { right }
}
```

The explicit `'a` says the returned reference is valid only for the lifetime shared by both inputs. Lifetime elision cannot choose an input when a function borrows two values; bounds alone do not solve lifetime relationships.

Multiple bounds can be written inline:

```
fn show_larger<T: PartialOrd + std::fmt::Display>(left: T, right: T) {
    let winner = if left > right { left } else { right };
    println!("{winner}");
}
```

Or use a `where` clause when the signature becomes crowded:

```
fn show_larger<T>(left: T, right: T)
where
    T: PartialOrd + std::fmt::Display,
{
    let winner = if left > right { left } else { right };
    println!("{winner}");
}
```

These forms mean the same thing.

SIMPLEST FIX

Add `T: PartialOrd`, the one capability used by `>`. Do not add `Clone`, `Copy`, `Debug`, or `'static` unless the implementation actually needs them. Excess bounds reject valid callers and make later changes harder.

Tradeoff alternative: accept one concrete type, such as `i64`, if that is all the program needs. Concrete code gives clearer error messages and avoids pretending reuse exists. Generalize only when a second real use case appears. If heterogeneous values must be stored together at runtime, a trait object may fit better than a generic collection, at the cost of dynamic dispatch and object-safety restrictions.

MINI CHALLENGE

Why does this fail, and what is the minimum bound?

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

- A. It needs `T: Copy` because the first tuple field moves `value`.
- B. It needs `T: Display` because tuples print values.
- C. It needs `T: PartialEq` because tuple fields must compare.
- D. No bound is needed.

ANSWER

A is the smallest change that preserves this exact body:

```
fn repeat<T: Copy>(value: T) -> (T, T) {
    (value, value)
}
```

That is appropriate for cheaply copyable values such as integers. A broader ownership-oriented version uses cloning:

```
fn repeat<T: Clone>(value: T) -> (T, T) {
    (value.clone(), value)
}
```

`Clone` supports types such as `String`, but cloning may allocate or otherwise be expensive. The function makes that cost part of its contract. If the caller only needs two readers, returning or accepting references can avoid duplication entirely.

The evidence across this part points to one habit: encode the program's real choices and requirements, but no more. Use a struct for one product of fields, an enum for one of several cases, `Option` for absence, `Result` for explained failure, a trait for shared capability, and a generic only when the implementation truly works across types.

Chapter 11 — Iterators and Closures: Computation on Demand

Goal: Read iterator chains as a sequence of delayed transformations, and predict how a closure captures its environment.

Start with a program that looks as if it should print three lines:

```
fn main() {
    let names = ["Ada", "Linus", "Grace"];
    names.iter().map(|name| println!("hello, {name}"));
}
```

? PREDICTION

What happens?

A. It prints all three greetings.
B. It prints one greeting.
C. It prints nothing and compiles quietly.
D. It prints nothing, with a warning that an iterator must be used.

→ REVEALD

The representative warning is:

```
warning: unused `Map` that must be used
  = note: iterators are lazy and do nothing unless consumed
```

`map` builds a value describing future work. It does not perform that work. `Iterator` is centered on one method:

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

Adapters such as `map`, `filter`, and `take` wrap an iterator and implement another `next`. A consumer such as `collect`, `sum`, `fold`, or a `for` loop repeatedly calls `next`.

Working model: an iterator chain is a recipe; a consumer runs it one item at a time.

Precise model: each adapter is a concrete state machine. Rust usually monomorphizes and inlines the chain, so the convenient pipeline need not allocate an intermediate collection. Laziness is about when `next` is called, not about background work.

The simplest fix is to use the operation meant for side effects:

```
fn main() {
    let names = ["Ada", "Linus", "Grace"];
    names.iter().for_each(|name| println!("hello, {name}"));
}
```

A plain loop is often clearer and is equally valid:

```
let names = ["Ada", "Linus", "Grace"];
for name in &names {
    println!("hello, {name}");
}
```

Use iterator adapters when they express a value transformation. Prefer `for` when the body is mostly effects, has several branches, or needs `break` and `continue`.

Closures make these pipelines local. Their capture mode follows use. A closure may borrow immutably, borrow mutably, or take ownership:

```
fn main() {
    let mut total = 0;
    let values = [2, 4, 6];

    values.iter().for_each(|n| total += n);
    println!("{total}"); // 12
}
```

Because the closure mutates `total`, calling it needs mutable access to its captured environment; it implements `FnMut`. A closure that only reads captures can implement `Fn`. A closure that consumes a captured value may only implement `FnOnce`. These traits describe how the closure can be called, not merely whether the source contains `move`.

`move` forces captures by value:

```
let label = String::from("worker");
let show = move || println!("{label}");
show();
```

It does not force every captured value to be destroyed on the first call. Here printing only borrows `label`, so `show` remains callable. `move` is commonly needed when a closure must outlive its creating scope, such as a thread closure.

Iterator ownership follows three familiar entry points:

- `items.iter()` yields `&T`.

- `items.iter_mut()` yields `&mut T`.
- `items.into_iter()` yields owned `T` for an owned collection.

MINI CHALLENGE

What does this print?

```
fn main() {
    let words = vec![String::from("ant"), String::from("bear")];
    let lengths: Vec<usize> = words.iter().map(String::len).collect();
    println!("{} {:?}", words.len(), lengths);
}
```

A. `0 [3, 4]`
B. `2 [3, 4]`
C. It fails because `map` moves each `String`.
D. It fails because `String::len` cannot be a closure.

Answer: B. `iter()` yields `&String`; `String::len` accepts a shared borrow, so the strings remain owned by `words`. Replacing `iter()` with `into_iter()` consumes the vector. That alternative avoids borrowing and is useful when the old collection is no longer needed.

Chapter 12 — Smart Pointers and Shared State

Goal: Choose `Box`, `Rc`, `Arc`, `RefCell`, and `Mutex` by answering two questions: who shares ownership, and where is mutation checked?

Consider this attempt to share a counter with another thread:

```
use std::cell::RefCell;
use std::rc::Rc;

fn main() {
    let count = Rc::new(RefCell::new(0));
    let other = Rc::clone(&count);

    std::thread::spawn(move || *other.borrow_mut() += 1);
}
```

? **PREDICTION**

Why does it fail?

A. `RefCell` can never contain an integer.
B. `spawn` accepts only functions, not closures.
C. `Rc<RefCell<i32>>` is not safe to send between threads.
D. The integer needs to be boxed first.

→ **REVEALC**

Condensed compiler evidence:

```
error[E0277]: `Rc<RefCell<i32>>` cannot be sent between threads safely
  --> ... std::thread::spawn(move || ...)
  = help: the trait `Send` is not implemented for `Rc<RefCell<i32>>`
```

The pointer names are not a ladder from primitive to advanced. Each buys a specific capability at a specific enforcement point.

Type	Ownership	Mutation rule	Thread use
`Box<T>`	one owner	normal `&` / `&mut`	if `T` permits
`Rc<T>`	shared, reference counted	shared access only by itself	single-thread only
`Arc<T>`	shared, atomic reference counted	shared access only by itself	cross-thread if `T` permits

Type	Ownership	Mutation rule	Thread use
`RefCell<T>`	does not add ownership	borrow rule checked at runtime	single-thread only
`Mutex<T>`	does not add ownership	one lock guard at runtime	cross-thread synchronization

`Box<T>` puts a value on the heap while retaining one owner. Its important use is often not "large data," but giving a recursive or trait-object value a known size:

```
enum List {
    Node(i32, Box<List>),
    End,
}
```

Without `Box`, `List` would contain another full `List` forever and have no finite compile-time size.

`Rc<T>` allows multiple owners in one thread. Cloning an `Rc` increments a count; it does not clone `T`. When the last strong owner drops, `T` drops. Reference-count cycles leak, so parent links in graph-like structures often use `Weak<T>`.

`Arc<T>` provides the same ownership shape with atomic count updates. Atomics make the count safe across threads, but do not make `T` mutable or automatically thread-safe. `Arc<RefCell<T>>` still fails because `RefCell<T>` is not `Sync`.

`RefCell<T>` moves the shared-versus-exclusive borrow check to runtime. `borrow()` and `borrow_mut()` return guards. Conflicting guards panic:

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

let value = RefCell::new(5);
let read = value.borrow();
let _write = value.borrow_mut(); // panics: already borrowed
println!("{read}");
```

This is **interior mutability**: an outer shared reference can initiate mutation because the inner type enforces the rule. It is useful when an API must expose `&self` but internally updates caches, mocks, or graph nodes. It is not a way to disable borrowing rules.

`Mutex<T>` also provides interior mutability, but blocks competing threads rather than panicking on ordinary contention. Locking returns a guard that acts like `&mut T`; dropping the guard unlocks. If a thread panics while holding the lock, later `lock` calls report poisoning through `Result`.

The simplest fix for the original program is `Arc<Mutex<T>>`:

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

fn main() {
    let count = Arc::new(Mutex::new(0));
    let other = Arc::clone(&count);

    let handle = std::thread::spawn(move || {
        *other.lock().unwrap() += 1;
    });

    handle.join().unwrap();
    println!("{}", *count.lock().unwrap()); // 1
}
```

Keep guards short; holding a lock while doing slow work increases contention and can create deadlocks. The main tradeoff alternative is message passing with `std::sync::mpsc`: one thread owns the state and others send updates. That avoids shared mutation but introduces a protocol and channel failure handling.

Working model: pointer type chooses ownership; cell or lock type chooses how mutation is coordinated.

Precise model: `Send` permits ownership transfer between threads, while `Sync` permits shared references across threads. These are unsafe marker traits implemented by types whose internals uphold the required guarantees.

MINI CHALLENGE

Choose the smallest fitting type for a syntax tree where many nodes share immutable source text, all work stays on one thread, and the text never changes.

A. `Box<String>`
B. `Rc<String>`
C. `Arc<Mutex<String>>`
D. `RefCell<String>`

Answer: B. The requirement is shared ownership, single-threaded, immutable data. `Rc<String>` states exactly that. `Arc<String>` is a valid tradeoff if the tree later crosses threads, but its atomic counting pays for a capability not currently required.

Chapter 13 — Lifetimes Are API Relationships

Goal: Read lifetime parameters as relationships among references, not as instructions to keep values alive.

Here is the classic function whose intended relationship is missing:

```
fn longer(a: &str, b: &str) -> &str {
    if a.len() >= b.len() { a } else { b }
}
```

? PREDICTION

What does the compiler need?

A. A heap allocation for the result.
B. A lifetime connecting the returned reference to the inputs.
C. Both inputs changed to `String`.
D. A `'static` annotation.

→ REVEALB

Representative evidence:

```
error[E0106]: missing lifetime specifier
  --> ... -> &str
  = help: this function's return type contains a borrowed value,
    but the signature does not say whether it is borrowed from `a` or `b`
```

The function body clearly chooses one input, but callers are checked from the signature. The smallest fix states that both accepted input borrows and the output share a lifetime parameter:

```
fn longer<'a>(a: &'a str, b: &'a str) -> &'a str {
    if a.len() >= b.len() { a } else { b }
}
```

This does not extend either string's life. At a call site, `'a` becomes a lifetime no longer than both input borrows. Therefore the returned reference may be used only within their overlap.

```
fn longer<'a>(a: &'a str, b: &'a str) -> &'a str {
    if a.len() >= b.len() { a } else { b }
}

fn main() {
    let outer = String::from("outside");
    let result;
    {
        let inner = String::from("in");
        result = longer(&outer, &inner);
        println!("{result}"); // valid inside the overlap
    }
    // println!("{result}"); // rejected: it might refer to `inner`
}
```

Working model: a lifetime annotation draws a line between borrowed inputs and outputs.

Precise model: a reference type includes a region during which it is valid. A generic lifetime parameter constrains those regions. `fn longer<'a>(...) -> &'a str` promises the output is valid for the chosen `'a`; because either input may be returned, each input must satisfy that promise. Lifetimes describe provenance and valid use. They do not alter runtime behavior and usually generate no runtime data.

Lifetime elision is why most functions need no written annotations. In `fn first(s: &str) -> &str`, the single input reference lifetime is assigned to the output. In methods, an output reference is usually tied to `&self`. Elision is a fixed signature shorthand, not inference from arbitrary function bodies.

Struct lifetimes express the same relationship:

```
struct Heading<'a> {
    text: &'a str,
}

impl<'a> Heading<'a> {
    fn text(&self) -> &str {
        self.text
    }
}
```

`Heading<'a>` cannot outlive the text it borrows. It does not own that text. Use this when borrowing avoids copying and the owner naturally outlives the view.

A tradeoff alternative is to return ownership:

```
fn longer_owned(a: &str, b: &str) -> String {
    if a.len() >= b.len() { a.to_owned() } else { b.to_owned() }
}
```

This is simpler for callers to store because the result is independent, but it allocates and copies. Do not add lifetimes merely to avoid every clone; choose borrowing when the API's natural meaning is a view into caller-owned data.

Avoid reaching for `'static` as a repair. `&'static str` means the referenced data is valid for the entire program, as string literals are. It does not mean “keep this local reference as long as needed.” Requiring

`'static` would reject ordinary borrowed `String` data rather than explain its relationship.

MINI CHALLENGE

Which signature can return the first argument without unnecessarily tying it to the second?

A. `fn first<'a>(a: &'a str, b: &'a str) -> &'a str`
B. `fn first<'a, 'b>(a: &'a str, b: &'b str) -> &'a str`
C. `fn first(a: &str, b: &str) -> &'static str`
D. No borrowed signature can do this.

Answer: B. The output comes only from `a`, so its lifetime should relate only to `a`. In real code write `fn first<'a>(a: &'a str, _b: &str) -> &'a str`; the second lifetime can be elided because it is unrelated. Option A compiles, but can shorten the usable output to the overlap with `b`.

Chapter 14 — Async Rust Without Magic

Goal: Understand an async function as a value implementing `Future`, and see the minimum work an executor performs using only `std`.

```
async fn answer() -> u32 {
    42
}

fn main() {
    let value = answer();
    println!("{value}");
}
```

? **PREDICTION**

Why does this fail?

A. Async functions require an external crate to compile.
B. `value` is a future, not a `u32`, and futures do not implement `Display`.
C. Async functions can return only `()`.
D. `answer` runs on another thread and has not finished.

→ **REVEALB**

Condensed evidence:

```
error[E0277]: `impl Future<Output = u32>` doesn't implement `std::fmt::Display`
```

Calling an `async fn` constructs a future. It does not run the body to completion and does not create a thread. Conceptually, the compiler converts the function into a state machine storing locals that must survive across `.await` points.

The core protocol is small:

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

`Poll::Ready(output)` means done. `Poll::Pending` means not ready; before returning it, a useful future arranges for the context's `Waker` to be notified when polling may make progress. An executor owns runnable futures, polls them, sleeps or does other work when they are pending, and polls them again after wakeups.

Here is a minimal demonstration. It is not a production executor; it manually polls a future known to complete immediately:

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

async fn answer() -> u32 {
    42
}

fn main() {
    let mut future = pin!(answer());
    let waker = Waker::noop();
    let mut context = Context::from_waker(waker);

    match future.as_mut().poll(&mut context) {
        Poll::Ready(value) => println!("{value}"),
        Poll::Pending => println!("not ready"),
    }
}
```

Output:

```
42
```

`Pin` prevents moving a future after polling begins. Compiler-generated futures may contain state whose correctness depends on a stable location. Pinning does not mean “heap allocate”: `pin!` pins this value in its stack scope.

Working model: a future is paused work; an executor repeatedly asks whether it can advance.

Precise model: a future is a poll-based state machine. `.await` polls the awaited future and, on `Pending`, suspends the enclosing state machine. The waker is a scheduling callback, not the result and not a thread. Async concurrency can interleave many tasks on one thread; parallel CPU execution requires multiple threads.

The simplest practical fix depends on context. Inside another async function, use `.await`:

```
async fn answer() -> u32 { 42 }

async fn report() {
    let value = answer().await;
    println!("{value}");
}
```

At the synchronous program boundary, a real application normally uses an executor. The standard library intentionally provides the `Future` protocol but no general `block_on`, reactor, timers, or async

file/network runtime. The tradeoff is either to select a runtime when the application needs those facilities, or keep the operation synchronous. Writing a general executor is systems work, not a shortcut around a dependency.

Cancellation is usually dropping a future. Its stored locals are dropped, but external effects already performed are not rolled back. Also, blocking calls inside `async fn` still block the executor thread; `async` does not convert blocking I/O into nonblocking I/O.

MINI CHALLENGE

An async function prints before its first `.await`. When does that print occur?

A. When the future is created by calling the function.
B. When the future is first polled.
C. On a newly created thread.
D. Only when the future is dropped.

Answer: B. Calling an async function creates its state machine. Polling begins executing its body. If it reaches a pending `.await`, execution pauses there. An eager API could perform work before returning a future, but an ordinary `async fn` is lazy.

Chapter 15 — Unsafe Rust: A Proof Boundary

Goal: Use `unsafe` as a narrow place where humans establish invariants, then expose an ordinary safe API whose guarantees are checkable.

This function tries to skip bounds checking:

```
fn first(bytes: &[u8]) -> u8 {
    unsafe { *bytes.get_unchecked(0) }
}

fn main() {
    println!("{}", first(&[]));
}
```

? **PREDICTION**

What is the correct diagnosis?

A. The program safely returns zero.
B. Rust inserts a bounds check despite `get_unchecked`.
C. Calling the safe function can cause undefined behavior, so the abstraction is unsound.
D. Unsafe code is allowed to panic but cannot cause undefined behavior.

→ **REVEALC**

There may be no reliable error message. Undefined behavior means the language places no requirements on what follows: a crash, an invented value, apparent success, or miscompilation are all possible. Documentation for `get_unchecked` supplies the evidence: the index must be in bounds, even if the resulting reference is not used.

The dangerous part is not merely the `unsafe` block. The public function is safe to call with any `&[u8]`, yet its implementation requires a nonempty slice. It therefore promises more than it proves.

The simplest fix uses the safe standard operation:

```
fn first(bytes: &[u8]) -> Option<u8> {
    bytes.first().copied()
}
```

That is the preferred solution: no unsafe proof is needed, and absence is represented in the type. If an API requires a value and empty input is a caller error, another safe option is:

```
fn first_required(bytes: &[u8]) -> u8 {
    bytes[0]
}
```

This panics on empty input, but panic is defined behavior. It is not memory unsafety.

When measurement proves bounds checking matters in a larger operation, a sound wrapper can validate once before unchecked access:

```
fn first_required_fast(bytes: &[u8]) -> u8 {
    assert!(!bytes.is_empty(), "expected at least one byte");
    // SAFETY: the assertion proves index 0 is in bounds.
    unsafe { *bytes.get_unchecked(0) }
}
```

The comment states the exact invariant, not "this seems safe." This version is mostly pedagogical: optimization commonly removes redundant checks, so benchmark before keeping it.

Unsafe Rust permits five categories of operation that safe Rust rejects:

1. Dereference a raw pointer.
2. Call an unsafe function or method.
3. Access or modify a mutable static.
4. Implement an unsafe trait.
5. Access fields of a `union`.

`unsafe` does not turn off the borrow checker for ordinary references, and it does not legalize undefined behavior. It says, "the unchecked obligations of these specific operations have been proved by the programmer."

Working model: an unsafe block is a small proof obligation surrounded by compiler-checked code.

Precise model: safe Rust's guarantees are conditional on every unsafe implementation being sound. Safe code cannot by itself cause undefined behavior, but safe code may reach unsound unsafe code through a safe wrapper. Sound unsafe code must handle every input and call pattern its safe API permits, including adversarial ones.

Safe Rust prevents data races, dangling references, invalid reference aliasing, and many other sources of undefined behavior. It does not prevent memory leaks, deadlocks, integer overflow policy mistakes, logical races, panics, or incorrect business results. "Memory safe" is powerful, not synonymous with "bug free."

Keep unsafe regions narrow, but optimize for a narrow **reasoning boundary**, not the lowest line count. Put validation in safe code, isolate pointer manipulation, document invariants, and test boundary cases. For foreign-function interfaces, check the other language's contract: nullability, alignment, ownership, lengths, thread rules, and whether callbacks may outlive their data.

A tradeoff alternative to writing unsafe internals is to change representation or API so safe operations express the invariant. For example, accept `&[u8; 1]` when exactly one byte is structurally required, or return `Option`. Stronger types often delete the proof burden entirely.

MINI CHALLENGE

Is this safe wrapper sound?

```
fn byte_at(bytes: &[u8], index: usize) -> Option<u8> {
    if index < bytes.len() {
        // SAFETY: checked immediately above; `bytes` is unchanged.
        Some(unsafe { *bytes.get_unchecked(index) })
    } else {
        None
    }
}
```

A. No, every public function containing `unsafe` must itself be `unsafe`.
B. No, raw access always invalidates the slice.
C. Yes, the bounds check establishes the documented precondition.
D. Yes, but only for ASCII bytes.

Answer: C. The wrapper accepts all slices and indices, checks the unchecked operation's requirement, and returns `None` otherwise. It can remain a safe function. The simpler `bytes.get(index).copied()` should still be preferred unless evidence justifies the unsafe implementation: identical meaning, less proof to maintain.

Part III's unifying question is not “which advanced feature should I use?” It is “where is the guarantee enforced?” Iterators encode delayed steps in types; smart pointers divide ownership from mutation; lifetimes expose reference relationships; futures expose a polling protocol; unsafe code marks obligations the compiler cannot verify. Choose the smallest mechanism that states the real relationship, and let evidence—not ceremony—justify anything more.

Practice Projects

Each project adds one twist. Use only the standard library.

1. Word ledger

Read text from command-line arguments and print each distinct word with its count.

Constraints:

- Normalize words with `to_lowercase`.
- Use `HashMap<String, usize>`.
- Do not clone a word after inserting it.
- Return a useful error when no text is supplied.

Evidence to collect:

```
$ cargo run -- one fish two fish
fish: 2
one: 1
two: 1
```

Sort before printing so output is deterministic.

2. Command parser

Parse these commands into an enum:

```
add <left> <right>
echo <text>
quit
```

Constraints:

- Parsing returns `Result<Command, String>`.
- Execution uses an exhaustive `match`.
- Bad integers and wrong argument counts produce different messages.

Write one test containing a successful command and one rejected command.

3. Shared counter

Spawn four threads. Each increments one shared counter 1,000 times.

Constraints:

- Use `Arc<Mutex<usize>>`.
- Join every thread.
- Explain why `Rc<RefCell<usize>>` is rejected.
- The final assertion is `4_000`.

4. Borrow-first refactor

Take a program that accepts `String` parameters and returns cloned strings. Change inputs to `&str` wherever ownership is unnecessary.

Keep owned return values only when the result is newly constructed or must outlive all inputs. Use compiler errors to justify each remaining `String`.

Completion check

A project is done when:

```
cargo fmt --check
cargo clippy --all-targets -- -D warnings
cargo test
```

passes and you can answer:

- Who owns each heap allocation?
- Which functions borrow rather than own?
- Which invalid states are represented by the type system?
- Which failures are returned as `Result`?

Compiler Field Guide

Rust errors are usually relationship errors. Start with the question matching the diagnostic.

Diagnostic clue	Ask this first	Common smallest fix
`use of moved value`	Who owns the value now?	Borrow it, or clone only if two owners are required.
`cannot borrow ... as mutable`	Is the binding mutable, and is another borrow active?	Add `mut`, or shorten the other borrow's last use.
`cannot borrow ... more than once`	Can two writers reach the same value?	Finish the first mutable borrow before creating the next.
`does not live long enough`	What value would the reference outlive?	Return owned data or move the owner outward.
`missing lifetime specifier`	Which input does the returned reference come from?	Express that relationship with a lifetime parameter.
`mismatched types`	What exact type does this expression produce?	Convert at the boundary; do not cast blindly.
`trait bound ... is not satisfied`	Which operation requires which capability?	Add the narrow bound, or stop asking the generic code to do that operation.
non-exhaustive `match`	Which valid state is untreated?	Handle the missing variant explicitly.
future cannot be sent safely	What non-`Send` value crosses an `.await`?	Drop it before `.await`, or use an appropriate thread-safe type.

A useful reading order

1. Read the short error headline.
2. Find the source line marked by the primary span.
3. Read the first `note`; it often explains the relationship.
4. Apply the smallest fix that matches your intent.
5. Re-run `cargo check`. Later diagnostics may be consequences of the first.

Do not respond to ownership errors with automatic `.clone()`. That makes the compiler quiet by changing cost and ownership. Decide whether the code needs another owner first.

Commercial edition

Title: Rust by Evidence
Subtitle: Learn ownership, types, and concurrency by reading what the compiler proves
Author: winterloo
Edition: 1.0 (2026)
Language: English
Format: responsive website + PDF ebook