https://simplifycpp.org

Designing High-Performance and Safe Systems with Rust

A Deep, Practical, and Example-Driven Guide

DRAFT

Prepared by Ayman Alheraki

Designing High-Performance and Safe Systems with Rust

A Deep, Practical, and Example-Driven Guide

Some drafting assistance and idea exploration were supported by modern AI tools, with full supervision, verification, correction, and authorship.

Prepared by Ayman Alheraki

February 2026

Contents

Author's Introduction

I have been a **C++ programmer since 1991**. Over these decades, I have lived through nearly every phase of its evolution—from its early, demanding days, through periods of near-total reliance on it, to the modern era of diverse programming languages and paradigms. Despite everything that has been said about C++, and despite the very real difficulties that anyone who has worked deeply with it knows well, I still love this language. I deeply respect its power, flexibility, and its exceptional ability to build real systems that cannot afford failure.

Since around **2005**, the programming language landscape has changed dramatically. New languages such as **Go** emerged, **Python 3** underwent significant evolution—not only in the language itself, but also in compilation mechanisms, libraries, package managers, and the overall developer experience. Then came **Rust**, offering a fundamentally different model for memory management, safety, and concurrency, while still delivering high performance. Even **Zig**, despite its relative youth, introduced bold ideas around simplicity, explicit control, and clarity.

I do not hide the fact that, as a C++ programmer, I often feel a certain technical envy toward these modern ecosystems: unified package managers, cohesive build systems, clearer compiler diagnostics, and memory safety by default. These are areas where C++, even in its modern post-C++11 form, still struggles to provide a unified and enforced model—despite the significant advances the language has made, including major improvements in memory management techniques.

Nevertheless, I remain a strong advocate of **C++**. I write extensively about it and defend it from a position of experience and understanding, not blind loyalty. At the same time, I have full respect for all programming languages and firmly believe that every language has its rightful place when used in the appropriate context.

I have attempted to work with **Rust** more than once, and each time I found it to be an outstanding language. It intelligently addresses many of the structural problems that C++ has struggled with for decades, particularly in the areas of memory safety, concurrency, and correctness. However, I did not continue at that time. My most recent related work was a book comparing **C++** and **Rust**, but I later realized that I had not yet reached a level of mastery in Rust that would allow for a truly fair comparison in large-scale, complex systems.

That realization is what led to this book.

With the beginning of the research and preparation for this work, I made a conscious decision to dive deeply into **Rust**—not only at the level of syntax or small examples, but through designing real systems, services, and libraries using the language. The goal was neither to abandon C++ nor to prove the superiority of one language over another, but to reach **technical certainty**: to truly understand what Rust offers in practice, where it excels, where it still needs maturity, and how it can be professionally compared to a deeply mature and widely adopted language like C++.

I hope to continue this journey until I reach true professional mastery of **Rust**, and to add it alongside **C++** as a core tool within my long-standing experience. I see no conflict in working with both languages; on the contrary, I see this as a natural and pragmatic form of integration—choosing the right tool for the right problem.

This book is part of that journey. A journey of learning, experimentation, and honest comparison—free from bias and preconceived judgments—guided only by sound engineering principles and real-world experience.

Ayman Alheraki

Book Introduction

Why Rust, and why now?

Rust did not appear because developers suddenly disliked C and C++. It appeared because the modern software world reached a point where the **cost of memory-unsafe bugs** became unacceptable in many domains: browsers, cryptography, operating-system components, networking stacks, infrastructure, embedded firmware, and long-running services where a single undefined-behavior bug can become a security incident or a costly outage.
Rust offers a practical answer to a difficult equation:

- **C/C++-level control over memory and performance** (no mandatory garbage collector, explicit data layout tools, predictable runtime model),
- **but with a stronger default safety model** enforced at compile time (ownership, borrowing, lifetimes),
- **and with a modern ecosystem workflow** (package management, testing, formatting, linting, documentation) that pushes teams toward consistent, maintainable code.

A major part of "why now" is maturity:

- Rust has stabilized multiple language editions (including a recent one), which demonstrates a deliberate model for evolving the language while keeping codebases maintainable over time.

- The unsafe story is clearer: unsafe is not “forbidden”; it is **isolated**, documented, and wrapped behind safe abstractions where possible.
- The tooling around Rust (builds, testing, documentation, static analysis) has become a serious production environment rather than an academic experiment.

A concrete problem Rust targets

Many real-world failures in systems software trace back to a small set of recurring bug classes:

- Use-after-free and double-free,
- Buffer overflows and out-of-bounds indexing,
- Null pointer dereference,
- Data races and shared-mutable-state bugs,
- Iterator invalidation and aliasing-related undefined behavior.

Rust does not magically remove all bugs. It **systematically reduces entire categories** of these bugs in safe Rust by making “invalid states” harder or impossible to express.

Example: a classic use-after-free in C (conceptual)

```c
/* A simplified example of a dangling pointer */
#include <stdlib.h>
#include <stdio.h>

int* make_ptr(void) {
    int* p = (int*)malloc(sizeof(int));
    *p = 42;
    free(p);
    return p; /* returns a dangling pointer */
}

int main(void) {
    int* x = make_ptr();
    /* Undefined behavior: x points to freed memory */
    printf("%d\n", *x);
    return 0;
}
```

In Rust, safe code is designed so that ownership and lifetimes prevent this pattern from compiling in the first place.

Rust’s direction: make the safe path the fast path

Rust’s core design encourages developers to write code that is:

- **safe by default**,
- **explicit where it matters** (mutability, ownership transfer),
- and **zero-cost in abstraction style** (you can write high-level code without accepting hidden runtime penalties in typical cases).

Where Rust outperforms C/C++ and where it deliberately overlaps

Where Rust outperforms (especially for teams and long-lived systems)

1) Memory safety by construction (safe Rust).
The ownership/borrowing system makes many memory-unsafe states unrepresentable without explicitly opting into unsafe.
2) Data-race resistance in safe concurrency.
Rust's type system makes it difficult to share mutable state across threads without synchronization primitives designed for concurrency.
3) Stronger refactoring safety.
In large projects, "it compiles" becomes a stronger signal: many lifetime, aliasing, and thread-safety invariants are enforced at compile time.
4) A cohesive production toolchain culture.
Formatting, linting, testing conventions, and packaging workflows are integrated into daily practice in a way that reduces friction in large teams.

Where Rust deliberately overlaps with C/C++ (because reality demands it)

Rust is not a "pure safe language at all costs". It intentionally keeps the doors open for systems-level reality:
1) Unsafe Rust exists for low-level control.
Some problems require raw pointers, custom allocation strategies, hardware interaction, or performance-critical tricks that cannot be expressed under the strict aliasing and borrowing rules. Rust allows this via `unsafe`, but expects developers to:

- isolate unsafe code,

- document invariants,
- provide a safe API boundary,
- and audit carefully.

2) FFI is a first-class capability.
Rust is designed to interoperate with C (and, with a disciplined boundary, with C++). Many practical Rust systems integrate with existing native libraries.
3) Manual control is still available.
You can control representation, alignment, allocation strategies, and performance-sensitive patterns. Rust does not force a heavy runtime or a garbage collector.

Example: safe vs controlled-unsafe boundary

A common professional pattern is: write a **small unsafe core** and expose a **safe wrapper**.

```rust
/* Conceptual pattern: unsafe internally, safe externally */
pub struct Buffer {
    ptr: *mut u8,
    len: usize,
}

impl Buffer {
    pub fn len(&self) -> usize { self.len }

    pub fn get(&self, i: usize) -> Option<u8> {
        if i < self.len {
            unsafe { Some(*self.ptr.add(i)) }
        } else {
            None
        }
    }
}
```

This overlap is a strength, not a weakness: Rust acknowledges that systems programming needs escape hatches, but it makes them explicit and auditable.

What Rust truly promises: memory safety, performance, and long-term maintainability

1) Memory safety (in safe Rust) without a garbage collector

Rust's promise is not "no bugs". Its promise is more precise:

- In **safe Rust**, many categories of memory safety violations are prevented by compile-time rules.
- The compiler enforces a disciplined model of ownership and borrowing so references do not outlive the data they point to.

Example: ownership prevents accidental double free (conceptual)

In C++ you might accidentally implement a type that copies raw pointers and frees them twice. Rust makes "who owns what" explicit: moves transfer ownership, copies are limited to types that are safe to copy.

```
fn main() {
    let s = String::from("hello");
    let t = s;
    println!("{}", t);
}
```

2) Performance: modern abstractions with a systems-level runtime model

Rust aims to give you:

- predictable cost models,
- no mandatory GC pauses,

- strong optimization opportunities via static typing and monomorphization,
- expressive high-level code (iterators, pattern matching) that can compile efficiently.

Example: iterator style without forcing you into slow code

```rust
fn sum_positive(xs: &[i32]) -> i32 {
    xs.iter()
      .copied()
      .filter(|&x| x > 0)
      .sum()
}
```

3) Long-term maintainability: the “engineering promise”

Rust’s maintainability is not marketing; it is structural:

- Ownership/borrowing makes resource lifetimes explicit (files, sockets, locks, buffers).
- **Enums and pattern matching** encourage modeling real state machines directly rather than scattered boolean flags.
- Error handling encourages explicit failure paths rather than silent undefined behavior or unchecked return values.
- Concurrency primitives integrate with the type system, reducing accidental thread-unsafe designs.

Example: modeling states with enums (reduces impossible states)

```rust
enum ConnectionState {
    Disconnected,
    Connecting { retries: u32 },
    Connected { peer: String },
    Closing,
}

fn describe(s: &ConnectionState) -> &'static str {
    match s {
        ConnectionState::Disconnected => "offline",
        ConnectionState::Connecting { .. } => "handshaking",
        ConnectionState::Connected { .. } => "online",
        ConnectionState::Closing => "shutting down",
    }
}
```

How to use this book: beginner path, C++ expert path, systems path

This book is intentionally designed to serve **three distinct audiences** without diluting technical depth or lowering professional standards. Rust is a language whose real strength does not lie in surface-level syntax, but in its underlying **computational and ownership model**. For this reason, regardless of your background, skipping the core principles will significantly weaken your long-term understanding and practical effectiveness.

You should treat this book not as a linear tutorial, but as a **guided technical map**. Each path emphasizes different priorities, yet all paths converge on the same foundations: ownership, borrowing, lifetimes, explicit state modeling, and disciplined interaction with memory and concurrency.

General guidance before choosing a path

Before selecting a specific reading path, it is important to understand several global principles that apply to every reader:

- Rust rewards **conceptual clarity over memorization**. Understanding *why* the compiler rejects certain programs is more important than learning how to silence errors.
- Rust is best learned by **writing many small, focused programs** rather than a few large ones early on.
- Performance, safety, and maintainability are not separate goals in Rust; they are deeply interconnected through the language model.
- The compiler is not an obstacle; it is an enforcement mechanism for invariants that would otherwise require discipline, documentation, and code reviews.

With these principles in mind, choose the path that best matches your background and goals.

Path A: Beginner (new to systems programming)

This path is intended for readers who may have experience in high-level languages or are new to systems-level concepts such as manual memory management, explicit lifetimes, and low-level concurrency.

Primary objective: build a correct mental model of ownership, borrowing, and explicit resource management before focusing on performance or advanced abstractions.

Recommended reading strategy

- Begin with Rust syntax and expressions only as a vehicle to understand ownership and immutability.
- Spend significant time on ownership transfer, borrowing rules, and lifetimes. These chapters should not be rushed.
- Progress to enums, pattern matching, and error handling to learn how Rust models real-world states safely.
- Introduce concurrency only after understanding why shared mutable state is restricted.
- Finish with at least one complete capstone project to consolidate learning.

Practice methodology

- Write many short programs (approximately 50–150 lines) that isolate a single concept.
- Avoid using `clone()` as a default solution; treat each clone as a design decision that must be justified.
- Read compiler error messages carefully and attempt to predict them before compiling.
- Rewrite simple programs multiple times using different ownership strategies.

Expected outcome

By the end of this path, you should be comfortable reasoning about:

- Who owns each piece of data,
- How long references are valid,
- Why certain patterns are rejected at compile time,
- And how Rust prevents entire classes of runtime failures.

Path B: C++ Expert (rapid and precise transition)

This path is designed for experienced C++ developers who already understand low-level memory management, RAII, templates, and concurrency, and want to adopt Rust without fighting the language.

Primary objective: replace familiar C++ idioms with their Rust equivalents while avoiding incorrect mental mappings.

Recommended reading strategy

- Skim basic syntax and focus immediately on ownership and borrowing, as this is the primary conceptual shift.
- Study traits and generics as Rust's alternative to inheritance, interfaces, and template metaprogramming.
- Focus on API boundaries: slices, iterators, error types, and module visibility.
- Learn Rust concurrency primitives with emphasis on how the type system enforces thread safety.
- Study `unsafe` Rust as the controlled equivalent of raw C++ power with explicit contracts.

Mental model translation

- A move in Rust corresponds to transferring unique ownership by default.
- `&T` represents a shared, read-only borrow with a lifetime contract.
- `&mut T` represents exclusive access, enforced statically.
- Traits describe capabilities and contracts, not class hierarchies.
- Enums with data often replace class hierarchies and tagged unions.

Common pitfalls to avoid

- Trying to emulate classic object-oriented inheritance patterns directly.
- Overusing heap allocation where stack-based or borrowed data would suffice.
- Treating the borrow checker as something to be bypassed rather than understood.

Expected outcome

After completing this path, you should be able to:

- Design idiomatic Rust APIs,
- Write performance-critical code without sacrificing safety,
- Confidently decide when `unsafe` is justified and how to contain it.

Path C: Systems Path (OS, embedded, networking, performance-critical software)

This path targets readers building operating system components, embedded firmware, networking stacks, databases, runtimes, or other performance- and correctness-critical systems.

Primary objective: use Rust as a systems language without illusions, while preserving explicit control and predictability.

Recommended reading strategy

- Treat ownership, borrowing, and lifetimes as mandatory prerequisites.
- Learn representation control, memory layout, and FFI boundaries early.
- Study concurrency, atomics, and memory ordering to the depth required by your domain.
- Master `unsafe` Rust patterns and safe abstraction design.
- Focus on profiling, benchmarking, and understanding generated code where necessary.

Practice methodology

- Build real components such as ring buffers, allocators, parsers, protocol state machines, or schedulers.
- Measure performance early and repeatedly; avoid speculative optimization.
- Keep unsafe code minimal, localized, documented, and reviewed.
- Treat safety invariants as part of the API contract.

Expected outcome

By the end of this path, you should be able to:

- Write low-level Rust that competes with C/C++ in performance,
- Integrate Rust safely with existing native codebases,
- Build abstractions that are both safe and efficient.

Final guidance: convergence, not separation

Although these paths differ in emphasis, they are not isolated. A beginner may eventually follow the systems path; a C++ expert may revisit beginner chapters to refine their mental model. The unifying principle of this book is simple but strict:

- Rust is not about writing code that merely compiles.
- Rust is about encoding correctness, ownership, and intent directly into the program structure.

If you respect this principle and follow your chosen path with discipline, the reward is not only safer code, but software that remains understandable, maintainable, and performant under real-world pressure.

Part I

Foundations of Rust’s Power

The Rust Mental Model

Rust is not primarily a new syntax for writing systems software. It is a new **contract** between you and the language: you describe ownership, mutability, and aliasing intent explicitly, and the compiler enforces the invariants that would otherwise be enforced by discipline, conventions, and post-mortem debugging.

This chapter builds the mental model you will use throughout the book. If you internalize it early, later topics (lifetimes, traits, concurrency, `unsafe`, and performance engineering) become natural rather than frustrating.

1.1 Why the compiler is part of your team

In many languages, the compiler is mostly a translator. In Rust, the compiler is also an **invariant checker**. It verifies properties about your program that are traditionally validated only by testing, reviews, or production incidents.

1.1.1 The compiler enforces invariants you would otherwise document

When you write Rust, you are not only writing instructions for the machine; you are also writing **constraints**:

- Who owns this value?

- Who is allowed to mutate it?
- Can two references alias the same memory while one mutates it?
- Can a reference outlive the value it points to?
- Can this value safely cross a thread boundary?

In C and C++, many of these constraints live in human documentation:

- “Caller must keep the buffer alive until callback returns.”
- “This pointer may be null.”
- “This object is not thread-safe.”
- “Do not store references to this parameter.”

Rust aims to move a large portion of these rules into the type system so they become mechanically verified.

1.1.2 The compiler gives earlier feedback than tests can

Tests are essential, but they are not designed to prove the absence of broad categories of bugs. They demonstrate the presence of correctness for chosen scenarios. Rust’s compiler checks structural properties:

- A borrow cannot outlive its owner (prevents many use-after-free patterns in safe code).
- You cannot have aliased mutable access (prevents many data races and iterator invalidation patterns in safe code).
- Moving a value invalidates the previous binding (prevents double-free patterns for uniquely owned resources).

1.1.3 Example: move semantics are a safety guarantee

In Rust, many types are moved by default, which makes ownership explicit.

```
fn main() {
    let s = String::from("hello");
    let t = s;              // move: ownership transferred
    // println!("{}", s);   // error: use of moved value
    println!("{}", t);
}
```

The compiler is not being strict for the sake of strictness. It is telling you that there is exactly one owner of the allocation behind `String`. This is a safety and maintainability contract.

1.1.4 Example: exclusive mutability is a correctness guarantee

Rust forbids simultaneous mutable and shared borrows.

```
fn main() {
    let mut v = vec![1, 2, 3];

    let a = &v[0];      // shared borrow of v
    // v.push(4);       // error: cannot borrow `v` as mutable because it is also borrowed as
    ↪  immutable
    println!("{}", a);
}
```

In C++ a similar pattern can compile and later fail due to iterator invalidation or reallocation. Rust rejects the possibility early because the code expresses an invalid aliasing pattern.

1.1.5 What “part of your team” means in practice

Treat the compiler as an engineering partner:

- It forces clarity: you must decide where ownership lives.
- It forces boundaries: you must decide where mutability is allowed.
- It forces discipline: you must decide how long references remain valid.
- It encourages better architecture: stable APIs tend to use borrowing, slices, and explicit error types.

Your job is not to “fight” the compiler. Your job is to **align your design** with the invariants the compiler is checking.

1.2 Common C/C++ thinking traps and how to unlearn them

Rust is close to C and C++ in performance goals and systems-level capability, but it is intentionally different in how it models correctness. Many frustrations come from carrying old habits into a new model.

1.2.1 Trap 1: “A pointer is just an address”

In C/C++, a pointer is fundamentally an address, and validity is a runtime concern. In Rust, safe references are **not** just addresses:

- `&T` means: a non-null, aligned, valid reference to `T` that remains valid for its lifetime.
- `&mut T` means: the only active reference to that `T` for the duration of the borrow (exclusive access).

How to unlearn it: stop thinking in “addresses” and start thinking in “permissions”:

- shared permission (`&T`) to read,
- exclusive permission (`&mut T`) to mutate,
- ownership permission (the owning value) to destroy and move.

1.2.2 Trap 2: “If it compiles, I will debug lifetime issues later”

In C/C++, lifetime errors compile and appear as intermittent crashes, corruption, or security vulnerabilities. In Rust, lifetime discipline is a first-class design element.
How to unlearn it: design lifetimes at the API boundary:

- Prefer borrowing (`&str`, `&[T]`) when you do not need ownership.
- Prefer returning owned values when the result must outlive the input.
- Use clear ownership transfer instead of storing references casually.

1.2.3 Trap 3: “Make everything mutable, then constrain later”

In C/C++, mutability is the default habit. In Rust, mutability is explicit because it affects aliasing and optimization assumptions.
How to unlearn it: start immutable and promote to mutable only when needed:

```rust
fn main() {
    let x = 10;         // immutable
    let mut y = 10;     // mutable only when required
    y += 1;
    println!("{}, {}", x, y);
}
```

This is not only style; it reduces accidental shared-mutable-state designs.

1.2.4 Trap 4: “Copying is cheap”

In C++ you can copy many types cheaply, and sometimes you accidentally copy expensively. In Rust, most non-trivial types move by default, and copying must be explicit (via `Copy` types or `clone()`).
How to unlearn it:

- Treat `clone()` as an intentional decision, not a quick fix.

- When you see `clone()`, ask: can I borrow instead?

1.2.5 Trap 5: "Shared mutable state is normal"

In C/C++, shared mutable state is common and protected by convention or runtime synchronization. Rust pushes you toward:

- local mutation,

- explicit sharing via `Arc`,

- explicit synchronization via `Mutex`/`RwLock`,

- or message passing via channels.

How to unlearn it: adopt one of two patterns:

- **Ownership transfer** (move values into workers and return results),

- **Controlled sharing** (wrap shared state behind synchronization primitives).

1.2.6 Trap 6: "Design around inheritance"

Rust is not built around class inheritance. It favors:

- composition via structs,

- behavior via traits,

- closed sets of states via enums.

How to unlearn it: model data first, then attach behavior:

```
trait Render {
    fn render(&self) -> String;
}

struct Button { caption: String }
struct Label  { text: String }

impl Render for Button {
    fn render(&self) -> String { format!("<button>{}</button>", self.caption) }
}

impl Render for Label {
    fn render(&self) -> String { format!("<span>{}</span>", self.text) }
}
```

This is not “less OOP”; it is a different decomposition that avoids many hierarchy hazards.

1.2.7 Trap 7: “Use exceptions for error flow”

Rust uses explicit error values, typically `Result<T, E>`, which makes failure paths visible and composable.

How to unlearn it: treat errors as part of the function contract:

```
use std::fs;

fn read_config(path: &str) -> Result<String, std::io::Error> {
    fs::read_to_string(path)
}
```

You will later learn how to structure error types for libraries vs applications, but the mental model starts here: errors are data.

1.3 Treating compiler errors as structured training

Rust compiler errors are not random obstacles. They are structured feedback pointing to a violated invariant. If you approach them correctly, each error becomes a lesson that permanently upgrades your intuition.

1.3.1 A practical workflow for compiler-driven learning

Use this loop:

1. Read the first error carefully, ignore the rest initially.

2. Identify which invariant was violated (move, borrow, lifetime, mutability, trait bound).

3. Predict the minimal change that satisfies the invariant.

4. Apply the change, recompile, repeat.

Over time, you will learn to predict errors before compiling, which is the beginning of real Rust fluency.

1.3.2 Error category 1: move and ownership

When you see errors about “moved value”, the compiler is telling you: “ownership already transferred.”

```rust
fn consume(s: String) {
    println!("{}", s);
}

fn main() {
    let s = String::from("hi");
    consume(s);
    // println!("{}", s); // error: use of moved value
}
```

Training question: should consume own the string, or borrow it?

Borrow fix:

```
fn consume(s: &str) {
    println!("{}", s);
}

fn main() {
    let s = String::from("hi");
    consume(&s);
    println!("{}", s); // ok
}
```

This is a design decision, not a syntax tweak.

1.3.3 Error category 2: conflicting borrows

When you see errors about mutable vs immutable borrows, the compiler is telling you: "you are trying to use incompatible permissions simultaneously."

```
fn main() {
    let mut v = vec![10, 20, 30];
    let first = &v[0];
    v.push(40);              // error: mutable borrow while immutable borrow active
    println!("{}", first);
}
```

Training question: how can you reduce the borrow scope?

Scope fix:

```
fn main() {
    let mut v = vec![10, 20, 30];

    {
        let first = v[0];    // copy the value out (i32 is Copy)
        println!("{}", first);
    }                        // borrow ends here

    v.push(40);              // ok
}
```

Sometimes the correct solution is not copying, but reorganizing scopes and responsibilities.

1.3.4 Error category 3: lifetimes and escaping references

When the compiler rejects a reference that “does not live long enough”, it is protecting you from returning or storing a reference to data that will be dropped.

```
fn bad_ref() -> &str {
    let s = String::from("temp");
    &s   // error: cannot return reference to local variable
}
```

Training question: should the function return an owned value instead?

Owned fix:

```
fn good_value() -> String {
    let s = String::from("temp");
    s
}
```

This is the core lifetime lesson: if output must outlive input, return ownership.

1.3.5 Error category 4: trait bounds and missing capabilities

Many errors are really about missing contracts: “type `T` does not implement trait `X`.”

```
fn print_all<T>(xs: &[T]) {
    for x in xs {
        // println!("{}", x); // error: `T` doesn't implement `Display`
    }
}
```

Training question: what capability do you require?

Trait bound fix:

```
use std::fmt::Display;

fn print_all<T: Display>(xs: &[T]) {
    for x in xs {
        println!("{}", x);
    }
}
```

Again, not a hack: you are encoding a requirement into the type system.

1.4 Lab: turning common errors into learning exercises

This lab is designed to transform typical Rust compiler failures into deliberate practice. Do not rush to “make it compile.” Instead, treat each exercise as training for the mental model.

1.4.1 Lab rules

- Do not use `clone()` unless the exercise explicitly asks you to justify it.

- Prefer solutions that improve the API contract (borrowing, ownership transfer, clearer scopes).
- After each fix, write one sentence describing the invariant you satisfied.

1.4.2 Exercise 1: Move vs borrow at API boundaries

Goal: decide whether a function should take ownership.

```rust
fn log_message(msg: String) {
    println!("{}", msg);
}

fn main() {
    let msg = String::from("server started");
    log_message(msg);
    println!("{}", msg); // fails: moved
}
```

Tasks:

- Fix it by changing `log_message` to borrow.
- Fix it by keeping ownership but restructuring usage (e.g., consume at end).
- For each fix, explain why the ownership model is appropriate.

1.4.3 Exercise 2: Borrow scope and mutation

Goal: shorten a borrow to allow later mutation.

```rust
fn main() {
    let mut data = vec![1, 2, 3, 4];
    let head = &data[0];
    data.push(5); // fails
    println!("{}", head);
}
```

Tasks:

- Fix it by adjusting scopes.
- Fix it by copying out the needed value where valid.
- Fix it by redesigning: compute head after mutation.

1.4.4 Exercise 3: Returning references correctly

Goal: understand why returning a reference to local data is invalid.

```
fn make_name() -> &str {
    let s = String::from("Ayman");
    &s
}
```

Tasks:

- Fix by returning `String`.
- Fix by taking an input buffer and returning a borrow tied to the input.

1.4.5 Exercise 4: Trait bounds as explicit contracts

Goal: make requirements explicit in generic code.

```rust
fn dump<T>(xs: &[T]) {
    for x in xs {
        println!("{:?}", x); // fails: Debug not guaranteed
    }
}

fn main() {
    let a = [1, 2, 3];
    dump(&a);
}
```

Tasks:

- Add the correct trait bound to compile.
- Explain why generics must encode capabilities.

1.4.6 Exercise 5: From "shared mutable state" to disciplined sharing

Goal: shift your design from implicit sharing to explicit sharing.

```rust
use std::thread;

fn main() {
    let mut counter = 0;

    let t1 = thread::spawn(|| {
        counter += 1; // fails: captured variable cannot be shared/mutated this way
    });

    let t2 = thread::spawn(|| {
        counter += 1; // fails
    });

    t1.join().unwrap();
    t2.join().unwrap();

    println!("{}", counter);
}
```

Tasks:

- Rewrite using message passing (each thread sends `1`, main sums).
- Rewrite using explicit sharing with `Arc` and `Mutex`.
- For each solution, state the concurrency invariant that makes it correct.

1.4.7 What you should gain from the lab

After completing these exercises, you should be able to recognize, without hesitation, the meaning behind common compiler diagnostics:

- “use of moved value” → ownership already transferred,
- “cannot borrow as mutable because it is also borrowed as immutable” → permission conflict,
- “does not live long enough” → reference would outlive its owner,
- “trait bound not satisfied” → missing explicit capability contract.

This is the foundation of Rust fluency: you stop seeing compiler errors as failures, and start seeing them as **guided training in correct systems design**.

Tooling and Environment Setup

Rust's tooling is part of the language experience. It is not an afterthought bolted onto a compiler; it is a coherent workflow designed to keep builds reproducible, testing routine, formatting consistent, and performance decisions explicit. This chapter establishes a professional, production-grade baseline that you will reuse throughout the book.

2.1 rustup, toolchains, targets

Rust installations are typically managed through `rustup`, which acts as a toolchain manager. The key idea is simple: you can install, switch, and pin compiler versions (`toolchains`), and you can install platform backends (`targets`) for cross-compilation.

2.1.1 Toolchains: stable, beta, nightly

A **toolchain** is a coherent set of tools: `rustc`, `cargo`, and standard libraries for selected targets. Rust commonly offers three primary channels:

- **stable**: the default for production and for this book.
- **beta**: a preview of the next stable release.
- **nightly**: bleeding-edge features, used when you need unstable compiler flags or experimental language/library features.

You should prefer **stable** unless you have a concrete reason for nightly. Most professional Rust codebases remain stable-first, and isolate nightly usage (if any) behind tooling scripts.

2.1.2 Pinning a toolchain per project

Teams need reproducibility. The simplest reproducible setup is to pin the toolchain at the repository level so every developer and CI run uses the same compiler.
A typical project-local pin uses a file such as `rust-toolchain.toml`:

```
[toolchain]
channel = "stable"
profile = "default"
components = ["rustfmt", "clippy"]
```

This ensures consistent formatting and lint behavior in addition to consistent compilation.

2.1.3 Targets: host vs cross-compilation

A **target** defines the platform triple the compiler generates code for (architecture, vendor, OS, ABI). Your **host** target is your current machine. Cross-compilation is adding other targets and building for them.
Typical workflows:

- Add a target once per machine.
- Build with `–target <triple>` when needed.
- Use CI to continuously validate important targets.

Example commands (conceptual workflow):

```
# Show installed and available toolchains
rustup show

# Install a toolchain channel
rustup toolchain install stable
rustup toolchain install nightly

# Set default toolchain globally
rustup default stable

# Add useful components to the current toolchain
rustup component add rustfmt
rustup component add clippy

# List targets and add a cross-compilation target
rustup target list
rustup target add x86_64-unknown-linux-gnu
rustup target add aarch64-unknown-linux-gnu
```

2.1.4 Profiles: minimal vs default

Rust toolchains can be installed with different profiles:

- **minimal**: installs only the essentials.
- **default**: adds common components.
- **complete**: installs more tools and docs (useful for offline setups).

For team productivity, **default** plus `rustfmt` and `clippy` is a practical baseline.

2.2 cargo: build, run, test, document

`cargo` is Rust's build system and package manager. In practice, it is also the standard project interface for:

- dependency resolution,
- building binaries and libraries,
- running tests,
- generating documentation,
- managing workspaces and feature flags.

2.2.1 Project anatomy

A basic binary crate looks like this:

```
my_app/
  Cargo.toml
  src/
    main.rs
```

A library crate uses `src/lib.rs`. A workspace can contain multiple crates.

2.2.2 The essential commands

The following is the minimal command set you should master early:

```
# Create a new binary crate
cargo new my_cli
```

```
# Build (debug by default)
cargo build

# Run the default binary
cargo run

# Run with args (everything after -- is passed to the program)
cargo run -- --help
cargo run -- sum 10 20

# Run tests
cargo test

# Generate documentation (local docs)
cargo doc

# Build in release mode (optimized)
cargo build --release
```

2.2.3 Dependency management

Dependencies are declared in `Cargo.toml`. Cargo resolves versions and records exact selections in `Cargo.lock`.

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

[dependencies]
```

Even before adding third-party crates, you should treat `Cargo.toml` as a contract:

- keep dependencies intentional,

- avoid unnecessary transitive bloat,
- and review changes to the lock file in code reviews.

2.2.4 Workspaces for multi-crate projects

A workspace is the standard way to structure larger systems. It allows you to:

- share a single lock file,
- build and test multiple crates consistently,
- separate library code from binaries,
- enforce consistent profiles and lints.

```
[workspace]
members = ["crates/core", "crates/cli"]
```

2.3 rustfmt and clippy

Rust treats formatting and linting as first-class components of professional codebases. This is not about style preferences; it is about:

- minimizing diff noise,
- reducing cognitive load,
- catching correctness and performance issues early,
- enforcing consistent idioms across teams.

2.3.1 rustfmt: deterministic formatting

`rustfmt` formats Rust source code to a standard style. Most teams format on save in the editor and enforce formatting in CI.

```
# Format the entire crate
cargo fmt

# Check formatting (CI-style)
cargo fmt -- --check
```

2.3.2 clippy: linting beyond the compiler

`clippy` is a lint collection that catches a broad range of issues:

- suspicious code patterns,
- unnecessary allocations and clones,
- suboptimal iterator usage,
- API design pitfalls,
- correctness hazards (where it can reason safely).

```
# Run clippy for the crate
cargo clippy

# Run clippy with warnings treated as errors (CI discipline)
cargo clippy -- -D warnings
```

2.3.3 A practical rule

Use the compiler to make code correct, and use clippy to make code **cleaner and more idiomatic**. If clippy suggests a change that harms readability or intent, you can justify not taking it; but in general, clippy is a strong guide toward production-quality Rust.

2.4 Practical metrics: build time, profiles, lock files

Systems engineering is not only about code correctness; it is also about build reproducibility and performance discipline. Rust provides explicit switches for these concerns.

2.4.1 Build profiles: debug vs release

Cargo has named profiles. The two most common are:

- **dev** (debug): fast builds, debug info, fewer optimizations.
- **release**: optimized builds suitable for benchmarking and deployment.

In practice:

- Use **debug** for day-to-day development and correctness work.
- Use **release** for performance measurement and production artifacts.

```
# Debug build (fast compile, less optimized)
cargo build

# Release build (optimized)
cargo build --release
```

2.4.2 Benchmarking discipline: always measure in release mode

Many performance discussions become misleading when measured in debug mode. When you measure performance:

- use `–release`,
- ensure you run enough iterations to warm up caches,
- compare changes with consistent input sizes,
- record results before and after changes.

2.4.3 Build time as a metric

Build time matters in large projects. Practical habits:

- keep dependencies minimal,
- separate stable core libraries from fast-changing UI/CLI layers,
- use workspaces to avoid unnecessary rebuilds,
- prefer incremental development workflows.

A simple way to get a feeling for costs is to compare clean builds:

```
# Clean all build artifacts
cargo clean

# Build debug
cargo build

# Build release
cargo build --release
```

2.4.4 Lock files: Cargo.lock is your reproducibility anchor

`Cargo.lock` records the exact versions selected by the resolver. For applications and binaries, committing `Cargo.lock` is standard practice to ensure:

- consistent builds across developers,
- reproducible CI,
- stable production behavior over time.

For libraries, policies may differ; however, for a book project and for most applications, treat `Cargo.lock` as a required artifact under version control.

2.5 Lab: building a small CLI project with tests

This lab builds a small, production-style command-line tool while exercising the complete workflow:

- project creation,
- argument parsing without external dependencies,
- structured functions suitable for testing,
- unit tests and integration-style tests,
- formatting and lint discipline.

The CLI will implement a single command:

- `sum <a> <b>` prints the sum of two signed integers.

2.5.1 Step 1: Create the project

```
cargo new rust_math_cli
cd rust_math_cli
```

2.5.2 Step 2: Implement the CLI with a testable core

Edit `src/main.rs` as follows. Notice the design:

- Parsing is separated from computation.
- A small pure function is easy to test.
- The program returns exit codes via `std::process::ExitCode`.

```rust
use std::process::ExitCode;

fn parse_i64(s: &str) -> Result<i64, String> {
    s.parse::<i64>()
        .map_err(|_| format!("invalid integer: {}", s))
}

fn sum(a: i64, b: i64) -> i64 {
    a + b
}

fn usage() -> String {
    let name = env!("CARGO_PKG_NAME");
    format!(
        "Usage:\n  {0} sum <a> <b>\n\nExamples:\n  {0} sum 10 32\n",
        name
    )
}
```

```
fn run(args: &[String]) -> Result<String, String> {
    if args.len() < 2 {
        return Err(usage());
    }

    match args[1].as_str() {
        "sum" => {
            if args.len() != 4 {
                return Err(usage());
            }
            let a = parse_i64(&args[2])?;
            let b = parse_i64(&args[3])?;
            Ok(format!("{}", sum(a, b)))
        }
        "--help" | "-h" => Err(usage()),
        _ => Err(usage()),
    }
}

fn main() -> ExitCode {
    let args: Vec<String> = std::env::args().collect();
    match run(&args) {
        Ok(out) => {
            println!("{}", out);
            ExitCode::SUCCESS
        }
        Err(e) => {
            eprintln!("{}", e);
            ExitCode::FAILURE
        }
    }
}
```

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

    #[test]
    fn test_sum() {
        assert_eq!(sum(10, 20), 30);
        assert_eq!(sum(-10, 5), -5);
    }

    #[test]
    fn test_parse_i64_ok() {
        assert_eq!(parse_i64("42").unwrap(), 42);
        assert_eq!(parse_i64("-7").unwrap(), -7);
    }

    #[test]
    fn test_parse_i64_err() {
        assert!(parse_i64("x").is_err());
        assert!(parse_i64("1.5").is_err());
    }

    #[test]
    fn test_run_sum_ok() {
        let args = vec![
            "rust_math_cli".to_string(),
            "sum".to_string(),
            "10".to_string(),
            "32".to_string(),
        ];
        assert_eq!(run(&args).unwrap(), "42");
    }
    #[test]
    fn test_run_usage_on_missing_args() {
        let args = vec!["rust_math_cli".to_string()];
        assert!(run(&args).is_err());
    }

    #[test]
    fn test_run_usage_on_unknown_cmd() {
        let args = vec![
            "rust_math_cli".to_string(),
            "mul".to_string(),
            "2".to_string(),
            "3".to_string(),
        ];
        assert!(run(&args).is_err());
    }
}
```

2.5.3 Step 3: Run, test, format, lint

```
# Run the CLI
cargo run -- sum 10 32

# Run tests
cargo test

# Format
cargo fmt

# Lint
cargo clippy -- -D warnings
```

2.5.4 Step 4: Release build and basic performance sanity

```
# Optimized build
cargo build --release

# Run the release binary
./target/release/rust_math_cli sum 100 200
```

2.5.5 Lab extensions (recommended)

To make this lab closer to real production practice, extend it as follows:

- Add a `sum-many` command: `sum-many <n1> <n2> ....`
- Add proper error variants using a small `enum` rather than `String`.
- Add an integration test under `tests/` that calls `run()` with multiple argument patterns.
- Add a `–version` flag that prints `CARGO_PKG_VERSION`.

By the end of this chapter, you should have a stable workflow:

- manage toolchains and targets with `rustup`,
- build and run consistently with `cargo`,
- enforce formatting with `rustfmt`,
- enforce idioms and warnings with `clippy`,
- measure performance in release builds,
- and keep builds reproducible through disciplined dependency and lock-file practices.

Core Language Constructs

This chapter introduces Rust’s core language constructs with a systems-oriented perspective. The goal is not to list syntax, but to build correct instincts: immutability-first design, explicit conversions, expression-based control flow, and structured pattern matching. These foundations matter because they directly support Rust’s safety and performance model.

3.1 Variables, immutability, shadowing

3.1.1 Bindings, not “mutable variables” by default

In Rust, a variable binding introduced with `let` is immutable by default. This is not a cosmetic rule; it is a design constraint that reduces accidental state changes, encourages functional-style reasoning, and makes aliasing and concurrency safer.

```rust
fn main() {
    let x = 10;      // immutable binding
    // x = 11;       // error: cannot assign twice to immutable variable
    let mut y = 10;  // mutable binding
    y = 11;
    println!("{}, {}", x, y);
}
```

3.1.2 Immutability is a performance and correctness hint

When data is immutable:

- you can freely share it as `&T` without synchronization,
- you reduce the mental burden of “who changed this?”,
- you allow the compiler to reason more aggressively about optimizations,
- and you prevent many classes of accidental bugs.

3.1.3 Shadowing: reuse names while changing meaning safely

Rust supports **shadowing**: you can redeclare a name with a new binding. This is a powerful tool for staged transformations (parse, validate, convert) without keeping multiple names or allowing mutation to blur intent.

```rust
fn main() {
    let s = "  42  ";
    let s = s.trim();           // shadow: now a &str with whitespace removed
    let n: i32 = s.parse().unwrap();
    let n = n * 2;              // shadow: now the computed value
    println!("{}", n);
}
```

Shadowing is especially useful when:

- you want to keep bindings immutable at each stage,
- you want to narrow a type (e.g., `&str` to `i32`),
- you want to emphasize transformation pipelines.

3.1.4 Constants and statics

Rust distinguishes compile-time constants from runtime bindings.

```rust
const MAX_RETRIES: u32 = 5;

fn main() {
    println!("{}", MAX_RETRIES);
}
```

Use:

- `const` for compile-time values with no fixed address requirement,
- `static` when a single memory location with a stable address is required (often used carefully, sometimes with synchronization).

3.2 Types, inference, conversions

3.2.1 Type inference: strong, but not magical

Rust uses type inference extensively, but it is still a statically typed language. The compiler infers types from context, and you can make types explicit whenever clarity is needed.

```rust
fn main() {
    let a = 10;          // inferred as i32 by default
    let b: i64 = 10;     // explicit type
    let c = 10_i128;     // suffix notation
    println!("{}, {}, {}", a, b, c);
}
```

3.2.2 Integer and float defaults

When Rust sees an integer literal with no context, it defaults to `i32`. For floats, the default is `f64`. This matters in APIs and generics, where explicit types can prevent confusion.

3.2.3 No implicit numeric narrowing

Rust does not perform implicit narrowing conversions. This is deliberate: implicit numeric conversions are a common source of bugs in systems code.

```rust
fn main() {
    let x: i32 = 1_000;
    let y: i64 = x as i64;   // explicit conversion required
    println!("{}", y);
}
```

3.2.4 Safe conversions: `try_from` and checked methods

When conversions can fail (overflow, out-of-range), prefer checked conversions rather than forced casts.

```rust
use std::convert::TryFrom;

fn main() {
    let big: i64 = 1_000;
    let small = i32::try_from(big).unwrap();
    println!("{}", small);

    let too_big: i64 = i64::from(i32::MAX) + 1;
    let r = i32::try_from(too_big);
    assert!(r.is_err());
}
```

For arithmetic, Rust provides checked, saturating, wrapping patterns explicitly on integer types:

```rust
fn main() {
    let x: u8 = 250;
    assert_eq!(x.saturating_add(10), 255);
    assert_eq!(x.wrapping_add(10), 4);
    assert_eq!(x.checked_add(10), None);
}
```

3.2.5 Strings: `String` vs `&str`

This is one of the most important type distinctions in Rust:

- `String` is an owned, growable UTF-8 buffer (heap allocated).
- `&str` is a borrowed string slice (a view into existing UTF-8 data).

```rust
fn takes_str(s: &str) {
    println!("{}", s);
}

fn main() {
    let owned = String::from("hello");
    let slice: &str = &owned;     // borrow as &str
    takes_str(slice);
    takes_str(&owned);
    takes_str("literal");
}
```

Prefer `&str` in function parameters when ownership is not required.

3.3 Functions, expressions, blocks

3.3.1 Functions: signatures are contracts

Rust function signatures specify:

- ownership vs borrowing,
- mutability intent,
- possible failures (commonly via `Result`),
- and how outputs relate to inputs (through lifetimes when needed).

```
fn add(a: i64, b: i64) -> i64 {
    a + b
}
```

3.3.2 Expressions, not statements

Most constructs in Rust are expressions: they produce values. This is a core part of Rust's style and supports concise, explicit logic.

```
fn main() {
    let x = 10;
    let y = if x > 0 { 1 } else { -1 }; // if is an expression
    println!("{}", y);
}
```

3.3.3 Blocks produce values

A block `{ ... }` can evaluate to a value. The last expression (without semicolon) is the block's value.

```
fn main() {
    let v = {
        let a = 10;
        let b = 20;
        a + b  // block value
    };
    println!("{}", v);
}
```

3.3.4 Early return and explicit control

Rust supports early returns and short-circuit logic. A key idiom for error flow uses `?` with `Result` (covered more deeply later), but you can already see the style:

```
fn parse_positive(s: &str) -> Result<i32, String> {
    let n: i32 = s.parse().map_err(|_| "not an integer".to_string())?;
    if n <= 0 {
        return Err("expected positive integer".to_string());
    }
    Ok(n)
}
```

3.4 Control flow: if/else, match, loops

3.4.1 if/else as an expression

Both branches must return compatible types when used as an expression.

```
fn signum(x: i32) -> i32 {
    if x > 0 { 1 } else if x < 0 { -1 } else { 0 }
}
```

3.4.2 match: exhaustive pattern matching

match enforces exhaustiveness. This often eliminates missing-case bugs and makes state transitions explicit.

```
fn classify(x: i32) -> &'static str {
    match x {
        0 => ”zero”,
        1..=9 => ”small positive”,
        -9..=-1 => ”small negative”,
        _ => ”large magnitude”,
    }
}
```

3.4.3 match with enums: structured state modeling

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

fn handle(m: Msg) -> String {
    match m {
        Msg::Quit => ”quit”.to_string(),
        Msg::Write(s) => format!(”write: {}”, s),
        Msg::Move { x, y } => format!(”move: {}, {}”, x, y),
    }
}
```

3.4.4 Loops: loop, while, for

Rust offers:

- `loop`: infinite loop with optional value return via `break expr`.
- `while`: condition-controlled loop.
- `for`: iterator-driven loop (the common choice).

```
fn main() {
    let mut i = 0;
    let sum = loop {
        i += 1;
        if i == 10 {
            break i * 2;  // break with a value
        }
    };
    println!("{}", sum);

    let mut n = 3;
    while n > 0 {
        n -= 1;
    }

    let mut total = 0;
    for x in [1, 2, 3, 4] {
        total += x;
    }
    println!("{}", total);
}
```

3.5 Lab: a mini arithmetic expression interpreter

This lab builds a small arithmetic expression interpreter that evaluates inputs like:

- `1 + 2 * 3`

- `(1 + 2) * 3`
- `-4 + 10 / 2`

The goal is not a full language, but a compact project that exercises:

- types and parsing,
- match and enums,
- functions returning `Result`,
- loops and structured control flow,
- and test-driven development.

3.5.1 Grammar (conceptual)

We will implement a standard precedence grammar:

- **Expr** := **Term** ((+|-) **Term**)*
- **Term** := **Factor** ((*|/) **Factor**)*
- **Factor** := **Number** | (**Expr**) | (+|-) **Factor**

3.5.2 Step 1: Tokenizer

Create a tokenizer that converts input text into tokens.

```
#[derive(Debug, Clone, PartialEq)]
enum Token {
    Num(i64),
    Plus,
    Minus,
```

```rust
    Star,
    Slash,
    LParen,
    RParen,
    End,
}

fn tokenize(input: &str) -> Result<Vec<Token>, String> {
    let mut chars = input.chars().peekable();
    let mut out = Vec::new();

    while let Some(&c) = chars.peek() {
        match c {
            ' ' | '\t' | '\n' | '\r' => {
                chars.next();
            }
            '0'..='9' => {
                let mut n: i64 = 0;
                while let Some(&d) = chars.peek() {
                    if ('0'..='9').contains(&d) {
                        chars.next();
                        n = n * 10 + (d as i64 - '0' as i64);
                    } else {
                        break;
                    }
                }
                out.push(Token::Num(n));
            }
            '+' => { chars.next(); out.push(Token::Plus); }
            '-' => { chars.next(); out.push(Token::Minus); }
            '*' => { chars.next(); out.push(Token::Star); }
            '/' => { chars.next(); out.push(Token::Slash); }
            '(' => { chars.next(); out.push(Token::LParen); }
            ')' => { chars.next(); out.push(Token::RParen); }
            _ => return Err(format!(”unexpected character: {}”, c)),
        }
    }

    out.push(Token::End);
    Ok(out)
}
```

3.5.3 Step 2: Recursive-descent parser with precedence

We parse using a small state struct with an index.

```
struct Parser {
    toks: Vec<Token>,
    pos: usize,
}

impl Parser {
    fn new(toks: Vec<Token>) -> Self {
        Self { toks, pos: 0 }
    }

    fn peek(&self) -> &Token {
        &self.toks[self.pos]
    }

    fn bump(&mut self) -> Token {
        let t = self.toks[self.pos].clone();
        self.pos += 1;
        t
    }
    fn expect(&mut self, want: Token) -> Result<(), String> {
        let got = self.bump();
        if got == want {
            Ok(())
        } else {
            Err(format!("expected {:?}, got {:?}", want, got))
        }
    }

    fn parse_expr(&mut self) -> Result<i64, String> {
        let mut v = self.parse_term()?;
        loop {
            match self.peek() {
                Token::Plus => {
                    self.bump();
                    let rhs = self.parse_term()?;
                    v = v + rhs;
                }
                Token::Minus => {
                    self.bump();
                    let rhs = self.parse_term()?;
                    v = v - rhs;
                }
                _ => break,
            }
        }
        Ok(v)
    }

    fn parse_term(&mut self) -> Result<i64, String> {
        let mut v = self.parse_factor()?;
        loop {
            match self.peek() {
                Token::Star => {
                    self.bump();
                    let rhs = self.parse_factor()?;
                    v = v * rhs;
                }
                Token::Slash => {
                    self.bump();
                    let rhs = self.parse_factor()?;
                    if rhs == 0 {
                        return Err(”division by zero”.to_string());
                    }
                    v = v / rhs;
                }
                _ => break,
            }
        }
        Ok(v)
    }

    fn parse_factor(&mut self) -> Result<i64, String> {
        match self.peek() {
            Token::Num(_) => {
                if let Token::Num(n) = self.bump() { Ok(n) } else { unreachable!() }
            }
            Token::LParen => {
                self.bump();
                let v = self.parse_expr()?;
                self.expect(Token::RParen)?;
                Ok(v)
            }
            Token::Plus => {
                self.bump();
                self.parse_factor()
            }
            Token::Minus => {
                self.bump();
                let v = self.parse_factor()?;
                Ok(-v)
            }
            t => Err(format!("unexpected token: {:?}", t)),
        }
    }
}

fn eval(input: &str) -> Result<i64, String> {
    let toks = tokenize(input)?;
    let mut p = Parser::new(toks);
    let v = p.parse_expr()?;
    match p.peek() {
        Token::End => Ok(v),
        t => Err(format!("trailing tokens starting at {:?}", t)),
    }
}
```

3.5.4 Step 3: A tiny CLI wrapper

This small `main` reads one expression from the command line and evaluates it.

```
fn main() {
    let args: Vec<String> = std::env::args().collect();
    if args.len() != 2 {
        eprintln!("Usage: expr_eval \"EXPR\"");
        std::process::exit(1);
    }

    match eval(&args[1]) {
        Ok(v) => println!("{}", v),
        Err(e) => {
            eprintln!("error: {}", e);
            std::process::exit(1);
        }
    }
}
```

3.5.5 Step 4: Tests

Write tests that capture operator precedence, parentheses, unary operators, and error cases.

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

    #[test]
    fn precedence() {
        assert_eq!(eval("1+2*3").unwrap(), 7);
        assert_eq!(eval("10-2*3").unwrap(), 4);
    }

    #[test]
    fn parentheses() {
        assert_eq!(eval("(1+2)*3").unwrap(), 9);
        assert_eq!(eval("2*(3+4)").unwrap(), 14);
    }

    #[test]
    fn unary() {
        assert_eq!(eval("-4+10/2").unwrap(), 1);
        assert_eq!(eval("--5").unwrap(), 5);
        assert_eq!(eval("-(2+3)*2").unwrap(), -10);
    }

    #[test]
    fn spaces() {
        assert_eq!(eval(” 1 + 2 * 3 ”).unwrap(), 7);
        assert_eq!(eval(” ( 1 + 2 ) * 3 ”).unwrap(), 9);
    }

    #[test]
    fn errors() {
        assert!(eval(”1/0”).is_err());
        assert!(eval(”(1+2”).is_err());
        assert!(eval(”1+”).is_err());
        assert!(eval(”x”).is_err());
    }
}
```

3.5.6 Lab extensions (recommended)

To deepen the lab while staying small:

- Add exponentiation ˆ with higher precedence.
- Add variables and a simple environment map (e.g., `x=10` then evaluate `x*2`).
- Add better error messages with position tracking (store indices in tokens).
- Switch the evaluator to build an AST first, then evaluate (prepares for later chapters on design).

By completing this chapter and lab, you will have practiced Rust’s core style:

- immutability-first design,

- explicit conversions and error handling,
- expression-oriented control flow,
- exhaustive pattern matching,
- and testable program structure.

Part II

The Secret Weapon — Ownership & Borrowing

Ownership from First Principles

Ownership is the central idea that makes Rust both safe and fast without requiring a garbage collector. It is not a “feature” layered on top of the language; it is the default rule that governs memory, lifetimes, aliasing, and resource cleanup.
If you fully understand ownership, you will find that many later topics become straightforward:

- borrowing and lifetimes become a precise tool rather than a mystery,
- API design becomes clearer because ownership boundaries are explicit,
- performance becomes easier to reason about because data movement and allocation are visible.

In this chapter you will learn ownership from first principles through concrete behavior: moves, copies, scope-based destruction, and observable stack/heap patterns. You will also complete a lab that trains you to “see” lifetimes in small programs.

4.1 Move vs copy

Rust distinguishes between **moving** a value (transferring ownership) and **copying** a value (duplicating bits so both bindings remain valid). This distinction is fundamental because it encodes resource ownership directly in the type system.

4.1.1 What a move means

A move transfers ownership from one binding to another. After the move, the original binding is no longer usable.

```rust
fn main() {
    let a = String::from("hello");
    let b = a;                  // move: ownership transferred
    // println!("{}", a);   // error: use of moved value
    println!("{}", b);
}
```

Why does this happen? Because `String` owns a heap allocation. If Rust allowed both `a` and `b` to remain valid without copying, then both would believe they own the same allocation, which would lead to double-free or use-after-free hazards.

4.1.2 What a copy means

Some types are inexpensive and safe to duplicate by simply copying their bits. These types implement the `Copy` trait. Typical examples: integers, floats, `bool`, `char`, and small tuples composed only of `Copy` types.

```rust
fn main() {
    let x: i32 = 10;
    let y = x;                  // copy
    println!("{}, {}", x, y);
}
```

Here, `x` remains usable because a bitwise copy is safe and does not duplicate a resource that needs unique destruction.

4.1.3 How to know if a type is Copy

A type is Copy if:

- it has no custom destructor (Drop),
- it does not manage ownership of heap resources,
- and all of its fields are also Copy.

```
#[derive(Copy, Clone)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p1 = Point { x: 1, y: 2 };
    let p2 = p1; // copy
    println!("({}, {}), ({}, {})", p1.x, p1.y, p2.x, p2.y);
}
```

4.1.4 Copy vs clone

Copy is implicit and cheap (bitwise copy). Clone is explicit and may allocate or perform deeper work.

```
fn main() {
    let a = String::from("hello");
    let b = a.clone();      // explicit deep copy (duplicates heap buffer)
    println!("{}, {}", a, b);
}
```

A professional rule: treat clone() as a design decision. When you see clone(), ask:

- do I truly need ownership here?
- could I borrow instead?
- could I restructure scopes to avoid duplication?

4.1.5 Moves occur at function boundaries

Passing an owned value to a function typically moves it into that function.

```
fn consume(s: String) {
    println!("{}", s);
} // s dropped here

fn main() {
    let s = String::from("data");
    consume(s);             // move into consume
    // println!("{}", s);   // error: moved
}
```

Borrowing avoids moving:

```
fn view(s: &str) {
    println!("{}", s);
}

fn main() {
    let s = String::from("data");
    view(&s);               // borrow
    println!("{}", s);      // still valid
}
```

4.1.6 Moves are not necessarily expensive

A move is usually cheap: it moves the *ownership handle*, not the entire heap buffer. For `String`, the move is typically a copy of a few machine words (pointer, length, capacity). The heap allocation is not copied.

This is one of the key performance insights of Rust:

- ownership transfer can be cheap,
- deep copies are explicit via `clone()`,
- borrowing avoids moving ownership entirely.

4.2 Scope and drop

Rust uses scope-based resource management. When a value goes out of scope, Rust runs its destructor (`Drop`) automatically. This generalizes RAII beyond memory: files, sockets, locks, temporary directories, and any user-defined resource can be cleaned up deterministically.

4.2.1 Scopes are lifetime boundaries

A binding is valid from its declaration until the end of its scope, unless it is moved earlier.

```
fn main() {
    let outer = String::from("outer");

    {
        let inner = String::from("inner");
        println!("{}", inner);
    } // inner dropped here

    println!("{}", outer);
} // outer dropped here
```

4.2.2 Drop order: reverse of construction

Values are dropped in reverse order of creation within a scope.

```rust
struct Tracker(&'static str);

impl Drop for Tracker {
    fn drop(&mut self) {
        println!(”drop {}”, self.0);
    }
}

fn main() {
    let a = Tracker(”A”);
    let b = Tracker(”B”);
    let c = Tracker(”C”);
    println!(”end of scope”);
}
```

The printed drop order is:

- `drop C`
- `drop B`
- `drop A`

This predictable order is critical when you manage resources with dependencies (for example, a lock guard that must be released before a buffer is destroyed).

4.2.3 Explicitly ending a value’s lifetime

Sometimes you want to drop a value before the end of the scope (for example, to release a lock early). You can do this with `drop(value)` from the standard library.

```
fn main() {
    let s = String::from(”temporary”);
    drop(s);                // dropped here
    // println!("{}", s);   // error: use after move (s already dropped)
}
```

This is not about “manual memory management”; it is about explicit resource boundaries in a deterministic destruction model.

4.2.4 Moving changes drop responsibility

When a value is moved, the responsibility to drop it moves as well. The original binding becomes invalid because it no longer owns the resource.
This is why “use of moved value” errors exist: they protect the single-owner invariant that makes deterministic cleanup correct.

4.3 Stack and heap behavior in practice

Understanding stack and heap behavior in Rust is about understanding where *ownership handles* live versus where *owned data* may live.

4.3.1 A practical mental model

- The **stack** holds fixed-size values with known size at compile time (including pointers and small structs).

- The **heap** holds dynamically sized allocations requested at runtime (e.g., `String`, `Vec<T>` buffers).

- Owning types like `String` and `Vec<T>` store a small handle on the stack that points to heap-allocated buffers.

4.3.2 Example: moving a String does not copy its heap buffer

```
fn main() {
    let s1 = String::from("hello world");
    let s2 = s1;  // move: transfers ownership of the heap buffer handle
    println!("{}", s2);
}
```

No deep copy occurs. The heap allocation remains the same; only ownership is transferred.

4.3.3 Example: Vec reallocation and why borrows matter

A `Vec<T>` may reallocate its buffer when it grows. This is why Rust forbids certain patterns: a reference into the vector could become invalid after reallocation.

```
fn main() {
    let mut v = vec![1, 2, 3];

    let first = &v[0];
    // v.push(4); // error: cannot borrow `v` as mutable because it is also borrowed as
    ↪  immutable
    println!("{}", first);
}
```

The compiler prevents you from holding a reference that could be invalidated by mutation that may reallocate.

4.3.4 Example: Box for explicit heap allocation of a single value

`Box<T>` allocates a `T` on the heap and owns it.

```
fn main() {
    let b = Box::new(123_i32);
    println!("{}", *b);
}
```

This is useful when:

- you need a stable address on the heap,
- you want to control ownership explicitly,
- you are building recursive data structures.

4.3.5 Borrowing: references are non-owning views

References (&T, &mut T) do not own the data they point to. They are views with a scope-bound lifetime contract.

```
fn len_of(s: &String) -> usize {
    s.len()
}

fn main() {
    let s = String::from("abcd");
    let n = len_of(&s);     // borrow: s remains the owner
    println!("{}", n);
}
```

The owner (s) is responsible for destruction at the end of its scope.

4.3.6 Returning ownership vs returning a reference

A common early mistake is attempting to return a reference to a local value.

```
fn bad() -> &str {
    let s = String::from("temp");
    &s // error: cannot return reference to local variable
}
```

This is rejected because s will be dropped at function exit. If the caller needs the data, you must return ownership:

```
fn good() -> String {
    let s = String::from("temp");
    s
}
```

4.4 Lab: tracing data lifetimes in small programs

This lab is designed to train you to “see” ownership transfers, borrow scopes, and drop points. You will write and run small programs, then predict their behavior before compiling.

4.4.1 Lab rules

- Do not use clone() unless the exercise explicitly asks for it.
- For each program, answer: Who owns what? When is it dropped? Where do borrows start and end?
- Prefer fixes that improve ownership design (borrowing or restructuring) rather than forcing copies.

4.4.2 Exercise 1: predict move points

Task: predict which line causes a move, and why.

```
fn main() {
    let a = String::from(”A”);
    let b = a;
    let c = b;
    println!(”{}”, c);
    // println!(”{}”, a);
    // println!(”{}”, b);
}
```

Questions:

- Which binding owns the allocation at the end?
- Why are a and b invalid after the moves?

4.4.3 Exercise 2: copy types behave differently

Task: predict why this compiles, and why the previous one does not.

```
fn main() {
    let x: i32 = 7;
    let y = x;
    let z = y;
    println!(”{}, {}, {}”, x, y, z);
}
```

Questions:

- What trait makes this possible?
- Why is String not Copy?

4.4.4 Exercise 3: track drops using Drop

Task: run and observe the drop order.

```
struct Tracker(&'static str);

impl Drop for Tracker {
    fn drop(&mut self) {
        println!("drop {}", self.0);
    }
}

fn main() {
    let a = Tracker("A");
    {
        let b = Tracker("B");
        let c = Tracker("C");
        println!("inner end");
    }
    println!("outer end");
}
```

Questions:

- What is the exact output order?
- Why is the order reversed within each scope?

4.4.5 Exercise 4: borrow scope is a lifetime boundary

Task: make the program compile by shrinking the borrow scope without cloning the entire vector.

```rust
fn main() {
    let mut v = vec![10, 20, 30];

    let first = &v[0];
    v.push(40); // does not compile
    println!("{}", first);
}
```

Hints (choose one approach):

- Copy out the element if it is `Copy`.
- Move printing before mutation.
- Create a tighter scope for the borrow.

4.4.6 Exercise 5: return ownership vs borrow

Task: fix the design by choosing the correct return type.

```rust
fn make_message() -> &str {
    let s = String::from("hello");
    &s
}
```

Fix A (return ownership):

```rust
fn make_message() -> String {
    let s = String::from("hello");
    s
}
```

Fix B (borrow from input):

```rust
fn echo<'a>(s: &'a str) -> &'a str {
    s
}
```

Questions:

- When is Fix A appropriate?
- When is Fix B appropriate?

4.4.7 Exercise 6: observe move into and out of functions

Task: predict which calls move ownership, and which only borrow.

```rust
fn takes_owned(s: String) -> usize {
    s.len()
}

fn takes_borrow(s: &str) -> usize {
    s.len()
}

fn main() {
    let s = String::from("abcd");

    let a = takes_borrow(&s);
    println!("{}", a);

    let b = takes_owned(s);
    println!("{}", b);

    // println!("{}", s); // does not compile: moved
}
```

Questions:

- Why is `s` still valid after `takes_borrow`?
- Why is `s` invalid after `takes_owned`?

4.4.8 What you should gain from this lab

After this lab, you should be able to:

- predict whether a line performs a move or a copy,
- identify the owner responsible for dropping a resource,
- reason about borrow scopes as strict lifetime boundaries,
- choose between returning ownership and returning a borrow,
- understand why Rust prevents patterns that could become use-after-free or invalidation bugs.

Ownership is the foundation. In the next chapters, you will learn how borrowing and lifetimes extend this model to enable efficient, safe APIs without unnecessary allocations or copies.

Borrowing and References

Borrowing is the mechanism that turns Rust's ownership model from a strict discipline into a flexible, high-performance engineering tool. Ownership answers the question *"who is responsible for this data?"*. Borrowing answers the equally important question *"who may temporarily use this data, and under what conditions?"*.

This chapter builds borrowing from the ground up: first as a conceptual model of permissions and lifetimes, and only later as syntax. If you internalize borrowing correctly, you will naturally write APIs that are safe, fast, and free of unnecessary allocations or cloning.

5.1 &T vs &mut T

5.1.1 Borrowing as permission, not as an address

In Rust, a reference is not merely a pointer. It is a **temporary permission** granted by the owner of a value.

There are two kinds of references:

- `&T`: a shared (immutable) borrow.
- `&mut T`: an exclusive (mutable) borrow.

The difference is not about syntax; it is about guarantees.

5.1.2 &T: shared read-only access

A shared reference allows reading but forbids mutation. Multiple shared references may coexist because they cannot change the underlying value.

```
fn print_len(s: &String) {
    println!("{}", s.len());
}

fn main() {
    let s = String::from("hello");
    let r1 = &s;
    let r2 = &s;
    print_len(r1);
    print_len(r2);
}
```

Key properties of `&T`:

- many readers are allowed,
- no mutation through shared references,
- safe to share across threads if `T` is thread-safe.

5.1.3 &mut T: exclusive mutable access

A mutable reference grants exclusive access. While it exists, no other references (mutable or shared) to the same value may exist.

```
fn increment(x: &mut i32) {
    *x += 1;
}
```

```
fn main() {
    let mut n = 10;
    increment(&mut n);
    println!("{}", n);
}
```

Key properties of `&mut T`:

- exactly one mutable reference at a time,
- no other references may coexist,
- mutation is explicit and localized.

This exclusivity is what prevents data races, iterator invalidation, and many subtle aliasing bugs.

5.1.4 Why Rust separates &T and &mut T

In C and C++, a pointer may be used to read or write, and aliasing rules are often informal or compiler-dependent. Rust makes mutability explicit so the compiler can enforce and rely on strong guarantees:

- if there is a mutable reference, it is the only access path,
- if there are multiple references, they are read-only.

This clarity enables both safety and aggressive optimization.

5.2 The three golden borrowing rules

Rust's borrowing rules can be summarized in three fundamental laws. Everything else follows from them.

5.2.1 Rule 1: At any time, you have either

- any number of shared references (&T), **or**
- exactly one mutable reference (&mut T),

but never both at the same time.

```rust
fn main() {
    let mut v = vec![1, 2, 3];

    let a = &v[0];
    // let b = &mut v[1]; // error: cannot borrow as mutable while shared borrow exists
    println!("{}", a);
}
```

5.2.2 Rule 2: References must always be valid

A reference must never outlive the value it refers to.

```rust
fn bad_ref() -> &i32 {
    let x = 10;
    &x // error: reference would outlive `x`
}
```

This rule eliminates use-after-free bugs in safe Rust.

5.2.3 Rule 3: Mutable references imply exclusivity

If you can mutate through a reference, no other code may observe or mutate that value at the same time.

```rust
fn main() {
    let mut x = 5;
    let r = &mut x;
    *r += 1;
    // let s = &x; // error: cannot borrow while mutable borrow exists
    println!("{}", r);
}
```

This rule is the foundation of Rust’s data-race freedom in safe code.

5.2.4 Borrow scopes matter

A borrow is active only for the duration of its scope. Often, fixing a borrow error is about reducing the scope, not changing ownership.

```rust
fn main() {
    let mut v = vec![1, 2, 3];

    {
        let first = &v[0];
        println!("{}", first);
    } // borrow ends here

    v.push(4); // now allowed
}
```

Understanding borrow scopes is essential for writing ergonomic code.

5.3 Lifetimes: concept before syntax

5.3.1 What a lifetime really is

A lifetime is not a counter, not a runtime value, and not a region of memory. A lifetime is a **compile-time description of how long a reference is guaranteed to be valid**.
The compiler already reasons about lifetimes even when you do not write them explicitly.

5.3.2 Implicit lifetimes

Most simple cases require no explicit lifetime annotations.

```rust
fn len(s: &str) -> usize {
    s.len()
}
```

Here, the compiler infers that the returned value does not borrow from `s`, so no lifetime annotation is required.

5.3.3 When lifetimes become visible

Lifetimes matter when references are returned or stored in ways that tie input and output together.

```rust
fn first<'a>(s: &'a str) -> &'a str {
    s.split_whitespace().next().unwrap_or("")
}
```

Conceptually, this means:

- the returned reference is valid for at most as long as `s` is valid,
- the function does not create new data; it only returns a view into existing data.

5.3.4 Lifetime annotations describe relationships

Lifetimes do not extend lifetimes. They only describe how lifetimes relate.

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

This states: the returned reference is valid as long as both inputs are valid.

5.3.5 Common lifetime mistake: returning references to temporaries

```
fn bad() -> &str {
    let s = String::from("temp");
    &s
}
```

This fails because the data does not live long enough. The correct fix is to return ownership when needed.

```
fn good() -> String {
    String::from("temp")
}
```

The guiding rule: **borrow when the caller owns the data; return ownership when the function creates the data**.

5.4 Lab: designing a clean API without unnecessary cloning

This lab trains you to design APIs that respect ownership and borrowing, avoiding performance-killing `clone()` calls unless they are truly required.

5.4.1 Lab rules

- Do not use `clone()` unless explicitly justified.
- Prefer borrowing over ownership for read-only access.
- Make ownership transfer explicit at API boundaries.

5.4.2 Exercise 1: removing unnecessary ownership

Initial version:

```
fn print_upper(s: String) {
    println!("{}", s.to_uppercase());
}

fn main() {
    let name = String::from("rust");
    print_upper(name);
    // println!("{}", name); // cannot use
}
```

Task: redesign the API so the caller retains ownership.

```
fn print_upper(s: &str) {
    println!("{}", s.to_uppercase());
}
```

5.4.3 Exercise 2: avoiding clone in data processing

Initial version:

```
fn count_chars(s: String) -> usize {
    s.chars().count()
}
```

Task: make the function usable without transferring ownership.

```
fn count_chars(s: &str) -> usize {
    s.chars().count()
}
```

5.4.4 Exercise 3: designing a reusable parser API

Initial version:

```
fn parse_number(s: String) -> Result<i64, String> {
    s.parse::<i64>().map_err(|_| "invalid".to_string())
}
```

Task: redesign to borrow input while returning owned results.

```
fn parse_number(s: &str) -> Result<i64, String> {
    s.parse::<i64>().map_err(|_| "invalid".to_string())
}
```

5.4.5 Exercise 4: returning borrowed data correctly

Design a function that returns a substring of its input.

```
fn first_word<'a>(s: &'a str) -> &'a str {
    s.split_whitespace().next().unwrap_or("")
}
```

Questions:

- Why is returning `&str` correct here?
- Why would returning `String` be less efficient?

5.4.6 Exercise 5: combining mutable and shared access safely

Initial version (does not compile):

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

Task: fix the design without cloning the vector.

One valid solution:

```
fn main() {
    let mut data = vec![1, 2, 3];

    let first = data[0];
    data.push(4);
    println!("{}", first);
}
```

5.4.7 What you should gain from this lab

After completing this lab, you should be able to:

- choose `&T` vs `&mut T` intentionally,
- recognize and apply the three borrowing rules naturally,
- reason about lifetimes without relying on syntax first,
- design APIs that borrow inputs instead of cloning,
- write Rust code that is both safe and efficient by construction.

Borrowing is what allows Rust to scale: from tiny functions to large systems, from single-threaded logic to concurrent architectures, without sacrificing correctness or performance.

Lifetimes Without Fear

Lifetimes are often perceived as the most intimidating part of Rust. In reality, lifetimes are not a new concept invented by Rust; they are a formalization of rules that systems programmers already rely on mentally when reasoning about pointers, references, and ownership. Rust simply makes those rules explicit, checkable, and enforceable at compile time.

This chapter demystifies lifetimes by focusing first on **why they exist**, then on how the compiler already reasons about them for you, and finally on how to express lifetime relationships clearly in structs, implementations, and traits. The chapter concludes with a practical lab that builds a small parser using only safe references.

Why lifetimes exist

6.0.1 The fundamental problem lifetimes solve

The core problem lifetimes address is simple:

> How long is a reference guaranteed to point to valid data?

In languages like C and C++, this question is answered informally by programmer discipline, comments, and conventions. The compiler generally does not verify that a pointer or reference remains valid for its entire use.

Rust makes this guarantee explicit:

- every reference has a scope during which it is valid,
- references must never outlive the data they point to,
- these relationships are verified statically.

Lifetimes are the mechanism Rust uses to describe and enforce these guarantees.

6.0.2 Lifetimes are about relationships, not durations

A common misconception is that lifetimes measure time. They do not. Lifetimes describe **relationships between references and the data they borrow**.

Consider this example:

```rust
fn len(s: &str) -> usize {
    s.len()
}
```

The compiler understands that:

- `s` is a reference,
- `len` does not return a reference tied to `s`,
- therefore, no explicit lifetime annotation is needed.

Now compare with a function that returns a reference:

```rust
fn first_char<'a>(s: &'a str) -> &'a str {
    &s[0..1]
}
```

Here, the lifetime annotation does not extend the lifetime of `s`. It states a relationship:

- the returned reference is valid for at most as long as `s` is valid.

6.0.3 Why lifetimes cannot be inferred in all cases

The compiler can infer lifetimes in many simple cases, but ambiguity arises when:

- multiple references are involved,
- references may be returned,
- or references are stored in data structures.

Example of ambiguity:

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

The compiler cannot decide whether the returned reference is tied to `a` or `b`. Explicit lifetime annotations resolve this by expressing intent:

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

6.0.4 What lifetimes prevent

By enforcing lifetime relationships, Rust prevents:

- use-after-free bugs,
- dangling references,
- references to stack data that has gone out of scope,
- iterator invalidation through mutation.

These are among the most costly and dangerous bugs in systems software.

6.1 Lifetime elision rules

6.1.1 Why elision exists

Explicit lifetime annotations everywhere would make Rust verbose and unpleasant to use. To avoid this, Rust applies a set of **lifetime elision rules** that allow the compiler to infer lifetimes in common patterns.

These rules do not change the semantics of the program; they merely remove the need to write obvious annotations.

6.1.2 The three elision rules

Rust applies the following rules in function signatures:

Rule 1: Each input reference gets its own lifetime parameter.

```
fn f(x: &i32, y: &i32)
```

is treated as:

```
fn f<'a, 'b>(x: &'a i32, y: &'b i32)
```

Rule 2: If there is exactly one input lifetime, it is assigned to all output references.

```
fn first(s: &str) -> &str
```

is treated as:

```
fn first<'a>(s: &'a str) -> &'a str
```

Rule 3: If there are multiple input lifetimes, but one of them is `&self` or `&mut self`, the lifetime of `self` is assigned to output references.

```
impl Buffer {
    fn data(&self) -> &str {
        &self.data
    }
}
```

This rule makes method signatures much cleaner.

6.1.3 When elision fails

Elision fails when the compiler cannot determine a single, unambiguous lifetime relationship.

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

In such cases, explicit annotations are required to communicate intent.

6.2 Lifetimes in structs, impls, and traits

6.2.1 Structs holding references

When a struct contains references, the struct must declare the lifetime of those references.

```
struct View<'a> {
    data: &'a str,
}
```

This states:

- an instance of `View<'a>` cannot outlive the data it references.

6.2.2 Constructing and using reference-holding structs

```
fn main() {
    let s = String::from(”hello world”);
    let v = View { data: &s };
    println!(”{}”, v.data);
}
```

The compiler ensures that `v` cannot outlive `s`.

6.2.3 Lifetimes in impl blocks

When implementing methods for a struct with lifetimes, the lifetime parameter must appear on the `impl`.

```
impl<'a> View<'a> {
    fn as_str(&self) -> &'a str {
        self.data
    }
}
```

This expresses that:

- methods may return references tied to the struct’s lifetime.

6.2.4 Multiple lifetimes in structs

Some structures reference multiple independent data sources.

```
struct Pair<'a, 'b> {
    left: &'a str,
    right: &'b str,
}
```

This allows each reference to have its own lifetime relationship.

6.2.5 Lifetimes in traits

Traits that expose borrowed data must also express lifetime relationships.

```rust
trait Source<'a> {
    fn get(&self) -> &'a str;
}
```

This trait states:

- implementors promise that returned references live at least as long as `’a`.

Implementing the trait:

```rust
struct Text<'a> {
    s: &'a str,
}

impl<'a> Source<'a> for Text<'a> {
    fn get(&self) -> &'a str {
        self.s
    }
}
```

6.2.6 Owned vs borrowed data in structs

A key design decision is whether a struct:

- owns its data (e.g., `String`),
- or borrows its data (e.g., `&str`).

Owned data simplifies lifetimes but may allocate. Borrowed data avoids allocation but requires lifetime parameters. Choosing between them is a core API design decision.

6.3 Lab: building a small parser using safe references

This lab builds a small, zero-allocation parser that borrows from its input string. The goal is to practice expressing lifetime relationships naturally, without cloning or allocating intermediate strings.

6.3.1 Problem statement

Implement a parser that reads a comma-separated list of identifiers and returns them as borrowed string slices.
Input example:

- `"alpha,beta,gamma"`

Output:

- `["alpha", "beta", "gamma"]`

All returned slices must borrow from the original input.

6.3.2 Design

We will:

- store a reference to the input,
- track the current parsing position,
- return slices with lifetimes tied to the input.

6.3.3 Parser structure

```
struct Parser<'a> {
    input: &'a str,
    pos: usize,
}
```

6.3.4 Implementation

```
impl<'a> Parser<'a> {
    fn new(input: &'a str) -> Self {
        Self { input, pos: 0 }
    }

    fn next_ident(&mut self) -> Option<&'a str> {
        if self.pos >= self.input.len() {
            return None;
        }

        let bytes = self.input.as_bytes();
        let start = self.pos;

        while self.pos < bytes.len() && bytes[self.pos] != b',' {
            self.pos += 1;
        }

        let end = self.pos;

        if self.pos < bytes.len() && bytes[self.pos] == b',' {
            self.pos += 1;
        }

        Some(&self.input[start..end])
    }
}
```

6.3.5 Using the parser

```
fn main() {
    let text = String::from(”alpha,beta,gamma”);
    let mut p = Parser::new(&text);

    while let Some(id) = p.next_ident() {
        println!(”{}”, id);
    }
}
```

6.3.6 Key observations

- No allocation occurs during parsing.
- All returned slices are valid as long as the input string lives.
- The lifetime `’a` connects the parser, its methods, and their outputs.

6.3.7 Lab extensions

To deepen understanding:

- Add whitespace trimming without allocation.
- Add validation rules (e.g., alphabetic only).
- Return an iterator instead of a method.
- Store parsed results in a struct that borrows from the input.

6.3.8 What you should gain from this lab

After completing this lab, you should be able to:

- explain why lifetimes exist and what problem they solve,
- read and write lifetime annotations without fear,
- design structs and traits that borrow data safely,
- build efficient, zero-allocation parsers using references,
- reason about lifetime relationships before reaching for owned data.

Lifetimes are not an obstacle; they are a language for expressing correctness. Once understood, they become one of Rust's greatest strengths.

Smart Pointers

Rust's ownership model already gives you deterministic resource cleanup and strong aliasing guarantees. Smart pointers extend that model to cover the real needs of systems and application design:

- explicit heap allocation and stable addresses,
- shared ownership within a single thread or across threads,
- interior mutability for carefully controlled mutation behind shared references,
- breaking reference cycles safely.

In Rust, "smart pointer" is not only about automatic memory management. It is about encoding ownership strategy and access permissions into types so that unsafe patterns become either impossible or explicit.

7.1 Box, Rc, Arc

7.1.1 `Box<T>`: single ownership on the heap

`Box<T>` allocates a value of type `T` on the heap and provides unique ownership. It is the simplest smart pointer: it behaves like owning `T`, but with the data placed on the heap.
Common reasons to use `Box<T>`:

- the type is large and you want to reduce stack usage,
- you need a stable address on the heap,
- you want to build recursive data structures (where size is not known without indirection),
- you want explicit heap ownership without shared ownership.

Example: heap allocation

```
fn main() {
    let b: Box<i32> = Box::new(123);
    println!("{}", *b);
}
```

Example: recursive types require indirection

A recursive enum cannot contain itself directly (infinite size). `Box` solves this by providing indirection.

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

fn main() {
    let xs = List::Cons(1, Box::new(List::Cons(2, Box::new(List::Nil))));
}
```

7.1.2 `Rc<T>`: shared ownership (single-threaded)

`Rc<T>` provides reference-counted shared ownership. Multiple owners can hold `Rc<T>` clones, and the value is dropped when the last owner goes away.

Important properties:

- Rc<T> is cheap to clone (increments a counter),
- it enables shared ownership in data structures (graphs, DAGs, trees with shared subtrees),
- it is not thread-safe, and therefore not usable across threads.

Example: sharing a value

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

fn main() {
    let shared = Rc::new(String::from("shared"));

    let a = Rc::clone(&shared);
    let b = Rc::clone(&shared);

    println!("count = {}", Rc::strong_count(&shared));
    println!("{}, {}", a, b);
}
```

7.1.3 Arc<T>: shared ownership across threads

Arc<T> is the thread-safe version of reference counting. It uses atomic operations to update its count, allowing safe cloning and sharing across threads.

Key properties:

- Arc<T> is for sharing read-mostly or synchronized data across threads,
- cloning an Arc is still relatively cheap, but more expensive than Rc due to atomics,
- mutation through Arc<T> typically requires additional synchronization (e.g., Mutex, RwLock, or atomics).

Example: sharing read-only data across threads

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

fn main() {
    let msg = Arc::new(String::from("hello"));

    let mut handles = Vec::new();
    for _ in 0..4 {
        let m = Arc::clone(&msg);
        handles.push(thread::spawn(move || {
            println!("{}", m);
        }));
    }

    for h in handles {
        h.join().unwrap();
    }
}
```

Example: shared mutable state using Arc + Mutex

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

fn main() {
    let counter = Arc::new(Mutex::new(0_i32));

    let mut handles = Vec::new();
    for _ in 0..4 {
        let c = Arc::clone(&counter);
        handles.push(thread::spawn(move || {
            let mut g = c.lock().unwrap();
            *g += 1;
        }));
    }

    for h in handles {
        h.join().unwrap();
    }

    println!("{}", *counter.lock().unwrap());
}
```

7.1.4 Choosing between Box, Rc, Arc

A practical selection rule:

- **Box**: single owner, heap allocation, recursive structures.
- **Rc**: shared ownership in one thread, graphs/trees/DAGs.
- **Arc**: shared ownership across threads, shared configuration/state with synchronization as needed.

7.2 RefCell and interior mutability

7.2.1 The idea: mutate through &T (with runtime checks)

Normally, Rust requires `&mut T` for mutation. Interior mutability types allow mutation even when you only have a shared reference `&T`. This is possible because the type itself enforces the borrowing rules at runtime.

The most common interior mutability primitive is `RefCell<T>`:

- it allows borrowing immutably via `borrow()`,

- borrowing mutably via `borrow_mut()`,
- and it panics at runtime if you violate borrowing rules.

7.2.2 Why interior mutability exists

Interior mutability enables patterns that are safe but hard to express with strict compile-time borrowing:

- caching/memoization,
- graph structures with shared nodes,
- self-referential-like designs where mutation is local and controlled,
- incremental computation where mutation is hidden behind a stable API.

Example: shared graph node with mutable payload (single-threaded)

A very common combination is `Rc<RefCell<T»`:

- `Rc` enables shared ownership,
- `RefCell` enables mutation inside shared ownership.

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

#[derive(Debug)]
struct Node {
    value: i32,
    next: Option<Rc<RefCell<Node>>>,
}
```

```
fn main() {
    let a = Rc::new(RefCell::new(Node { value: 1, next: None }));
    let b = Rc::new(RefCell::new(Node { value: 2, next: None }));

    a.borrow_mut().next = Some(Rc::clone(&b));

    b.borrow_mut().value += 10;

    println!("{:?}", a.borrow());
    println!("{:?}", b.borrow());
}
```

7.2.3 RefCell borrowing is still borrowing

`RefCell` does not remove borrowing rules; it moves enforcement from compile-time to runtime.

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

fn main() {
    let x = RefCell::new(10);

    let _a = x.borrow();        // shared borrow
    // let _b = x.borrow_mut(); // panic if executed: cannot mutably borrow while immutably
    ↪  borrowed
}
```

Use `RefCell` when:

- the borrowing structure is truly correct,
- but the compiler cannot prove it easily,
- and runtime checks are acceptable.

7.2.4 Threaded interior mutability: Mutex and RwLock

For cross-thread mutation, use synchronization primitives that provide interior mutability safely:

- `Mutex<T>` for exclusive access,
- `RwLock<T>` for many-readers/one-writer patterns.

These types enforce borrowing at runtime through locking, not through the borrow checker.

7.3 Weak references and cycle prevention

7.3.1 The reference cycle problem

Reference counting cannot reclaim cycles. If two `Rc` values point to each other, their strong counts never reach zero, and memory leaks.
This is not undefined behavior; it is a logical leak.

7.3.2 Weak references solve cycles

A `Weak<T>` reference does not contribute to the strong count. It allows pointing to data without owning it.
A common design:

- strong references (`Rc`) represent ownership,
- weak references (`Weak`) represent back-references or non-owning links.

Example: parent pointers with Weak

A classic pattern: children own their parent strongly? That creates cycles. Instead:

- parent owns children via Rc,
- child points to parent via Weak.

```
use std::cell::RefCell;
use std::rc::{Rc, Weak};

#[derive(Debug)]
struct Node {
    name: String,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}

fn main() {
    let root = Rc::new(Node {
        name: "root".to_string(),
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(Vec::new()),
    });

    let child = Rc::new(Node {
        name: "child".to_string(),
        parent: RefCell::new(Rc::downgrade(&root)),
        children: RefCell::new(Vec::new()),
    });

    root.children.borrow_mut().push(Rc::clone(&child));

    println!("root strong = {}, weak = {}", Rc::strong_count(&root),
    ↪   Rc::weak_count(&root));
    println!("child strong = {}, weak = {}", Rc::strong_count(&child),
    ↪   Rc::weak_count(&child));
    if let Some(p) = child.parent.borrow().upgrade() {
        println!("child parent = {}", p.name);
    }
}
```

7.3.3 Upgrading Weak

To use a `Weak<T>`, you attempt to `upgrade()` it to an `Rc<T>`. This returns `Option<Rc<T»` because the value may have been dropped.
This forces correct handling of dangling back-references.

7.3.4 Arc cycles and Weak

The same concept exists in multithreaded form:

- `Arc<T>` with `Weak<T>`,

to avoid leaks in cyclic graphs shared across threads.

7.4 Lab: building a safe AST tree

This lab builds a safe Abstract Syntax Tree (AST) with:

- recursive structure (requires `Box`),
- optional shared subtrees (upgrade to `Rc` for DAGs),
- a deliberate design that avoids cycles by construction.

7.4.1 Goal

We will implement an expression AST for arithmetic:

- integers,
- unary minus,
- binary operators: `+`, `-`, `*`, `/`.

7.4.2 Step 1: define operators

```
#[derive(Debug, Copy, Clone, PartialEq, Eq)]
enum BinOp {
    Add,
    Sub,
    Mul,
    Div,
}
```

7.4.3 Step 2: define the AST using Box

```
#[derive(Debug, Clone, PartialEq)]
enum Expr {
    Int(i64),
    Neg(Box<Expr>),
    Bin {
        op: BinOp,
        left: Box<Expr>,
        right: Box<Expr>,
    },
}
```

7.4.4 Step 3: implement evaluation

```
impl Expr {
    fn eval(&self) -> Result<i64, String> {
        match self {
            Expr::Int(n) => Ok(*n),
            Expr::Neg(e) => Ok(-e.eval()?),
            Expr::Bin { op, left, right } => {
                let a = left.eval()?;
                let b = right.eval()?;
                match op {
                    BinOp::Add => Ok(a + b),
                    BinOp::Sub => Ok(a - b),
                    BinOp::Mul => Ok(a * b),
                    BinOp::Div => {
                        if b == 0 {
                            Err("division by zero".to_string())
                        } else {
                            Ok(a / b)
                        }
                    }
                }
            }
        }
    }
}
```

7.4.5 Step 4: building trees safely

```
fn main() -> Result<(), String> {
    let ast = Expr::Bin {
        op: BinOp::Add,
        left: Box::new(Expr::Int(1)),
        right: Box::new(Expr::Bin {
            op: BinOp::Mul,
            left: Box::new(Expr::Int(2)),
            right: Box::new(Expr::Int(3)),
        }),
    };

    let v = ast.eval()?;
    println!("{}", v); // 7
    Ok(())
}
```

7.4.6 Step 5: tests

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

    #[test]
    fn eval_basic() {
        let ast = Expr::Bin {
            op: BinOp::Add,
            left: Box::new(Expr::Int(1)),
            right: Box::new(Expr::Bin {
                op: BinOp::Mul,
                left: Box::new(Expr::Int(2)),
                right: Box::new(Expr::Int(3)),
            }),
        };
        assert_eq!(ast.eval().unwrap(), 7);
    }

    #[test]
    fn eval_neg() {
        let ast = Expr::Neg(Box::new(Expr::Int(10)));
        assert_eq!(ast.eval().unwrap(), -10);
    }

    #[test]
    fn div_by_zero() {
        let ast = Expr::Bin {
            op: BinOp::Div,
            left: Box::new(Expr::Int(10)),
            right: Box::new(Expr::Int(0)),
        };
        assert!(ast.eval().is_err());
    }
}
```

7.4.7 Lab extension A: shared subtrees (DAG) using Rc

Sometimes you want to share a subtree without copying it. You can switch children to `Rc<Expr>`:

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

#[derive(Debug, Clone, PartialEq)]
enum ExprRc {
    Int(i64),
    Neg(Rc<ExprRc>),
    Bin { op: BinOp, left: Rc<ExprRc>, right: Rc<ExprRc> },
}
```

7.4.8 Lab extension B: adding parent pointers without cycles

If you add parent pointers, you must use `Weak` for the parent link. The safe rule:

- children own parents? never.

- parents own children strongly, children refer to parents weakly.

7.4.9 What you should gain from this lab

After this chapter you should be able to:

- choose the correct smart pointer for an ownership strategy,
- understand when reference counting is appropriate and when it is a code smell,
- use interior mutability deliberately and safely,
- prevent memory leaks caused by cycles using `Weak`,
- build recursive AST structures that are safe, testable, and extensible.

Smart pointers are not a replacement for ownership; they are the controlled extensions that let you model real systems without sacrificing Rust's guarantees.

Part III

Practical Power — Types, Traits, and API Design

struct and enum as Design Tools

Rust's power is not only about memory safety. It is also about **design correctness**: using types to make invalid states unrepresentable, using enums to model real state transitions, and using `Option` and `Result` to replace hidden failure paths with explicit, composable contracts.

This chapter treats `struct` and `enum` not as syntax, but as **engineering tools**. You will learn how to model state machines, how to embrace explicit absence and explicit failure, and how to design APIs that are both ergonomic and robust. The lab builds a protocol handler as a state machine to cement these ideas.

8.1 Modeling state machines with enum

8.1.1 Why enums are a state-machine tool

In many codebases, state machines are encoded using:

- integer flags,
- scattered booleans,
- magic strings,
- or loosely coupled fields where some combinations are invalid.

These designs frequently allow impossible combinations and require runtime checks everywhere. Rust's enums allow you to represent a system with **explicit states** and **state-specific data**. A value of an enum is *exactly one* of its variants at any time, and pattern matching forces you to handle transitions explicitly.

8.1.2 A minimal state machine

```rust
#[derive(Debug, Copy, Clone, PartialEq, Eq)]
enum ConnState {
    Disconnected,
    Connecting,
    Connected,
    Closing,
}

fn step(s: ConnState) -> ConnState {
    match s {
        ConnState::Disconnected => ConnState::Connecting,
        ConnState::Connecting => ConnState::Connected,
        ConnState::Connected => ConnState::Closing,
        ConnState::Closing => ConnState::Disconnected,
    }
}
```

This is already stronger than a boolean-based design because it is exhaustive. You cannot forget a case without the compiler telling you.

8.1.3 State-specific data makes invalid combinations unrepresentable

Real protocols often have state-dependent fields:

- in `Connecting`, you may have a retry counter or a timeout,

- in `Connected`, you have a session id or negotiated parameters,
- in `Closing`, you have a reason code.

Enums support this naturally:

```
#[derive(Debug, Clone)]
enum ConnState {
    Disconnected,
    Connecting { retries: u32 },
    Connected { session_id: u64, peer: String },
    Closing { reason: CloseReason },
}

#[derive(Debug, Copy, Clone)]
enum CloseReason {
    Timeout,
    RemoteClosed,
    ProtocolError,
}
```

This prevents nonsense like: “connected = true but session_id is None”.

8.1.4 Transitions as functions (state-in, state-out)

A strong pattern is to model transitions as pure functions:

- they take the current state,
- take an input event,
- return the next state (or an error).

```
#[derive(Debug, Copy, Clone)]
enum Event {
    Connect,
    HandshakeOk { session_id: u64 },
    Timeout,
    Close,
}

fn transition(state: ConnState, ev: Event) -> ConnState {
    match (state, ev) {
        (ConnState::Disconnected, Event::Connect) => ConnState::Connecting { retries: 0 },

        (ConnState::Connecting { .. }, Event::HandshakeOk { session_id }) =>
            ConnState::Connected { session_id, peer: "peer".to_string() },

        (ConnState::Connecting { .. }, Event::Timeout) =>
            ConnState::Closing { reason: CloseReason::Timeout },

        (ConnState::Connected { .. }, Event::Close) =>
            ConnState::Closing { reason: CloseReason::RemoteClosed },

        (s, _) => s, // default: ignore unexpected events for this simple example
    }
}
```

Note that by matching on `(state, event)` you encode allowed transitions directly.

8.1.5 State machine as a type-level discipline

The more you push into types, the less you need runtime validation. Professional design goal:

- illegal transitions should be difficult to express,
- legal transitions should be easy to write,

- state-specific data should only exist in the state where it is valid.

8.2 Option and Result as philosophy

8.2.1 The central philosophy: make absence and failure explicit

Rust treats absence and failure as **normal program outcomes** that should be modeled explicitly, not hidden behind:

- null pointers,
- out-of-band sentinel values,
- exception-based control flow,
- or implicit global error flags.

The two core types are:

- `Option<T>`: value is either present (`Some`) or absent (`None`).
- `Result<T, E>`: computation is either successful (`Ok`) or failed (`Err`).

8.2.2 Option replaces null as a design choice

```rust
fn find_user(id: u64) -> Option<String> {
    if id == 1 {
        Some("Ayman".to_string())
    } else {
        None
    }
}
```

Handling is explicit:

```
fn main() {
    match find_user(1) {
        Some(name) => println!("found {}", name),
        None => println!("not found"),
    }
}
```

This is not verbosity for its own sake; it forces correct handling of the absence case.

8.2.3 Result encodes failure as part of the API contract

```
fn parse_port(s: &str) -> Result<u16, String> {
    let n: u32 = s.parse().map_err(|_| "not a number".to_string())?;
    if n > u16::MAX as u32 {
        return Err("out of range".to_string());
    }
    Ok(n as u16)
}
```

Consumers cannot ignore failures accidentally. They must match or use ? to propagate.

8.2.4 Option/Result encourage compositional design

Rather than deeply nested checks, Rust code often composes:

- mapping,
- chaining,
- early returns via ?,
- and pattern matching when logic branches.

```rust
fn first_word_len(s: &str) -> Option<usize> {
    s.split_whitespace().next().map(|w| w.len())
}
```

This is small, explicit, and safe.

8.2.5 The deeper design lesson

`Option` and `Result` are not merely utility types. They encourage a system design discipline:

- define contracts where absence and failure are expected and modeled,
- force the caller to choose what to do,
- reduce hidden control flow and implicit states,
- create APIs that are easier to refactor and reason about.

8.3 Lab: implementing a protocol as a state machine

This lab implements a small text protocol handler as a state machine. The goal is not networking; the goal is to practice:

- modeling protocol phases with enums,
- representing illegal states as unrepresentable,
- using `Result` for errors and `Option` for incomplete input,
- implementing transitions in a structured way.

8.3.1 Protocol specification (simplified)

We define a simple handshake protocol where the peer must send:

- `HELLO <name>`
- then `AUTH <token>`
- then the session is established and accepts `PING` and `QUIT`.

Rules:

- Any invalid command in a given phase results in a protocol error.
- `QUIT` closes the session (allowed only after authentication).

8.3.2 Step 1: commands and parse logic

We parse lines into a command enum.

```rust
#[derive(Debug, Clone, PartialEq, Eq)]
enum Cmd {
    Hello { name: String },
    Auth  { token: String },
    Ping,
    Quit,
}

fn parse_cmd(line: &str) -> Result<Cmd, String> {
    let line = line.trim();
    if line.is_empty() {
        return Err(”empty command”.to_string());
    }
    let mut it = line.split_whitespace();
    let head = it.next().unwrap();

    match head {
        "HELLO" => {
            let name = it.next().ok_or_else(|| "HELLO requires a name".to_string())?;
            if it.next().is_some() {
                return Err("HELLO takes exactly one argument".to_string());
            }
            Ok(Cmd::Hello { name: name.to_string() })
        }
        "AUTH" => {
            let token = it.next().ok_or_else(|| "AUTH requires a token".to_string())?;
            if it.next().is_some() {
                return Err("AUTH takes exactly one argument".to_string());
            }
            Ok(Cmd::Auth { token: token.to_string() })
        }
        "PING" => {
            if it.next().is_some() {
                return Err("PING takes no arguments".to_string());
            }
            Ok(Cmd::Ping)
        }
        "QUIT" => {
            if it.next().is_some() {
                return Err("QUIT takes no arguments".to_string());
            }
            Ok(Cmd::Quit)
        }
        _ => Err(format!("unknown command: {}", head)),
    }
}
```

8.3.3 Step 2: define protocol states

Each state contains only the data valid for that phase.

```
#[derive(Debug, Clone)]
enum State {
    AwaitHello,
    AwaitAuth { name: String },
    Established { name: String, session_id: u64 },
    Closed,
}

#[derive(Debug, Copy, Clone, PartialEq, Eq)]
enum ProtoError {
    UnexpectedCommand,
    InvalidToken,
    Closed,
}
```

8.3.4 Step 3: define responses

The state machine produces responses. We keep this explicit:

```
#[derive(Debug, Clone, PartialEq, Eq)]
enum Response {
    Ok(String),
    Err(String),
    Bye,
}
```

8.3.5 Step 4: implement the protocol machine

The machine consumes commands and advances state.

```
struct Machine {
    state: State,
    next_session: u64,
}

impl Machine {
    fn new() -> Self {
        Self { state: State::AwaitHello, next_session: 1 }
    }

    fn handle(&mut self, cmd: Cmd) -> Result<Response, ProtoError> {
        match (&mut self.state, cmd) {
            (State::AwaitHello, Cmd::Hello { name }) => {
                self.state = State::AwaitAuth { name };
                Ok(Response::Ok(”HELLO-OK”.to_string()))
            }

            (State::AwaitAuth { name }, Cmd::Auth { token }) => {
                if token != ”secret” {
                    return Err(ProtoError::InvalidToken);
                }
                let sid = self.next_session;
                self.next_session += 1;
                let name = name.clone();
                self.state = State::Established { name, session_id: sid };
                Ok(Response::Ok(format!(”AUTH-OK {}”, sid)))
            }

            (State::Established { name, .. }, Cmd::Ping) => {
                Ok(Response::Ok(format!(”PONG {}”, name)))
            }

            (State::Established { .. }, Cmd::Quit) => {
                self.state = State::Closed;
                Ok(Response::Bye)
            }

            (State::Closed, _) => Err(ProtoError::Closed),

            _ => Err(ProtoError::UnexpectedCommand),
        }
    }
}
```

8.3.6 Step 5: driving the machine with input lines

This function demonstrates how to connect parsing (`Result`) with protocol handling and explicit errors.

```
fn process_line(m: &mut Machine, line: &str) -> Response {
    let cmd = match parse_cmd(line) {
        Ok(c) => c,
        Err(e) => return Response::Err(format!("PARSE-ERR {}", e)),
    };

    match m.handle(cmd) {
        Ok(r) => r,
        Err(ProtoError::UnexpectedCommand) =>
        ↪   Response::Err("PROTO-ERR unexpected command".to_string()),
        Err(ProtoError::InvalidToken) =>
        ↪   Response::Err("PROTO-ERR invalid token".to_string()),
        Err(ProtoError::Closed) => Response::Err("PROTO-ERR session closed".to_string()),
    }
}
```

8.3.7 Step 6: tests

Tests enforce allowed transitions and ensure impossible ones are caught.

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

    #[test]
    fn happy_path() {
        let mut m = Machine::new();

        assert_eq!(process_line(&mut m, "HELLO Ayman"),
        ↪   Response::Ok("HELLO-OK".to_string()));
        match process_line(&mut m, "AUTH secret") {
            Response::Ok(s) => assert!(s.starts_with("AUTH-OK ")),
            _ => panic!("expected auth ok"),
        }
        assert_eq!(process_line(&mut m, "PING"), Response::Ok("PONG Ayman".to_string()));
        assert_eq!(process_line(&mut m, "QUIT"), Response::Bye);
    }

    #[test]
    fn protocol_errors() {
        let mut m = Machine::new();

        assert!(matches!(
            process_line(&mut m, "PING"),
            Response::Err(_)
        ));

        assert_eq!(process_line(&mut m, "HELLO X"), Response::Ok("HELLO-OK".to_string()));
        assert!(matches!(
            process_line(&mut m, "AUTH wrong"),
            Response::Err(_)
        ));
    }

    #[test]
    fn parse_errors() {
        let mut m = Machine::new();
        assert!(matches!(
            process_line(&mut m, "HELLO"),
            Response::Err(_)
        ));
        assert!(matches!(
            process_line(&mut m, ""),
            Response::Err(_)
        ));
    }
}
```

8.3.8 Lab extension ideas

To deepen the exercise:

- Add a `DATA <len>` command that transitions into a `ReceivingData` state.
- Store partial input and return `Option<Response>` for incomplete frames.
- Introduce timeouts as explicit events that force transitions.
- Separate parsing, state machine, and IO layers into modules with clear APIs.

8.3.9 What this lab teaches

This lab should leave you with practical intuition:

- Enums are not just “either/or” types; they are a rigorous state modeling tool.
- `Option` and `Result` shift failure and absence from hidden behavior into explicit design.
- Pattern matching makes protocol correctness and completeness a compile-time discipline.
- Your design becomes safer and more maintainable because illegal states are hard to represent.

These are the design instincts that scale to real systems: network protocols, parsers, compilers, device drivers, and long-lived services.

Traits from Basics to Advanced Design

Traits are Rust's primary abstraction mechanism for designing reusable, testable, and performant APIs. They unify several ideas that are often spread across different features in other languages: interfaces, constraints, ad-hoc polymorphism, extension methods, and dynamic dispatch. Traits also integrate deeply with Rust's ownership and borrowing model, which means good trait design often leads naturally to good resource and performance behavior.

This chapter builds traits progressively:

- trait bounds as the foundation of generic programming,
- default methods as a tool for stable, evolvable APIs,
- associated types versus generics as a key design choice,
- trait objects as the runtime-polymorphism escape hatch,
- and a lab that implements a small plugin framework with both static and dynamic dispatch paths.

9.1 Trait bounds

9.1.1 Why trait bounds exist

When you write generic code, you must state what operations are allowed on the generic type. Trait bounds are Rust's way of expressing those requirements precisely and compositionally.

9.1.2 The simplest bound

```
fn print_debug<T: std::fmt::Debug>(x: T) {
    println!("{:?}", x);
}

fn main() {
    print_debug(123);
    print_debug("hello");
}
```

Here, `T: Debug` is a promise: `T` supports debug formatting.

9.1.3 Bounds on references and ownership choices

Bounds are often applied to references rather than owned values to avoid moves and cloning.

```
fn print_debug_ref<T: std::fmt::Debug>(x: &T) {
    println!("{:?}", x);
}
```

This design is typically more flexible: it accepts both owned and borrowed callers without consuming data.

9.1.4 Multiple bounds

You can combine bounds:

```
fn show<T: std::fmt::Debug + std::fmt::Display>(x: &T) {
    println!("display = {}, debug = {:?}", x, x);
}
```

9.1.5 Where-clauses for readability

As bounds grow, `where` clauses keep signatures readable.

```
fn transform<T, U>(xs: &[T]) -> Vec<U>
where
    T: Clone,
    U: From<T>,
{
    xs.iter().cloned().map(U::from).collect()
}
```

9.1.6 Trait bounds in impl blocks

Bounds also appear in implementations, allowing methods only when the type meets conditions.

```
struct Wrapper<T> {
    value: T,
}

impl<T> Wrapper<T> {
    fn new(value: T) -> Self {
        Self { value }
    }
}
```

```rust
impl<T> Wrapper<T>
where
    T: std::fmt::Display,
{
    fn show(&self) {
        println!("{}", self.value);
    }
}
```

9.1.7 Bounds as API contracts, not just compiler requirements

In professional Rust, bounds are part of your API design language:

- too weak bounds make implementations impossible or error-prone,
- too strong bounds make APIs unusable and force callers into cloning or allocations,
- correct bounds describe the minimal capability needed.

9.2 Default methods

9.2.1 Why default methods matter

Default methods allow you to evolve traits without breaking all implementors and provide shared logic in one place. They also encode best practices and invariants.

9.2.2 A trait with a default method

```rust
trait Counter {
    fn next(&mut self) -> u64;

    fn next_n(&mut self, n: usize) -> Vec<u64> {
        let mut out = Vec::with_capacity(n);
        for _ in 0..n {
            out.push(self.next());
        }
        out
    }
}

struct Inc {
    cur: u64,
}

impl Counter for Inc {
    fn next(&mut self) -> u64 {
        self.cur += 1;
        self.cur
    }
}

fn main() {
    let mut c = Inc { cur: 0 };
    println!("{:?}", c.next_n(5));
}
```

Here, implementors only provide `next`, while `next_n` is shared.

9.2.3 Default methods can call required methods

A powerful design pattern:

- define a minimal required core,
- build richer behavior as default methods on top.

This keeps the trait easy to implement, while giving users rich functionality.

9.2.4 Default methods and invariants

Default methods can encode invariants so implementors have fewer ways to get it wrong. When your trait represents a protocol or a state machine, default methods can enforce correct sequencing.

9.3 Associated types vs generics

This section is a key design decision point. Both approaches express “a trait relates types to other types,” but they affect usability, inference, and extensibility.

9.3.1 Generics in traits: explicit at every use

A trait may be generic:

```
trait ConvertTo<T> {
    fn convert(&self) -> T;
}
```

This means the trait is parameterized by `T`. A type could implement `ConvertTo<i32>` and `ConvertTo<String>` simultaneously.

9.3.2 Associated types: the implementor chooses the type

With associated types, the trait declares a placeholder type that each implementor defines.

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

The implementor chooses what `Item` is.

9.3.3 Why associated types exist

Associated types reduce clutter and improve inference when there is a single “natural” related type.

In an iterator, each iterator has exactly one `Item` type. You do not want to write `IteratorLike<T>` everywhere; you want `IteratorLike` with `Item` determined by the type itself.

9.3.4 Side-by-side example: a parsing trait

Generic version:

```
trait ParseTo<T> {
    fn parse(&self) -> Result<T, String>;
}
```

Associated-type version:

```
trait Parser {
    type Output;
    fn parse(&self) -> Result<Self::Output, String>;
}
```

When is each better?

- Use **generics** when a single type might naturally support multiple target types.
- Use **associated types** when the trait has one canonical related type per implementor.

9.3.5 Associated types and trait objects

Associated types are often easier to use in trait-object designs because they avoid needing to name type parameters at the trait level, but they still require you to fix associated types when making objects.

9.4 Trait objects and dynamic dispatch

9.4.1 Static dispatch vs dynamic dispatch

Rust supports two primary polymorphism strategies:

- **Static dispatch (generics)**: resolved at compile time, typically monomorphized, often fastest.
- **Dynamic dispatch (trait objects)**: resolved at runtime via a vtable, enables heterogeneous collections and plugin-like designs.

9.4.2 Static dispatch example

```rust
trait Greeter {
    fn greet(&self) -> String;
}

struct English;
impl Greeter for English {
    fn greet(&self) -> String { "hello".to_string() }
}

fn greet_all<T: Greeter>(g: &T) {
    println!("{}", g.greet());
}
```

9.4.3 Trait objects: `dyn Trait`

A trait object is typically used behind a pointer like `&dyn Trait`, `Box<dyn Trait>`, `Arc<dyn Trait>`. This enables storing different concrete types behind a common interface.

```
fn main() {
    let gs: Vec<Box<dyn Greeter>> = vec![
        Box::new(English),
    ];

    for g in gs {
        println!("{}", g.greet());
    }
}
```

9.4.4 Object safety (practical constraints)

Not every trait can become a trait object. For a trait to be used as `dyn Trait`, its methods must be compatible with dynamic dispatch. Practically, that means:

- methods must not require knowing `Self` size at compile time in a way incompatible with `dyn`,
- return types should not be `Self` unless boxed or otherwise erased,
- generic methods generally prevent object safety.

A common pattern is to provide:

- a generic API for static dispatch,
- and a separate object-safe trait for dynamic dispatch if needed.

9.4.5 Performance perspective

Dynamic dispatch has:

- an extra indirection (vtable call),

- possible missed inlining opportunities,
- but typically small overhead compared to IO or large computations.

Use dynamic dispatch when it simplifies architecture significantly (plugins, heterogeneous pipelines, runtime configuration). Use static dispatch when performance and inlining matter and types are known.

9.5 Lab: a small plugin framework

This lab builds a small plugin framework in two layers:

- **A statically dispatched pipeline** for maximal performance when plugins are known at compile time.
- **A dynamically dispatched registry** for runtime selection and heterogeneous storage.

The example plugins operate on strings: they transform input text (trim, uppercase, replace patterns).

9.5.1 Step 1: define a plugin trait

We want an object-safe trait:

- methods take `&self`,
- return owned outputs for simplicity,
- avoid generics in the trait itself.

```
trait Plugin {
    fn name(&self) -> &'static str;
    fn transform(&self, input: &str) -> String;

    fn enabled_by_default(&self) -> bool {
        true
    }
}
```

9.5.2 Step 2: implement a few plugins

```
struct Trim;
impl Plugin for Trim {
    fn name(&self) -> &'static str { "trim" }
    fn transform(&self, input: &str) -> String { input.trim().to_string() }
}

struct Upper;
impl Plugin for Upper {
    fn name(&self) -> &'static str { "upper" }
    fn transform(&self, input: &str) -> String { input.to_uppercase() }
}

struct ReplaceSpaces;
impl Plugin for ReplaceSpaces {
    fn name(&self) -> &'static str { "replace_spaces" }
    fn transform(&self, input: &str) -> String { input.replace(' ', "_") }
    fn enabled_by_default(&self) -> bool { false }
}
```

9.5.3 Step 3: a registry using trait objects

```
use std::collections::HashMap;

struct Registry {
    plugins: HashMap<&'static str, Box<dyn Plugin>>,
}

impl Registry {
    fn new() -> Self {
        Self { plugins: HashMap::new() }
    }

    fn register(&mut self, p: Box<dyn Plugin>) {
        let name = p.name();
        self.plugins.insert(name, p);
    }

    fn get(&self, name: &str) -> Option<&dyn Plugin> {
        self.plugins.get(name).map(|b| b.as_ref())
    }

    fn defaults(&self) -> Vec<&dyn Plugin> {
        self.plugins
            .values()
            .filter(|p| p.enabled_by_default())
            .map(|p| p.as_ref())
            .collect()
    }
}
```

9.5.4 Step 4: executing a pipeline

```
fn run_pipeline(mut s: String, ps: &[&dyn Plugin]) -> String {
    for p in ps {
        s = p.transform(&s);
    }
    s
}
```

9.5.5 Step 5: end-to-end demo

```
fn main() {
    let mut r = Registry::new();
    r.register(Box::new(Trim));
    r.register(Box::new(Upper));
    r.register(Box::new(ReplaceSpaces));

    let input = "  hello rust world  ".to_string();

    let defaults = r.defaults();
    let out = run_pipeline(input, &defaults);
    println!("{}", out); // "HELLO RUST WORLD" (trim + upper)

    if let Some(p) = r.get("replace_spaces") {
        let out2 = p.transform("hello rust");
        println!("{}", out2); // "hello_rust"
    }
}
```

9.5.6 Lab extension A: static dispatch pipeline

If plugins are known at compile time, you can avoid trait objects and dynamic dispatch by using generics and tuples, or a simple vector of function pointers. For a minimal static approach:

```
fn apply_all(input: &str, fs: &[fn(&str) -> String]) -> String {
    let mut s = input.to_string();
    for f in fs {
        s = f(&s);
    }
    s
}

fn trim_fn(s: &str) -> String { s.trim().to_string() }
fn upper_fn(s: &str) -> String { s.to_uppercase() }

fn main() {
    let fs: [fn(&str) -> String; 2] = [trim_fn, upper_fn];
    let out = apply_all("  hi  ", &fs);
    println!("{}", out);
}
```

9.5.7 Lab extension B: associated types for richer plugins

If plugins have different output types, you can design a trait with an associated type. However, heterogeneous storage becomes harder. The typical solution is:

- keep an object-safe trait for runtime plugins with a unified output type,
- use associated types for compile-time pipelines where types remain known and composable.

9.5.8 Lab extension C: errors and Result

Upgrade `transform` to return `Result<String, E>` (or `Result<String, String>` for simplicity) so that plugins can fail and pipelines can short-circuit. This forces explicit error handling and aligns with Rust's design philosophy.

9.5.9 What you should gain from this lab

After this chapter, you should be able to:

- express precise capability requirements using trait bounds,

- use default methods to create stable, evolvable APIs,
- choose between generics and associated types as a deliberate design decision,
- understand when trait objects are necessary and what dynamic dispatch implies,
- design small plugin-like architectures without sacrificing safety or clarity.

Traits are Rust's abstraction backbone. With good trait design, you can build systems that are both high-level and zero-cost where it matters, while preserving explicit correctness contracts.

Generics and Constraint Engineering

Generics are Rust's primary mechanism for writing reusable code without sacrificing performance. The language is designed so that abstractions can be expressed at a high level while still compiling down to efficient machine code. But generics are not only about reuse; they are also about **constraint engineering**: specifying the exact capabilities required from a type so that your API is both powerful and safe, without forcing callers into unnecessary cloning, allocations, or runtime overhead.

This chapter focuses on the practical skills that separate "generic code that compiles" from "generic code that scales in real systems":

- using `where` clauses to keep signatures readable,
- expressing complex bounds correctly and minimally,
- understanding zero-cost abstractions in practice (monomorphization, inlining, specialization-like patterns),
- and building a small generic collections library as a lab.

10.1 where clauses

10.1.1 Why where exists

As soon as you combine multiple type parameters and multiple trait bounds, inline bound syntax becomes unreadable. where clauses move constraints into a separate block so the function signature expresses intent first, and constraints second.

10.1.2 Inline bounds become noisy quickly

```
fn merge_and_show<T: Clone + std::fmt::Debug, U: Into<T> + Clone + std::fmt::Debug>(
    a: &[T],
    b: &[U],
) -> Vec<T> {
    let mut out = a.to_vec();
    out.extend(b.iter().cloned().map(Into::into));
    println!("{:?}", out);
    out
}
```

This works, but the signature is hard to scan.

10.1.3 The same function with where

```
fn merge_and_show<T, U>(a: &[T], b: &[U]) -> Vec<T>
where
    T: Clone + std::fmt::Debug,
    U: Clone + Into<T> + std::fmt::Debug,
{
    let mut out = a.to_vec();
    out.extend(b.iter().cloned().map(Into::into));
    println!("{:?}", out);
    out
}
```

10.1.4 where clauses on impl blocks

`where` is also essential for conditional methods: you implement or enable methods only when constraints are satisfied.

```
struct Bag<T> {
    xs: Vec<T>,
}

impl<T> Bag<T> {
    fn new() -> Self {
        Self { xs: Vec::new() }
    }

    fn push(&mut self, x: T) {
        self.xs.push(x);
    }
}

impl<T> Bag<T>
where
    T: std::fmt::Debug,
{
    fn debug_dump(&self) {
        println!("{:?}", self.xs);
    }
}
```

This lets you keep the base type broadly usable while offering extra capabilities conditionally.

10.1.5 where clauses improve error messages

Complex generic errors are unavoidable sometimes. Well-structured `where` clauses often produce clearer compiler diagnostics because constraints are more explicit and localized.

10.2 Complex bounds

10.2.1 The art: minimal required capability

The goal of bounds is not to ask for as much as possible. The goal is to require the minimal capability that makes the implementation possible.
Common engineering mistakes:

- adding `Clone` “to make it easy” when borrowing would suffice,
- requiring `Debug` for internal logging rather than for the public API,
- forcing `Send + Sync` for types that never cross threads.

10.2.2 Bounds on references, not values

Prefer bounds on borrowed inputs when you do not need ownership.

```
fn sum<I>(it: I) -> i64
where
    I: IntoIterator<Item = i64>,
{
    it.into_iter().sum()
}

fn sum_refs<'a, I>(it: I) -> i64
where
```

```
    I: IntoIterator<Item = &'a i64>,
{
    it.into_iter().copied().sum()
}
```

This demonstrates a common pattern:

- accept owned items when that is natural,
- accept references when callers want to retain ownership.

10.2.3 Higher constraints: tying types together

Often you need constraints that connect associated types and generic types.

```
fn collect_mapped<I, F, U>(it: I, f: F) -> Vec<U>
where
    I: IntoIterator,
    F: FnMut(I::Item) -> U,
{
    it.into_iter().map(f).collect()
}
```

Here, `I::Item` (an associated type) is tied to `F` and to `U` through the bound.

10.2.4 Complex bounds with multiple generic parameters

```
fn join_as_strings<T, I>(it: I, sep: &str) -> String
where
    I: IntoIterator<Item = T>,
    T: std::fmt::Display,
{
    let mut out = String::new();
    let mut first = true;

    for x in it {
        if !first {
            out.push_str(sep);
        }
        first = false;
        out.push_str(&x.to_string());
    }

    out
}
```

This function expresses:

- we can accept any iterable of any type,
- as long as the element type can be displayed.

10.2.5 Conditional implementations with multiple bounds

Conditional impl blocks let you build layered APIs.

```
struct SmallVec<T, const N: usize> {
    buf: [Option<T>; N],
    len: usize,
}

impl<T, const N: usize> SmallVec<T, N> {
    fn new() -> Self {
        Self { buf: std::array::from_fn(|_| None), len: 0 }
    }
}
```

```
impl<T, const N: usize> SmallVec<T, N>
where
    T: Copy,
{
    fn filled(x: T) -> Self {
        let mut v = Self::new();
        for i in 0..N {
            v.buf[i] = Some(x);
            v.len += 1;
        }
        v
    }
}
```

This demonstrates:

- base functionality for all `T`,
- extra functionality only when `T: Copy`.

10.2.6 Bound patterns you will use often

Practical patterns that appear frequently in high-quality Rust libraries:

- `T: AsRef<str>` to accept both `String` and `&str`,
- `T: Into<Vec<u8»` to accept different buffer sources,
- `I: IntoIterator<Item = T>` for flexible iteration inputs,
- `T: Borrow<K>` for map/set lookups with different key forms,
- `T: Send + Sync + ’static` for cross-thread and long-lived storage (only when needed).

10.3 Zero-cost abstractions in practice

10.3.1 What "zero-cost" means in Rust

The goal is:

- high-level abstractions compile down to code as efficient as hand-written specialized code,
- without hidden runtime dispatch, allocation, or reference counting,
- unless you explicitly choose those costs (trait objects, `Box`, `Arc`, etc.).

In practice, Rust achieves this through:

- **monomorphization**: generic functions are compiled into specialized versions for concrete types,
- **inlining**: small generic functions often disappear into callers,
- **static dispatch**: trait bounds resolve calls at compile time,
- **explicit opt-in for dynamic dispatch**: `dyn Trait` when you need it.

10.3.2 Static dispatch example: generic function specializes

```rust
fn add_one<T>(x: T) -> T
where
    T: std::ops::Add<Output = T> + From<u8>,
{
    x + T::from(1)
}

fn main() {
    let a = add_one(10_i32);
    let b = add_one(10_u64);
    println!("{}, {}", a, b);
}
```

Conceptually:

- the compiler creates a version for `i32` and one for `u64`,
- each is as if you wrote it manually,
- there is no runtime type checking.

10.3.3 Iterator pipelines are typically zero-cost

Iterators look high-level, but they are designed to be optimized heavily.

```
fn sum_even_squares(xs: &[i32]) -> i32 {
    xs.iter()
      .copied()
      .filter(|x| x % 2 == 0)
      .map(|x| x * x)
      .sum()
}
```

The abstraction cost is paid at compile time; the runtime often sees tight loops after optimization.

10.3.4 The boundary where costs become real

Costs appear when you choose runtime features explicitly:

- trait objects (`dyn Trait`) introduce vtable calls,
- reference counting (`Rc`, `Arc`) introduces increments/decrements,

- interior mutability types add runtime checks or locking,
- heap allocation (`Box`, `Vec`, `String`) is explicit.

Rust's philosophy is not "no costs"; it is "costs are explicit, visible, and chosen deliberately."

10.4 Lab: building a small generic collections library

This lab builds a minimal generic collections module with:

- a small generic container,
- flexible iteration APIs,
- carefully designed trait bounds,
- and tests that enforce correctness.

The goal is to practice writing clean generic APIs with minimal constraints and no unnecessary cloning.

10.4.1 Lab design: `StableBag<T>`

We build a container that:

- stores elements in a `Vec<T>`,
- provides insertion and retrieval,
- supports generic extension from any iterator,
- provides functional-style operations without forcing allocation unless requested.

10.4.2 Step 1: define the container

```rust
pub struct StableBag<T> {
    xs: Vec<T>,
}

impl<T> StableBag<T> {
    pub fn new() -> Self {
        Self { xs: Vec::new() }
    }

    pub fn with_capacity(n: usize) -> Self {
        Self { xs: Vec::with_capacity(n) }
    }

    pub fn len(&self) -> usize {
        self.xs.len()
    }

    pub fn is_empty(&self) -> bool {
        self.xs.is_empty()
    }

    pub fn push(&mut self, x: T) {
        self.xs.push(x);
    }

    pub fn iter(&self) -> std::slice::Iter<'_, T> {
        self.xs.iter()
    }

    pub fn iter_mut(&mut self) -> std::slice::IterMut<'_, T> {
        self.xs.iter_mut()
    }
}
```

10.4.3 Step 2: extend from any iterator

```
impl<T> StableBag<T> {
    pub fn extend<I>(&mut self, it: I)
    where
        I: IntoIterator<Item = T>,
    {
        self.xs.extend(it);
    }
}
```

This is a classic “flexibility with minimal bounds” design:

- any iterable that produces `T` can extend the bag.

10.4.4 Step 3: map without forcing ownership

We provide two mapping styles:

- `map_ref`: transforms using borrowed items (`&T`) and produces a `Vec<U>`,
- `map_into`: consumes the bag and transforms owned items (`T`) into `U`.

```
impl<T> StableBag<T> {
    pub fn map_ref<U, F>(&self, mut f: F) -> Vec<U>
    where
        F: FnMut(&T) -> U,
    {
        self.xs.iter().map(|x| f(x)).collect()
    }
    pub fn map_into<U, F>(self, mut f: F) -> StableBag<U>
    where
        F: FnMut(T) -> U,
    {
        let xs = self.xs.into_iter().map(|x| f(x)).collect();
        StableBag { xs }
    }
}
```

This demonstrates a critical design lesson:

- provide a borrowed API when you can,
- provide an owned-consuming API when it is necessary or more efficient.

10.4.5 Step 4: lookup without requiring Clone

We implement a find method that returns a reference, not a copy.

```
impl<T> StableBag<T> {
    pub fn find<P>(&self, mut pred: P) -> Option<&T>
    where
        P: FnMut(&T) -> bool,
    {
        self.xs.iter().find(|x| pred(x))
    }
}
```

Returning `&T` avoids cloning and respects ownership.

10.4.6 Step 5: display and debug helpers with conditional bounds

We add methods only when `T` supports the required formatting.

```rust
impl<T> StableBag<T>
where
    T: std::fmt::Display,
{
    pub fn join_display(&self, sep: &str) -> String {
        let mut out = String::new();
        let mut first = true;
        for x in &self.xs {
            if !first {
                out.push_str(sep);
            }
            first = false;
            out.push_str(&x.to_string());
        }
        out
    }
}

impl<T> StableBag<T>
where
    T: std::fmt::Debug,
{
    pub fn debug_dump(&self) {
        println!("{:?}", self.xs);
    }
}
```

This is constraint engineering in practice:

- the base container remains usable for all `T`,
- extra capabilities appear only when valid.

10.4.7 Step 6: tests

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

    #[test]
    fn basic_ops() {
        let mut b = StableBag::new();
        assert!(b.is_empty());
        b.push(1);
        b.push(2);
        assert_eq!(b.len(), 2);
    }

    #[test]
    fn extend_from_iter() {
        let mut b = StableBag::new();
        b.extend([1, 2, 3]);
        assert_eq!(b.len(), 3);
    }

    #[test]
    fn map_ref_and_map_into() {
        let mut b = StableBag::new();
        b.extend([1, 2, 3]);

        let squares = b.map_ref(|x| x * x);
        assert_eq!(squares, vec![1, 4, 9]);

        let b2 = b.map_into(|x| x.to_string());
        assert_eq!(b2.join_display(","), "1,2,3");
    }
    #[test]
    fn find_returns_ref() {
        let mut b = StableBag::new();
        b.extend([”aa”.to_string(), ”bbb”.to_string()]);
        let r = b.find(|s| s.len() == 3).unwrap();
        assert_eq!(r, ”bbb”);
    }
}
```

10.4.8 Lab extensions

To expand this into a richer micro-library:

- Add `IntoIterator` impls for `&StableBag<T>` and `StableBag<T>`.
- Add `retain` with `FnMut(&T) -> bool`.
- Add a small `SmallVec`-style container using const generics for stack storage.
- Provide a trait `Collection` with associated type `Item` and implement it for multiple containers.
- Add feature flags that enable extra methods only under certain bounds or environment needs.

10.4.9 What you should gain from this lab

By completing this lab you should be able to:

- write generic APIs with minimal, correct bounds,
- use `where` clauses to keep constraints readable,

- structure a library so core functionality remains widely usable,
- add conditional capabilities through bounded impl blocks,
- and reason about when abstractions remain zero-cost versus when you opt into runtime costs.

Generics are not complexity for its own sake. In Rust, they are the engineering language that lets you build high-level APIs that still compile into predictable, efficient systems code.

Professional Error Handling

Error handling is where Rust's design philosophy becomes most visible: failure is not exceptional; it is part of the program's contract. Professional Rust code treats errors as first-class values that are:

- explicit in signatures,
- composable across layers,
- informative without leaking internal details,
- and designed for long-term maintainability.

This chapter focuses on real-world patterns:

- practical `Result` patterns that scale,
- the difference between application errors and library errors,
- unified error design across a project,
- and a lab that builds a project-wide error system suitable for multi-module systems code.

11.1 Result patterns

11.1.1 The signature is the contract

In Rust, a function that can fail should express that in its return type:

```
fn read_config(path: &str) -> Result<String, std::io::Error> {
    std::fs::read_to_string(path)
}
```

This makes failure explicit and forces callers to decide:

- handle locally,
- propagate,
- transform,
- or convert to a unified error type.

11.1.2 The ? operator: structured propagation

? is not a shortcut; it is a disciplined propagation mechanism:

- it returns early on `Err`,
- it unwraps `Ok` values,
- it converts error types via `From` when needed.

```
fn load_user(path: &str) -> Result<String, std::io::Error> {
    let s = std::fs::read_to_string(path)?;
    Ok(s)
}
```

11.1.3 Mapping and enriching errors

It is often correct to attach context before returning.

```
fn parse_port(s: &str) -> Result<u16, String> {
    let n: u32 = s.parse().map_err(|_| format!("port is not a number: {}", s))?;
    if n > u16::MAX as u32 {
        return Err(format!("port out of range: {}", n));
    }
    Ok(n as u16)
}
```

11.1.4 Converting Option to Result

A common pattern is turning missing values into structured errors.

```
fn require_env(name: &str) -> Result<String, String> {
    std::env::var(name).map_err(|_| format!("missing environment variable: {}", name))
}
```

If you already have an `Option<T>`, use `ok_or` / `ok_or_else`:

```
fn first_word(s: &str) -> Result<&str, String> {
    s.split_whitespace()
        .next()
        .ok_or_else(|| "no words".to_string())
}
```

11.1.5 Early-return validation

Validation logic is often clearest as early returns:

```
fn checked_div(a: i64, b: i64) -> Result<i64, String> {
    if b == 0 {
        return Err("division by zero".to_string());
    }
    Ok(a / b)
}
```

11.1.6 Error layering: keep low-level errors low-level

A key discipline:

- low-level modules return precise errors for their domain,
- higher layers decide what to expose and how to present it.

This prevents leakage of internal implementation details.

11.2 Application errors vs library errors

11.2.1 Application errors: user-facing, unified, contextual

Applications often want a **single error type** that:

- can represent failures across the entire program,
- carries enough context for logging and debugging,
- can be rendered into user-friendly messages,
- can map into exit codes or HTTP status codes.

In applications, it is common to unify errors early.

11.2.2 Library errors: stable, precise, non-leaky

Libraries face different constraints:

- error types become part of the public API and must remain stable,
- callers may want to match error variants programmatically,
- the library must avoid forcing a specific error framework on users,
- errors should be precise and domain-specific.

11.2.3 Design rule: libraries should not over-erase errors

A library that returns only `String` loses structured matching ability. A library that returns only `Box<dyn std::error::Error>` may make matching harder unless you provide downcasting or expose variants separately.

A strong approach for libraries is:

- define an `enum Error` with meaningful variants,
- implement `Display` and `Error`,
- optionally carry a source error for IO or parsing details.

11.3 Unified error design

11.3.1 Why unify errors

Projects with multiple modules typically face:

- IO errors, parse errors, protocol errors,

- invariant violations,
- configuration failures,
- external library failures.

Without a unified strategy, code becomes:

- inconsistent in propagation,
- inconsistent in logging,
- and harder to refactor.

A unified error design gives:

- consistent signatures,
- consistent error presentation,
- consistent handling at boundaries (CLI, service endpoints, FFI).

11.3.2 A practical unified error enum

In a project, define a top-level error type that represents cross-cutting failures while preserving sources.

```
use std::fmt;

#[derive(Debug)]
pub enum AppError {
    Io { op: &'static str, path: String, source: std::io::Error },
    Parse { what: &'static str, input: String, msg: String },
    Protocol { msg: String },
    Config { msg: String },
}

impl fmt::Display for AppError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            AppError::Io { op, path, source } =>
                write!(f, "io error during {} on {}: {}", op, path, source),
            AppError::Parse { what, input, msg } =>
                write!(f, "parse error for {} ({}): {}", what, input, msg),
            AppError::Protocol { msg } =>
                write!(f, "protocol error: {}", msg),
            AppError::Config { msg } =>
                write!(f, "config error: {}", msg),
        }
    }
}

impl std::error::Error for AppError {
    fn source(&self) -> Option<&(dyn std::error::Error + 'static)> {
        match self {
            AppError::Io { source, .. } => Some(source),
            _ => None,
        }
    }
}
```

This design:

- encodes categories as variants,
- includes context fields (`op`, `path`, `what`),
- optionally stores a source error for chaining.

11.3.3 Convenience constructors for consistency

Avoid repeating error formatting everywhere by providing constructors.

```rust
impl AppError {
    pub fn io(op: &'static str, path: impl Into<String>, source: std::io::Error) -> Self {
        AppError::Io { op, path: path.into(), source }
    }

    pub fn parse(what: &'static str, input: impl Into<String>, msg: impl Into<String>) ->
    ↪  Self {
        AppError::Parse { what, input: input.into(), msg: msg.into() }
    }

    pub fn protocol(msg: impl Into<String>) -> Self {
        AppError::Protocol { msg: msg.into() }
    }

    pub fn config(msg: impl Into<String>) -> Self {
        AppError::Config { msg: msg.into() }
    }
}
```

11.3.4 Using `From` to enable `?`

A professional technique is to use `From` conversions so `?` can automatically unify errors. However, when you need context (like path and operation), `From` alone is insufficient. In those cases, use `map_err` to inject context.

```rust
fn read_file(path: &str) -> Result<String, AppError> {
    std::fs::read_to_string(path).map_err(|e| AppError::io("read_to_string", path, e))
}
```

11.3.5 Boundary handling: errors become outputs

At program boundaries, errors become:

- CLI exit codes and messages,
- log entries and structured logs,
- HTTP status codes and JSON error bodies,
- or FFI error codes.

Core rule:

- internal errors should carry context and preserve sources,
- boundary layers decide how to present them.

11.4 Lab: building a project-wide error system

This lab builds a small multi-module project with a unified error type. The project simulates a realistic workflow:

- read configuration from a file,
- parse it,
- load a resource file,
- parse a protocol command,
- and execute a small action.

11.4.1 Step 1: define the project error type

Use the AppError definition shown earlier.

11.4.2 Step 2: build modules that return Result<AppError>

Module: config

```
fn parse_kv_line(line: &str) -> Result<(&str, &str), AppError> {
    let mut it = line.splitn(2, '=');
    let k = it.next().unwrap_or("").trim();
    let v = it.next().unwrap_or("").trim();
    if k.is_empty() || v.is_empty() {
        return Err(AppError::parse("config line", line, "expected key=value"));
    }
    Ok((k, v))
}

#[derive(Debug, Clone)]
struct Config {
    data_path: String,
}

fn load_config(path: &str) -> Result<Config,, AppError> {
    let text = std::fs::read_to_string(path).map_err(|e| AppError::io("read config", path,
    ↪   e))?;

    let mut data_path: Option<String> = None;

    for raw in text.lines() {
        let line = raw.trim();
        if line.is_empty() || line.starts_with('#') {
            continue;
        }
        let (k, v) = parse_kv_line(line)?;
        if k == "data_path" {
            data_path = Some(v.to_string());
        }
    }

    let data_path = data_path.ok_or_else(|| AppError::config("missing data_path"))?;
    Ok(Config { data_path })
}
```

11.4.3 Step 3: protocol parsing with Result

```
#[derive(Debug, Clone, PartialEq, Eq)]
enum Cmd {
    Get { key: String },
}

fn parse_cmd(line: &str) -> Result<Cmd, AppError> {
    let line = line.trim();
    let mut it = line.split_whitespace();
    let head = it.next().ok_or_else(|| AppError::protocol("empty command"))?;

    match head {
        "GET" => {
            let key = it.next().ok_or_else(|| AppError::protocol("GET requires a key"))?;
            if it.next().is_some() {
                return Err(AppError::protocol("GET takes exactly one argument"));
            }
            Ok(Cmd::Get { key: key.to_string() })
        }
        _ => Err(AppError::protocol(format!("unknown command: {}", head))),
    }
}
```

11.4.4 Step 4: a data loader with contextual IO errors

```
fn load_data(path: &str) -> Result<String, AppError> {
    std::fs::read_to_string(path).map_err(|e| AppError::io("read data", path, e))
}
```

11.4.5 Step 5: wiring everything together

```
fn run(config_path: &str, cmd_line: &str) -> Result<String, AppError> {
    let cfg = load_config(config_path)?;
    let cmd = parse_cmd(cmd_line)?;

    match cmd {
        Cmd::Get { key } => {
            let data = load_data(&cfg.data_path)?;
            for line in data.lines() {
                if let Some((k, v)) = line.split_once('=') {
                    if k.trim() == key {
                        return Ok(v.trim().to_string());
                    }
                }
            }
            Err(AppError::protocol(format!("key not found: {}", key)))
        }
    }
}
```

11.4.6 Step 6: boundary handler (CLI-style)

```
fn main() {
    let config_path = "app.conf";
    let cmd = "GET answer";

    match run(config_path, cmd) {
        Ok(v) => {
            println!("{}", v);
        }
        Err(e) => {
            eprintln!("{}", e);
            let mut src = e.source();
            while let Some(s) = src {
                eprintln!("caused by: {}", s);
                src = s.source();
            }
            std::process::exit(1);
        }
    }
}
```

11.4.7 Lab extension: error codes and classification

Add a method that maps errors to exit codes:

```
impl AppError {
    fn exit_code(&self) -> i32 {
        match self {
            AppError::Config { .. } => 2,
            AppError::Parse { .. } => 3,
            AppError::Protocol { .. } => 4,
            AppError::Io { .. } => 5,
        }
    }
}
```

Then call `exit(e.exit_code())` at the boundary.

11.4.8 What you should gain from this chapter

By the end of this chapter you should be able to:

- use `Result` patterns that keep code clean and correct,
- distinguish application-level error unification from library-level error precision,
- design a unified error type that preserves context and source errors,
- keep error presentation at boundaries while preserving structured internal meaning,
- build a project-wide error system that scales across modules.

Professional error handling is not about fancy types; it is about disciplined contracts, consistent propagation, and reliable boundaries.

Modules, Crates, and Large-Scale Structure

As Rust projects grow, correctness and performance are not enough. You also need a structure that keeps the codebase navigable, enforces boundaries, and supports long-term maintenance. Rust's module system and visibility rules are intentionally designed to make architecture enforceable by the compiler:

- you can hide internals by default,
- expose only stable APIs,
- design clear layers (core, platform, adapters, app),
- and refactor with confidence because boundaries are explicit.

This chapter focuses on the practical mechanics and architectural patterns that appear in real Rust systems:

- organizing code into modules and crates,
- using `pub`, `pub(crate)`, and related visibility tools,
- using re-exports to present clean public APIs,

- and a lab that refactors a single binary project into a reusable library plus an application crate.

12.1 Code organization

12.1.1 Two levels: modules and crates

Rust organizes code at two major levels:

- **Modules** (`mod`): a logical namespace boundary inside a crate.
- **Crates**: a compilation unit and distribution unit (a library crate or a binary crate).

A single Cargo package may contain:

- one library crate (`src/lib.rs`),
- one or more binary crates (`src/main.rs` and/or `src/bin/*.rs`),
- shared modules in `src/`.

12.1.2 The “folder maps to module” mental model

A practical file layout pattern:

```
myproj/
  Cargo.toml
  src/
    lib.rs
    main.rs
    core/
      mod.rs
      parse.rs
      protocol.rs
    util/
      mod.rs
      bytes.rs
```

And in `lib.rs`:

```
pub mod core;
pub mod util;
```

Then in `src/core/mod.rs`:

```
pub mod parse;
pub mod protocol;
```

This scales well because:

- each module becomes a directory of focused submodules,
- deep subsystems remain discoverable,
- public surface is controlled at a single point (`lib.rs`).

12.1.3 Architectural layering pattern

A robust large-scale structure often follows layers:

- **domain/core**: types, invariants, state machines, pure logic,
- **services**: orchestration of domain logic, error boundaries,
- **adapters**: IO boundaries (files, network, DB), serialization,
- **app/bin**: CLI/service main entry points, configuration, wiring.

Rust’s visibility controls help enforce these layers.

12.1.4 Avoiding “module spaghetti”

Common anti-patterns in growing projects:

- every module is pub and everything imports everything,
- types are redefined in multiple places,
- many unrelated things are dumped into `util`,
- internal helper functions become public “just to make it compile”.

Good practices:

- keep most modules private,
- expose a small API at the crate root,
- re-export carefully rather than making internals public.

12.2 `pub`, `pub(crate)`, and visibility

12.2.1 Privacy by default

In Rust:

- items are private by default,
- you must explicitly opt into visibility.

This default supports encapsulation without extra ceremony.

12.2.2 pub: public to the outside world

pub makes an item visible to external crates (depending on the visibility of its module path).

```
pub struct Config {
    pub data_path: String,
}
```

But note: even a pub item is not reachable externally unless its containing modules are also publicly reachable (exported through the module tree).

12.2.3 pub(crate): public inside the crate only

pub(crate) is a powerful large-project tool. It allows internal sharing across modules without committing to a stable public API.

```
pub(crate) fn load_internal_cache() -> Vec<u8> {
    vec![1, 2, 3]
}
```

This supports:

- crate-internal refactoring freedom,
- stable public API with flexible internals,
- clearer separation between “library API” and “internal implementation.”

12.2.4 Other useful visibility scopes

Rust also supports restricted visibility for subtrees:

```
pub(super) fn helper_for_parent() {}

pub(in crate::core) fn visible_only_in_core() {}
```

These are especially useful for:

- enforcing layer boundaries,
- limiting helper leakage,
- keeping certain utilities inside a subsystem.

12.2.5 Visibility in structs and enums

Struct fields are private by default even if the struct is public. This allows you to expose a type while controlling how it can be constructed or mutated.

```
pub struct Token {
    raw: String,
}

impl Token {
    pub fn new(raw: String) -> Result<Self, String> {
        if raw.is_empty() {
            return Err("token cannot be empty".to_string());
        }
        Ok(Self { raw })
    }

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

This pattern encodes invariants:

- callers cannot construct invalid tokens,
- internal representation can change without breaking API.

12.3 Re-exports

12.3.1 Why re-exports exist

Large systems often have many internal modules, but you want users to see a clean, stable public surface.
Re-exports let you:

- present a curated API from the crate root,
- hide internal module layout,
- reorganize internals without breaking users.

12.3.2 Re-exporting types for a clean public API

Suppose you have internal structure:

```
src/
  lib.rs
  core/
    mod.rs
    protocol.rs
```

And core/protocol.rs defines:

```
pub struct Machine { /* ... */ }
pub enum Cmd { /* ... */ }
```

In lib.rs, re-export:

```
pub mod core;

pub use crate::core::protocol::{Machine, Cmd};
```

Now users can write:

```
use mycrate::{Machine, Cmd};
```

Even if you later move `protocol.rs` into a different module, you can keep the re-export stable.

12.3.3 Re-exporting modules as namespaces

Sometimes you want to expose a namespace but still curate internals.

```
pub mod api {
    pub use crate::core::protocol::{Machine, Cmd};
    pub use crate::core::parse::Parser;
}
```

Users import:

```
use mycrate::api::{Machine, Cmd};
```

This makes the public surface discoverable and clean.

12.3.4 Do not re-export everything

A disciplined guideline:

- re-export only what you are willing to support long-term,
- keep experimental or internal pieces `pub(crate)` or private,
- keep internal module paths free to change.

12.4 Lab: refactoring a project into library + application

This lab takes a typical early Rust project (single `main.rs`) and refactors it into:

- a reusable library crate that contains core logic,
- a binary application crate that wires inputs/outputs.

The goal is to practice:

- module layout,
- visibility control (`pub` vs `pub(crate)`),
- re-exports for a clean API,
- and architectural separation of concerns.

12.4.1 Starting point (single binary)

```
myproj/
  Cargo.toml
  src/
    main.rs
```

12.4.2 Target structure

```
myproj/
  Cargo.toml
  src/
    lib.rs
    main.rs
    core/
      mod.rs
      protocol.rs
      parse.rs
    error.rs
```

12.4.3 Step 1: create `src/lib.rs`

```
pub mod core;
mod error;

pub use crate::core::protocol::{Machine, Cmd};
pub use crate::error::AppError;
```

Here:

- `core` is public (but you could keep it private and re-export only selected items),
- `error` is internal, but `AppError` is re-exported.

12.4.4 Step 2: define an internal error module

```
use std::fmt;

#[derive(Debug)]
pub enum AppError {
    Parse { msg: String },
    Protocol { msg: String },
}

impl fmt::Display for AppError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            AppError::Parse { msg } => write!(f, ”parse error: {}”, msg),
            AppError::Protocol { msg } => write!(f, ”protocol error: {}”, msg),
        }
    }
}

impl std::error::Error for AppError {}
```

12.4.5 Step 3: split core logic into modules

src/core/mod.rs:

```
pub mod parse;
pub mod protocol;
```

src/core/parse.rs:

```
use crate::AppError;

#[derive(Debug, Clone, PartialEq, Eq)]
pub enum Cmd {
    Ping,
    Quit,
}

pub fn parse_cmd(line: &str) -> Result<Cmd, AppError> {
    let line = line.trim();
    match line {
        "PING" => Ok(Cmd::Ping),
        "QUIT" => Ok(Cmd::Quit),
        _ => Err(AppError::Parse { msg: format!("unknown command: {}", line) }),
    }
}
```

src/core/protocol.rs:

```
use crate::{AppError};
use crate::core::parse::Cmd;

#[derive(Debug, Copy, Clone, PartialEq, Eq)]
enum State {
    Active,
    Closed,
}

pub struct Machine {
    state: State,
}

impl Machine {
    pub fn new() -> Self {
        Self { state: State::Active }
    }

    pub fn handle(&mut self, cmd: Cmd) -> Result<&'static str, AppError> {
        match (self.state, cmd) {
            (State::Active, Cmd::Ping) => Ok("PONG"),
            (State::Active, Cmd::Quit) => {
                self.state = State::Closed;
                Ok("BYE")
            }
           (State::Closed, _) => Err(AppError::Protocol { msg: "session closed".to_string()
            ↪  }),
        }
    }
}
```

Note the design:

- `State` is private: callers cannot forge states.

- Machine is public: it is a stable API entry point.
- errors are unified through AppError.

12.4.6 Step 4: keep the binary thin (src/main.rs)

```
use myproj::{Machine, Cmd};
use myproj::core::parse::parse_cmd;

fn main() {
    let mut m = Machine::new();

    for line in ["PING", "QUIT", "PING"] {
        let cmd = match parse_cmd(line) {
            Ok(c) => c,
            Err(e) => {
                eprintln!("{}", e);
                continue;
            }
        };

        match m.handle(cmd) {
            Ok(resp) => println!("{}", resp),
            Err(e) => eprintln!("{}", e),
        }
    }
}
```

12.4.7 Step 5: tighten visibility with pub(crate)

If core is an implementation detail, hide it:

```
mod core;        /* private */
```

```rust
mod error;

pub use crate::core::protocol::Machine;
pub use crate::core::parse::Cmd;
pub use crate::error::AppError;
```

Then inside the crate, modules can still share via `pub(crate)` functions if needed, without exposing them publicly.

12.4.8 Step 6: public API curation through re-exports

A clean library often exposes only a few top-level symbols:

- primary types,
- primary functions,
- error type(s),
- and a small prelude module if appropriate.

Example prelude:

```rust
pub mod prelude {
    pub use crate::{Machine, Cmd, AppError};
}
```

12.4.9 What you should gain from this lab

After completing the lab, you should be able to:

- split a Rust project into modules with clear responsibilities,
- enforce boundaries using visibility controls,

- design crate-level public APIs with re-exports,
- keep binaries thin and focused on wiring,
- preserve refactoring freedom by keeping most internals non-public.

Large-scale Rust structure is not about file layout aesthetics. It is about using the language's privacy model to enforce architecture, reduce coupling, and keep systems maintainable for years.

Part IV

Performance and C/C++-Level Control

Performance Without Myths

High performance in Rust is not achieved by “clever tricks” or by fighting the language. It comes from understanding what the compiler does, how code maps to machine behavior, and how to design APIs that avoid unnecessary work: allocations, copies, dynamic dispatch, and cache-unfriendly layouts.

This chapter focuses on performance foundations that matter in real systems:

- how inlining and monomorphization shape code generation,
- how allocation strategy affects throughput and latency,
- how borrowing and slices eliminate copies without sacrificing safety,
- and a step-by-step lab that demonstrates performance tuning as a disciplined workflow.

13.1 Inlining and monomorphization

13.1.1 What monomorphization really means

Rust’s generic code is typically compiled using **monomorphization**:

- each generic function is instantiated for concrete types that use it,
- those instantiations are compiled as if you wrote type-specific versions by hand,

- this enables inlining and optimization across abstraction boundaries.

A generic function:

```
fn add_one<T>(x: T) -> T
where
    T: std::ops::Add<Output = T> + From<u8>,
{
    x + T::from(1)
}

fn main() {
    let a = add_one(10_i32);
    let b = add_one(10_u64);
    println!("{}, {}", a, b);
}
```

Conceptually, the compiler produces two specialized versions:

- one for `i32`,
- one for `u64`.

This often yields “C-like” performance because there is no runtime type dispatch.

13.1.2 Inlining: removing abstraction overhead

Inlining replaces a function call with the function body at the call site. This can:

- eliminate call overhead,
- enable constant propagation,
- enable dead-code elimination across boundaries,

- improve vectorization opportunities in hot loops.

Simple helper:

```rust
#[inline]
fn clamp_u8(x: i32) -> u8 {
    if x < 0 { 0 } else if x > 255 { 255 } else { x as u8 }
}

fn main() {
    let xs = [-10, 10, 500];
    let ys: Vec<u8> = xs.iter().map(|&x| clamp_u8(x)).collect();
    println!("{:?}", ys);
}
```

The important point: in Rust, small generic or non-generic helpers are often inlined automatically under optimization. The `#[inline]` attribute is a hint, not a command, but it can influence cross-crate inlining decisions.

13.1.3 Cross-crate inlining and API design

When performance matters across crate boundaries:

- keep small hot functions simple,
- avoid forcing dynamic dispatch when static dispatch would do,
- prefer generic wrappers for tight loops,
- re-export stable APIs while keeping internals refactorable.

13.1.4 Monomorphization trade-offs

Monomorphization can increase code size because multiple instantiations are generated. This can matter for:

- embedded targets,
- instruction-cache pressure,
- build times.

Professional tuning balances:

- static dispatch (fast, larger code),
- dynamic dispatch (smaller code, runtime indirection),
- and careful generic design (avoid exploding instantiations).

13.2 Memory allocation strategies (conceptual)

13.2.1 Allocation costs are real

Heap allocation can dominate performance in real systems:

- allocation and deallocation overhead,
- allocator contention in multi-threaded programs,
- fragmentation and cache effects,
- unpredictable latency spikes in hot paths.

Rust does not hide allocation; it makes it explicit. But you must design to control it.

13.2.2 Core allocation tools

The common allocating types:

- `Vec<T>`, `String` for growable buffers,
- `Box<T>` for single heap allocation,
- `Rc<T>` and `Arc<T>` for shared ownership with refcounts,
- `HashMap<K, V>` for dynamic associative storage.

A professional approach is to:

- pre-allocate when you can (`with_capacity`),
- reuse buffers in loops,
- avoid intermediate allocations in transformations,
- use slices and iterators to process borrowed views.

13.2.3 Capacity planning and amortized growth

A `Vec` grows geometrically (implementation-defined strategy), which means pushes are amortized O(1) but can cause reallocations and copies when growth occurs.

```
fn build(n: usize) -> Vec<u64> {
    let mut v = Vec::with_capacity(n);
    for i in 0..n as u64 {
        v.push(i);
    }
    v
}
```

13.2.4 Arena-style allocation (conceptual)

For workloads that create many small objects with the same lifetime (e.g., AST nodes, request-scoped objects), an arena/bump allocation strategy can reduce overhead:

- allocate many objects in a big chunk,
- free them all at once when the arena drops,
- avoid per-object deallocation cost.

Even when you do not implement a custom arena, you can mimic the lifetime grouping idea:

- collect objects in a `Vec`,
- reuse the vector between iterations,
- clear without deallocating capacity.

```rust
fn reuse_buffer(iters: usize) -> usize {
    let mut buf: Vec<u64> = Vec::with_capacity(1024);
    let mut total = 0;

    for k in 0..iters {
        buf.clear(); /* keeps capacity */
        for i in 0..1000 {
            buf.push((k as u64) ^ (i as u64));
        }
        total += buf.len();
    }

    total
}
```

13.2.5 Allocations are sometimes the right choice

Avoiding allocation blindly is a myth. The correct rule:

- allocate when it simplifies design and the cost is not dominant,
- avoid allocation in hot loops or high-throughput paths,
- measure before optimizing.

13.3 Eliminating copies with borrowing and slices

13.3.1 Borrowing is a performance tool

Borrowing is not only for safety. It is often the simplest way to remove copies:

- take `&str` instead of `String`,
- take `&[T]` instead of `Vec<T>`,
- return references when the data already exists in the caller.

13.3.2 `&str` vs `String`: allocation-free APIs

Bad API for performance (forces allocation):

```
fn count_words_bad(s: String) -> usize {
    s.split_whitespace().count()
}
```

Better API (borrowed input, no allocation):

```
fn count_words(s: &str) -> usize {
    s.split_whitespace().count()
}
```

13.3.3 Slices for zero-copy processing

A slice `&[T]` is a borrowed view into contiguous memory. It lets you process data without copying.

```rust
fn sum(xs: &[i32]) -> i32 {
    xs.iter().copied().sum()
}

fn main() {
    let v = vec![1, 2, 3];
    println!("{}", sum(&v));
}
```

13.3.4 Splitting without copying

Many operations return borrowed slices.

```rust
fn parse_csv_line(line: &str) -> Vec<&str> {
    line.split(',').map(|s| s.trim()).collect()
}

fn main() {
    let parts = parse_csv_line("a, b, c");
    println!("{:?}", parts);
}
```

This returns `Vec<&str>` referencing the original `line`. No substring allocations occur.

13.3.5 Avoiding clone in transformations

A typical C/C++ “thinking trap” is to clone or allocate “just to satisfy the compiler.” The Rust way is to design the function boundary so it borrows.

Example: searching in a collection without cloning keys:

```
use std::collections::HashMap;

fn get_value<'a>(m: &'a HashMap<String, i32>, k: &str) -> Option<&'a i32> {
    m.get(k)
}
```

The key idea:

- accept `&str` rather than forcing `String`,
- return `Option<&i32>` rather than copying the value.

13.4 Lab: step-by-step performance tuning

This lab demonstrates performance tuning as a workflow. We will start with a correct but suboptimal implementation, then systematically remove bottlenecks:

- eliminate repeated allocations,
- eliminate unnecessary copies,
- choose better API boundaries,
- and measure at each step.

The example task: compute the total length of all tokens across many input lines.

13.4.1 Version 0: correct but allocation-heavy

This version allocates `String` tokens repeatedly.

```
fn total_token_len_v0(lines: &[String]) -> usize {
    let mut total = 0;
    for line in lines {
        let tokens: Vec<String> = line
            .split_whitespace()
            .map(|s| s.to_string())
            .collect();
        for t in tokens {
            total += t.len();
        }
    }
    total
}
```

Problems:

- allocates a `Vec<String>` per line,
- allocates a `String` per token.

13.4.2 Version 1: zero-copy tokens using `&str`

Return borrowed token slices rather than allocating `String`s.

```
fn total_token_len_v1(lines: &[String]) -> usize {
    let mut total = 0;
    for line in lines {
        for t in line.split_whitespace() {
            total += t.len();
        }
    }
    total
}
```

This eliminates all token allocations.

13.4.3 Version 2: accept &str inputs (better API boundary)

If callers already have &str lines (e.g., from reading a file buffer), accept them directly:

```
fn total_token_len_v2(lines: &[&str]) -> usize {
    let mut total = 0;
    for line in lines {
        for t in line.split_whitespace() {
            total += t.len();
        }
    }
    total
}
```

This prevents forcing callers into String ownership.

13.4.4 Version 3: reuse buffers when you must allocate

Sometimes you must allocate (e.g., building normalized tokens). In that case, reuse buffers.

```
fn normalize_to_lower(buf: &mut String, s: &str) {
    buf.clear();
    buf.push_str(s);
    buf.make_ascii_lowercase();
}

fn total_token_len_v3(lines: &[&str]) -> usize {
    let mut total = 0;
    let mut tmp = String::with_capacity(64);

    for line in lines {
        for t in line.split_whitespace() {
            normalize_to_lower(&mut tmp, t);
            total += tmp.len();
        }
    }
    total
}
```

This allocates once and reuses.

13.4.5 Step-by-step measurement (conceptual)

A disciplined tuning loop:

- establish a baseline,
- change one thing,
- measure again,
- keep improvements, revert regressions.

13.4.6 Optional: demonstrate static vs dynamic dispatch cost

If you implement token processing via a trait object, you may introduce vtable overhead. Compare designs:

- generic function parameter (`F: FnMut(&str)`): static dispatch, inlining possible,
- `&mut dyn FnMut(&str)`: dynamic dispatch, flexible but indirect calls.

Static dispatch style:

```
fn for_each_token<F>(lines: &[&str], mut f: F)
where
    F: FnMut(&str),
{
    for line in lines {
        for t in line.split_whitespace() {
            f(t);
        }
    }
}
```

Dynamic dispatch style:

```
fn for_each_token_dyn(lines: &[&str], f: &mut dyn FnMut(&str)) {
    for line in lines {
        for t in line.split_whitespace() {
            f(t);
        }
    }
}
```

Both are correct. Choose based on:

- performance sensitivity,
- architectural flexibility needs,
- code size and compile-time trade-offs.

13.4.7 What you should learn from this lab

The key lesson: performance is usually won by eliminating unnecessary work:

- avoid allocations in hot paths,
- avoid copies by borrowing and slicing,
- design APIs that accept borrowed inputs,

- reuse buffers and capacity when allocation is needed,
- measure improvements systematically.

Rust does not magically make code fast. It makes the cost model explicit and gives you strong tools to reach C/C++-level performance without sacrificing safety.

Iterators and Zero-Cost Functional Patterns

Rust iterators are not "slow functional sugar." They are a carefully engineered abstraction that often compiles down to tight loops with predictable performance. The goal is to write code that is:

- expressive and composable,
- allocation-free by default,
- friendly to compiler optimization (inlining, vectorization, dead-code elimination),
- and still transparent about costs when you opt into them.

This chapter builds a practical performance intuition:

- iterator adapters as a vocabulary for data pipelines,
- closures and capture rules so you know what is moved, borrowed, or copied,
- when a `for` loop is better (and when iterators are better),
- and a lab that builds a high-performance data-processing pipeline step by step.

14.1 Iterator adapters

14.1.1 The iterator mental model

An iterator is a state machine that produces values one-by-one through `next()`. Adapters transform iterators into other iterators without allocating intermediate collections.
The power comes from two properties:

- adapters are lazy (they do not do work until consumed),
- adapters compose (pipelines are built by chaining).

14.1.2 Core adapters you must master

A professional iterator vocabulary includes:

- `map`, `filter`, `filter_map`,
- `take`, `skip`, `take_while`, `skip_while`,
- `enumerate`,
- `zip`,
- `chain`,
- `flat_map`,
- `cloned` / `copied`,
- `inspect` for debugging,
- `peekable` for lookahead,
- `fold`, `try_fold` for reduction.

14.1.3 Example: map + filter + sum (no allocations)

```
fn sum_even_squares(xs: &[i32]) -> i32 {
    xs.iter()
      .copied()
      .filter(|x| x % 2 == 0)
      .map(|x| x * x)
      .sum()
}
```

14.1.4 Example: filter_map for parsing and selection

```
fn sum_parsed_positive(nums: &[&str]) -> i64 {
    nums.iter()
        .filter_map(|s| s.parse::<i64>().ok())
        .filter(|&x| x > 0)
        .sum()
}
```

14.1.5 Example: zip for structured parallel traversal

```
fn dot(a: &[f32], b: &[f32]) -> f32 {
    a.iter().zip(b.iter()).map(|(&x, &y)| x * y).sum()
}
```

14.1.6 Example: flat_map for flattening nested sources

```
fn total_words(lines: &[&str]) -> usize {
    lines.iter()
         .flat_map(|l| l.split_whitespace())
         .count()
}
```

14.1.7 Adapter composition without intermediate Vec

A common performance mistake is:

- collect into a `Vec` after each step.

Instead, keep a single pipeline and collect once at the end (or not at all).

```rust
fn normalize_and_collect(lines: &[&str]) -> Vec<String> {
    lines.iter()
         .flat_map(|l| l.split_whitespace())
         .map(|w| w.to_ascii_lowercase())
         .filter(|w| w.len() >= 3)
         .collect()
}
```

This allocates only for the final `String`s you explicitly create; it does not allocate intermediate vectors.

14.1.8 Consuming adapters vs producing adapters

Producing adapters return iterators (`map`, `filter`, `zip`). **Consuming adapters** end the pipeline and produce a final result (`collect`, `sum`, `count`, `fold`).
The tuning rule:

- stay in producing adapters as long as possible,
- consume only at the end or at deliberate boundaries.

14.2 Closures and capture rules

14.2.1 Closures are typed objects, not magic

A closure is not a function pointer. It is a value with a compiler-generated anonymous type that may store captured variables as fields.
This matters because capture affects:

- ownership (move vs borrow),
- lifetime requirements,
- whether the closure can be called multiple times,
- whether the closure is thread-safe.

14.2.2 The three closure traits

Closures implement one or more of these traits depending on capture behavior:

- `Fn`: can be called repeatedly, does not require mutable access to captured environment,
- `FnMut`: may mutate captured environment, requires `&mut self`,
- `FnOnce`: may consume captured values, can be called at most once.

14.2.3 Capture by borrow (default when possible)

```
fn main() {
    let factor = 10;
    let xs = [1, 2, 3];

    let ys: Vec<i32> = xs.iter().map(|&x| x * factor).collect();
    println!("{:?}", ys);
}
```

Here, `factor` is captured by shared borrow if possible.

14.2.4 Capture by mutable borrow

```
fn main() {
    let mut count = 0;
    let xs = [1, 2, 3];

    xs.iter().for_each(|_| {
        count += 1;
    });

    println!("{}", count);
}
```

This closure requires `FnMut` because it mutates `count`.

14.2.5 Capture by move

Use `move` when:

- you want the closure to own captured values,
- you need it to outlive the current stack frame,
- you want to pass it into a thread.

```
use std::thread;

fn main() {
    let msg = String::from("hello");
    let h = thread::spawn(move || {
        println!("{}", msg);
    });

    h.join().unwrap();
}
```

14.2.6 Avoiding accidental allocations in closures

A classic performance trap:

- calling `to_string()` or building `String` in a tight `map`.

Instead:

- process borrowed `&str` as long as possible,

- allocate only at the final boundary.

```
fn total_len(lines: &[&str]) -> usize {
    lines.iter()
        .flat_map(|l| l.split_whitespace())
        .map(|w| w.len())
        .sum()
}
```

14.3 Loops vs iterators comparison

14.3.1 The truth: both can be optimal

In optimized builds, iterator pipelines often compile into code comparable to hand-written loops. But **architecture and clarity** matter more than ideology.

14.3.2 When loops are better

Loops may be better when:

- you need early exits with complex control flow,
- you need indexed mutation and multiple mutable borrows,
- you want to minimize iterator adapter layering for readability in a specific hot loop,
- you need to fuse multiple passes manually with careful branch control.

14.3.3 When iterators are better

Iterators may be better when:

- you want composable pipelines without intermediate allocations,
- you want to express dataflow clearly (map/filter/fold),
- you want fewer mutable states in the loop body,
- you want to leverage specialized adapters (`zip`, `chunks`, `windows`).

14.3.4 Same computation: loop vs iterator

Task: sum of positive numbers.

Loop version:

```rust
fn sum_pos_loop(xs: &[i32]) -> i32 {
    let mut s = 0;
    for &x in xs {
        if x > 0 {
            s += x;
        }
    }
    s
}
```

Iterator version:

```
fn sum_pos_iter(xs: &[i32]) -> i32 {
    xs.iter().copied().filter(|&x| x > 0).sum()
}
```

Both are fine. The iterator version is shorter and often just as fast. The loop version is sometimes easier to tune when the logic grows.

14.3.5 Mutation-heavy patterns: loop can be clearer

Example: in-place transform.

```
fn clamp_in_place(xs: &mut [i32]) {
    for x in xs {
        if *x < 0 { *x = 0; }
        if *x > 255 { *x = 255; }
    }
}
```

Iterator-style in-place mutation exists (e.g., `iter_mut().for_each(...)`) but loops are often clearer for this pattern.

14.4 Lab: high-performance data processing

This lab builds a high-performance log-like processor. Input is a slice of lines; each line is either:

- `OK <value>`

- `ERR <code>`

Goal:

- sum all OK values,
- count error codes,
- avoid allocations in the hot path,
- and structure the pipeline for readability and performance.

14.4.1 Step 1: define a borrowed record view

Instead of allocating parsed strings, parse into borrowed views.

```rust
#[derive(Debug, Copy, Clone, PartialEq, Eq)]
enum Kind {
    Ok,
    Err,
}

#[derive(Debug, Clone, PartialEq, Eq)]
struct Record<'a> {
    kind: Kind,
    payload: &'a str,
}

fn parse_line<'a>(line: &'a str) -> Option<Record<'a>> {
    let line = line.trim();
    let mut it = line.split_whitespace();
    let head = it.next()?;
    let payload = it.next()?;
    if it.next().is_some() {
        return None;
    }

    match head {
        "OK" => Some(Record { kind: Kind::Ok, payload }),
        "ERR" => Some(Record { kind: Kind::Err, payload }),
        _ => None,
    }
}
```

This returns `Option<Record>` because malformed lines are expected and can be ignored or counted.

14.4.2 Step 2: iterator pipeline for parsing and aggregation

```
use std::collections::HashMap;

fn process(lines: &[&str]) -> (i64, HashMap<i32, usize>, usize) {
    let mut ok_sum: i64 = 0;
    let mut err_counts: HashMap<i32, usize> = HashMap::new();
    let mut bad_lines: usize = 0;

    for rec in lines.iter().filter_map(|l| parse_line(l)) {
        match rec.kind {
            Kind::Ok => {
                if let Ok(v) = rec.payload.parse::<i64>() {
                    ok_sum += v;
                } else {
                    bad_lines += 1;
                }
            }
            Kind::Err => {
                if let Ok(code) = rec.payload.parse::<i32>() {
                    *err_counts.entry(code).or_insert(0) += 1;
                } else {
                    bad_lines += 1;
                }
            }
        }
    }

    (ok_sum, err_counts, bad_lines)
}
```

This approach:

- does not allocate during parsing,
- processes each line once,
- uses iterator adapters only for the parsing filter step.

14.4.3 Step 3: fully functional-style fold

If you prefer a pipeline that stays in iterator style, use `fold`. This can be clean, but be careful about readability.

```
use std::collections::HashMap;

#[derive(Default)]
struct Acc {
    ok_sum: i64,
    err_counts: HashMap<i32, usize>,
    bad_lines: usize,
}
```

```
fn process_fold(lines: &[&str]) -> Acc {
    lines.iter()
        .filter_map(|l| parse_line(l))
        .fold(Acc::default(), |mut acc, rec| {
            match rec.kind {
                Kind::Ok => match rec.payload.parse::<i64>() {
                    Ok(v) => acc.ok_sum += v,
                    Err(_) => acc.bad_lines += 1,
                },
                Kind::Err => match rec.payload.parse::<i32>() {
                    Ok(code) => *acc.err_counts.entry(code).or_insert(0) += 1,
                    Err(_) => acc.bad_lines += 1,
                },
            }
            acc
        })
}
```

14.4.4 Step 4: micro-tuning ideas (conceptual)

For extreme throughput:

- avoid `HashMap` if error code domain is small: use `Vec<usize>` indexed by code,
- reduce parsing overhead by using custom numeric parsing if profiles show `parse()` dominates,
- avoid repeated trimming if input format is already normalized,
- use `lines()` on a single buffer rather than allocating `Vec<String>`.

14.4.5 Step 5: tests

```
#[cfg(test)]
```

```rust
mod tests {
    use super::*;

    #[test]
    fn parsing() {
        assert!(parse_line("OK 10").is_some());
        assert!(parse_line("ERR 5").is_some());
        assert!(parse_line("BAD 10").is_none());
        assert!(parse_line("OK").is_none());
        assert!(parse_line("OK 1 2").is_none());
    }

    #[test]
    fn processing() {
        let lines = ["OK 10", "OK x", "ERR 2", "ERR 2", "ERR y", "BAD 1"];
        let (sum, map, bad) = process(&lines);

        assert_eq!(sum, 10);
        assert_eq!(*map.get(&2).unwrap(), 2);
        assert_eq!(bad, 2);
    }
}
```

14.4.6 What you should learn from this lab

Key takeaways:

- iterator adapters help you build pipelines without intermediate allocations,
- closures capture rules affect ownership and performance; know when you borrow and when you move,
- loops and iterators are tools; choose based on clarity and control flow,

- the highest gains come from eliminating allocations and copies and from keeping data borrowed as long as possible.

Rust’s iterator model is one of the clearest examples of “zero-cost functional style” done in systems programming: expressive code that still maps to predictable, efficient machine behavior.

Practical Memory Management

Rust makes memory management explicit in a disciplined way:

- ownership defines who is responsible for freeing memory,
- borrowing defines who can view or mutate memory without owning it,
- and standard containers (`String`, `Vec`) expose their allocation behavior so you can design for throughput and latency.

This chapter is practical and performance-oriented. You will learn:

- how to choose between `String` and `&str`,
- how `Vec` growth affects cost and how to control it,
- how slices define ownership boundaries and eliminate copies,
- and how to parse large text efficiently with minimal allocation.

15.1 String vs &str

15.1.1 The key distinction

`String` is an **owned**, growable UTF-8 buffer allocated on the heap. `&str` is a **borrowed** view into UTF-8 text (a slice of bytes known to be valid UTF-8).

Practical consequence:

- use &str for reading/processing without ownership transfer,
- use String when you must own, store, or modify text.

15.1.2 API design: accept &str, return String only when necessary

Bad API (forces allocation by the caller):

```rust
fn normalize_bad(s: String) -> String {
    s.trim().to_lowercase()
}
```

Better API:

```rust
fn normalize(s: &str) -> String {
    s.trim().to_lowercase()
}
```

Best API when you can avoid allocation (return a borrowed slice):

```rust
fn trimmed(s: &str) -> &str {
    s.trim()
}
```

15.1.3 When returning &str is possible and when it is not

You can return &str when the output is a subslice of the input.

```rust
fn after_colon(s: &str) -> Option<&str> {
    let (_, rhs) = s.split_once(':')?;
    Some(rhs.trim())
}
```

You must return String when you construct new text:

```
fn replace_spaces(s: &str) -> String {
    s.replace(' ', "_")
}
```

15.1.4 UTF-8 reality: indexing costs

A String is UTF-8, so indexing by integer is not constant-time in terms of characters. Rust avoids unsafe indexing semantics by design.
Iterate as:

- bytes (as_bytes()),
- chars (chars()),
- or graphemes (requires higher-level logic).

Example: byte scanning for ASCII-like formats:

```
fn count_commas_ascii(s: &str) -> usize {
    s.as_bytes().iter().filter(|&&b| b == b',').count()
}
```

15.1.5 Borrowed text in collections: beware lifetimes

A Vec<&str> is valid only as long as the original buffer lives.

```
fn split_words<'a>(s: &'a str) -> Vec<&'a str> {
    s.split_whitespace().collect()
}
```

Professional guideline:

- use Vec<&str> when the source buffer stays alive for the needed scope,

- use `Vec<String>` when you must store independent owned strings beyond the input buffer lifetime.

15.2 Vec growth strategy

15.2.1 Why growth strategy matters

`Vec<T>` is the core dynamic array. Its performance is shaped by:

- capacity growth and reallocations,
- copying/moving elements during reallocation,
- allocation behavior under contention,
- memory locality and cache behavior.

15.2.2 Length vs capacity

```rust
fn main() {
    let mut v: Vec<i32> = Vec::new();
    println!("len={}, cap={}", v.len(), v.capacity());

    v.push(1);
    println!("len={}, cap={}", v.len(), v.capacity());
}
```

`len` is the number of elements. `capacity` is the reserved space (how many elements can be pushed before reallocation).

15.2.3 Amortized growth and reallocation cost

A `Vec` grows geometrically (exact policy is an implementation detail), meaning:

- `push` is amortized O(1),
- but some pushes trigger reallocation,
- reallocation may move/copy all elements to a new region.

This cost matters in hot loops.

15.2.4 Controlling growth with `with_capacity` and `reserve`

If you can estimate size, pre-allocate:

```rust
fn build(n: usize) -> Vec<u64> {
    let mut v = Vec::with_capacity(n);
    for i in 0..n as u64 {
        v.push(i);
    }
    v
}
```

If size grows gradually, reserve in chunks:

```rust
fn push_many(mut v: Vec<u8>, input: &[u8]) -> Vec<u8> {
    v.reserve(input.len());
    v.extend_from_slice(input);
    v
}
```

15.2.5 Avoiding repeated allocations by reuse

If you repeatedly fill a vector in a loop, reuse capacity:

```
fn process_batches(batches: &[&[u32]]) -> u64 {
    let mut buf: Vec<u32> = Vec::with_capacity(1024);
    let mut total: u64 = 0;

    for b in batches {
        buf.clear();          /* keeps capacity */
        buf.extend_from_slice(b);
        total += buf.len() as u64;
    }

    total
}
```

This is a practical, allocator-friendly pattern.

15.3 Slices and ownership boundaries

15.3.1 Slices are borrowed views

A slice is a view into contiguous memory:

- `&[T]`: shared borrowed view,
- `&mut [T]`: exclusive borrowed view,
- `&str`: a UTF-8 string slice (conceptually `&[u8]` with UTF-8 validity).

Slices are the primary tool for defining ownership boundaries without copying.

15.3.2 Design rule: accept slices in APIs

Instead of accepting `Vec<T>` (which implies ownership), accept `&[T]`:

```rust
fn sum(xs: &[i32]) -> i32 {
    xs.iter().copied().sum()
}
```

This allows callers to pass:

- a vector,
- an array,
- a subslice,
- or a static buffer,

without allocation or ownership transfer.

15.3.3 Returning slices to avoid copies

If your function can return a view, do it:

```rust
fn head<T>(xs: &[T], n: usize) -> &[T] {
    let n = n.min(xs.len());
    &xs[..n]
}
```

15.3.4 Ownership boundary patterns

Common professional patterns:

- parse from `&[u8]` and produce borrowed `&[u8]` slices into the same buffer,
- parse from `&str` and produce `&str` subslices,
- allocate only at the boundary where you must store results beyond input lifetime.

15.3.5 Mutation boundaries with &mut [T]

When you need in-place modification, accept `&mut [T]`:

```
fn clamp_u8_in_place(xs: &mut [i32]) {
    for x in xs {
        if *x < 0 { *x = 0; }
        if *x > 255 { *x = 255; }
    }
}
```

This enforces exclusive access and prevents aliasing bugs.

15.4 Lab: parsing large text efficiently

This lab builds a high-throughput parser for a simple large text format: each line is `key=value`. We want to:

- parse without allocating per key/value,
- store results either as borrowed views (fastest) or owned (if needed),
- handle malformed lines robustly,
- scale to large inputs.

15.4.1 Step 1: parse into borrowed slices

We take input as one large `&str` buffer, then return pairs of `&str` slices referencing it.

```
fn parse_kv_lines<'a>(text: &'a str) -> Vec<(&'a str, &'a str)> {
    let mut out: Vec<(&'a str, &'a str)> = Vec::new();
    for raw in text.lines() {
        let line = raw.trim();
        if line.is_empty() || line.starts_with('#') {
            continue;
        }

        let Some((k, v)) = line.split_once('=') else {
            continue;
        };

        let k = k.trim();
        let v = v.trim();
        if k.is_empty() || v.is_empty() {
            continue;
        }

        out.push((k, v));
    }

    out
}
```

This creates a vector of references only; it does not allocate strings for keys/values.

15.4.2 Step 2: pre-allocate result capacity (simple heuristic)

If you expect many lines, reserve to reduce reallocations. A common heuristic is: estimate based on newline count.

```
fn parse_kv_lines_reserved<'a>(text: &'a str) -> Vec<(&'a str, &'a str)> {
    let approx = text.as_bytes().iter().filter(|&&b| b == b'\n').count() + 1;
    let mut out: Vec<(&'a str, &'a str)> = Vec::with_capacity(approx / 2);
    for raw in text.lines() {
        let line = raw.trim();
        if line.is_empty() || line.starts_with('#') {
            continue;
        }

        let Some((k, v)) = line.split_once('=') else {
            continue;
        };

        let k = k.trim();
        let v = v.trim();
        if k.is_empty() || v.is_empty() {
            continue;
        }

        out.push((k, v));
    }

    out
}
```

15.4.3 Step 3: when you must own results

If you need to store the pairs beyond the input buffer lifetime, you must allocate owned strings.

```
fn parse_kv_owned(text: &str) -> Vec<(String, String)> {
    parse_kv_lines(text)
        .into_iter()
        .map(|(k, v)| (k.to_string(), v.to_string()))
        .collect()
}
```

Professional rule:

- keep things borrowed as long as possible,
- allocate at the boundary where ownership is truly needed.

15.4.4 Step 4: building a fast lookup without cloning keys

If you want lookup, build a map. If you can keep the input buffer alive, you can store `&str` keys and values.

```rust
use std::collections::HashMap;

fn parse_to_map<'a>(text: &'a str) -> HashMap<&'a str, &'a str> {
    let mut m: HashMap<&'a str, &'a str> = HashMap::new();
    for (k, v) in parse_kv_lines_reserved(text) {
        m.insert(k, v);
    }
    m
}
```

This avoids allocating keys and values, but the map is tied to the lifetime of `text`.

15.4.5 Step 5: avoiding intermediate vectors

You can build the map directly while scanning:

```rust
use std::collections::HashMap;

fn parse_to_map_direct<'a>(text: &'a str) -> HashMap<&'a str, &'a str> {
    let approx = text.as_bytes().iter().filter(|&&b| b == b'\n').count() + 1;
    let mut m: HashMap<&'a str, &'a str> = HashMap::with_capacity(approx / 2);

    for raw in text.lines() {
        let line = raw.trim();
        if line.is_empty() || line.starts_with('#') {
            continue;
        }

        let Some((k, v)) = line.split_once('=') else {
            continue;
        };

        let k = k.trim();
        let v = v.trim();
        if k.is_empty() || v.is_empty() {
            continue;
        }

        m.insert(k, v);
    }

    m
}
```

15.4.6 Step 6: tests

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

    #[test]
    fn borrowed_parse() {
        let txt = "\
# comment
a = 1
b=2
badline
c = 3
”;
        let kv = parse_kv_lines(txt);
        assert_eq!(kv.len(), 3);
        assert_eq!(kv[0], (”a”, ”1”));
        assert_eq!(kv[1], (”b”, ”2”));
        assert_eq!(kv[2], (”c”, ”3”));
    }

    #[test]
    fn map_parse() {
        let txt = ”x=10\ny=20\n”;
        let m = parse_to_map_direct(txt);
        assert_eq!(m.get(”x”), Some(&”10”));
        assert_eq!(m.get(”y”), Some(&”20”));
    }

    #[test]
    fn owned_parse() {
        let txt = ”k=v\n”;
        let kv = parse_kv_owned(txt);
        assert_eq!(kv[0].0, ”k”.to_string());
        assert_eq!(kv[0].1, ”v”.to_string());
    }
}
```

15.4.7 Performance checklist from this chapter

When parsing large text efficiently:

- keep input as one buffer and produce borrowed slices into it,
- avoid allocating per token unless necessary,

- reserve capacity for vectors/maps when you can estimate sizes,
- reuse buffers if you must allocate in a loop,
- make ownership boundaries explicit: borrow internally, own at the boundary.

Rust's memory tools are not complicated when approached correctly: they are explicit, composable, and performance-friendly by default.

Unsafe Rust — Controlled Power

Unsafe Rust is not “turning safety off.” It is Rust’s explicit mechanism for writing low-level code that the compiler cannot fully verify, while keeping the rest of the program within the strong guarantees of safe Rust.

A professional Rust codebase treats `unsafe` as:

- small, isolated, reviewed,
- justified by concrete needs (FFI, performance, hardware, custom layouts),
- documented with explicit invariants,
- and wrapped behind safe APIs that prevent misuse.

This chapter teaches unsafe Rust as controlled engineering:

- when unsafe is justified,
- the five core safety rules you must enforce,
- raw pointers and aliasing realities,
- how to build safe abstractions around unsafe internals,
- and a lab that implements a minimal arena allocator (and an optional Vec-like design outline).

16.1 When unsafe is justified

You use `unsafe` only when safe Rust cannot express a requirement without unacceptable cost or cannot express it at all. Common legitimate cases:

16.1.1 FFI and platform boundaries

Calling C APIs, OS syscalls, and interfacing with existing system libraries often requires:

- raw pointers,
- manual memory ownership contracts,
- C layout compatibility.

16.1.2 Performance-critical internals

Sometimes you need:

- manual bounds-check elimination (with proof),
- custom data layouts and packed representations,
- uninitialized memory for efficient construction,
- specialized algorithms that require pointer arithmetic.

16.1.3 Custom allocators and arenas

Allocators, pools, arenas, and object graphs often require pointer-level bookkeeping.

16.1.4 Intrinsics and hardware interaction

Some hardware-level operations cannot be expressed purely in safe Rust.

16.1.5 Key rule

Unsafe is justified by invariants you can prove and enforce. If you cannot explain the invariant, you should not write the unsafe code.

16.2 The five core safety rules

Unsafe Rust allows you to perform operations that the compiler cannot verify. But you must still uphold Rust's fundamental safety invariants.
A practical set of five core rules you must enforce when writing unsafe:

16.2.1 Rule 1: Validity (no invalid values for a type)

You must not create values that violate type validity rules:

- references must not be null,
- references must be properly aligned,
- `bool` must be 0 or 1 at the representation level,
- `char` must be a valid Unicode scalar value,
- `&str` must be valid UTF-8.

Unsafe code that constructs values must ensure they are valid for their types.

16.2.2 Rule 2: Initialization (do not read uninitialized memory)

If you allocate raw memory, you must initialize it before reading. Uninitialized reads are undefined behavior.

16.2.3 Rule 3: Aliasing and exclusivity (no violating &mut rules)

Rust's safety model relies on aliasing rules:

- shared references (`&T`) may alias but must not be used to mutate through another path in a way that violates assumptions,
- mutable references (`&mut T`) must be unique (no other active references alias the same location).

Unsafe code must ensure it does not create overlapping `&mut` references or violate the exclusivity contract.

16.2.4 Rule 4: Lifetimes (no dangling references)

References must not outlive their referents. Unsafe code must ensure:

- a reference points to live memory,
- the pointed-to object remains valid for the entire reference lifetime,
- moved or freed memory is not referenced again.

16.2.5 Rule 5: Thread safety (no data races)

Even in unsafe Rust, data races are undefined behavior. If you share memory between threads, you must enforce synchronization and correctness.

These five rules are not "suggestions." They are the foundation that makes safe Rust safe. Unsafe Rust exists so you can implement low-level mechanisms, but the guarantees are still your responsibility.

16.3 Raw pointers and aliasing

16.3.1 Raw pointers are not references

`*const T` and `*mut T` are raw pointers:

- they can be null,
- they can be dangling,
- they do not carry borrowing rules automatically,
- dereferencing them is unsafe.

16.3.2 Creating raw pointers from references

```rust
fn main() {
    let mut x = 10_i32;

    let p1: *const i32 = &x as *const i32;
    let p2: *mut i32 = &mut x as *mut i32;

    unsafe {
        println!("{}", *p1);
        *p2 += 5;
        println!("{}", *p2);
    }
}
```

This compiles, but it does not mean “anything goes.” The moment you dereference `p2`, you must still obey aliasing and lifetime rules. The raw pointer does not magically authorize violating `&mut` uniqueness.

16.3.3 Aliasing hazards

A common unsafe mistake is to create two mutable views to the same memory.

```
fn bad_aliasing(xs: &mut [i32]) {
    let p = xs.as_mut_ptr();
    unsafe {
        let a = &mut *p;         /* points to xs[0] */
        let b = &mut *p;         /* another &mut to the same element */
        *a += 1;
        *b += 1;
    }
}
```

This is undefined behavior because it creates two live `&mut` references to the same location. The compiler assumes `&mut` is unique, and violating this breaks optimizations and correctness.

16.3.4 Preferred unsafe style: keep raw pointers raw, create references briefly

A safe pattern:

- do pointer arithmetic in raw pointers,
- convert to references only briefly for a single operation,
- avoid creating long-lived references from raw pointers.

```
fn add_to_first(xs: &mut [i32], delta: i32) {
    let p = xs.as_mut_ptr();
    unsafe {
        *p = *p + delta; /* raw pointer deref, no extra references created */
    }
}
```

16.3.5 Bounds checks and `get_unchecked`

Unsafe indexing is allowed when you can prove bounds correctness.

```rust
fn sum_first_two(xs: &[i32]) -> i32 {
    assert!(xs.len() >= 2);
    unsafe { *xs.get_unchecked(0) + *xs.get_unchecked(1) }
}
```

The proof here is the assert. In real systems, you often structure code so the proof is enforced by higher-level invariants.

16.4 Building safe abstractions on top of unsafe

16.4.1 The professional pattern

A stable professional pattern:

- unsafe code implements a low-level core,
- all invariants are documented,
- public API is safe and prevents violating invariants,
- unsafe blocks are small and locally justified.

16.4.2 Example: a safe wrapper over raw buffer

We store a raw pointer and a length. The safe API:

- prevents out-of-bounds access,
- ensures pointer validity assumptions are respected by construction.

```rust
pub struct RawBuf {
    ptr: *mut u8,
    len: usize,
}

impl RawBuf {
    pub fn new(ptr: *mut u8, len: usize) -> Self {
        Self { ptr, len }
    }

    pub fn len(&self) -> usize {
        self.len
    }

    pub fn get(&self, i: usize) -> Option<u8> {
        if i < self.len {
            unsafe { Some(*self.ptr.add(i)) }
        } else {
            None
        }
    }

    pub fn set(&mut self, i: usize, v: u8) -> bool {
        if i < self.len {
            unsafe { *self.ptr.add(i) = v; }
            true
        } else {
            false
        }
    }
}
```

This is still dangerous if constructed with invalid `ptr/len`. In a real design, you usually build

RawBuf only from a trusted allocator or from an owned backing store so construction itself enforces validity.

16.4.3 Document invariants next to unsafe

A strong habit is to place a comment directly above every unsafe block that states the invariant being relied on.

```
fn read_at(ptr: *const u8, i: usize) -> u8 {
    unsafe {
        /* Safety: caller guarantees ptr points to at least i+1 valid bytes. */
        *ptr.add(i)
    }
}
```

16.5 Lab: implementing an arena allocator (safe API)

This lab implements a minimal bump arena:

- allocate a fixed buffer once,
- return slices into it,
- advance an offset,
- free everything when the arena drops.

This demonstrates real unsafe technique:

- raw pointer arithmetic,
- alignment considerations,
- preventing aliasing violations by API design.

16.5.1 Design goals

We will build:

- Arena that owns a Vec<u8> buffer,
- a method alloc_bytes(n, align) that returns &mut [u8],
- safe behavior: no out-of-bounds, correct alignment, unique mutable slice returned.

16.5.2 Implementation

```
pub struct Arena {
    buf: Vec<u8>,
    off: usize,
}

impl Arena {
    pub fn with_capacity(cap: usize) -> Self {
        let mut buf = Vec::with_capacity(cap);
        buf.resize(cap, 0);
        Self { buf, off: 0 }
    }

    pub fn capacity(&self) -> usize {
        self.buf.len()
    }

    pub fn used(&self) -> usize {
        self.off
    }

    pub fn remaining(&self) -> usize {
        self.capacity().saturating_sub(self.off)
    }

    fn align_up(x: usize, align: usize) -> usize {
        debug_assert!(align.is_power_of_two());
        (x + (align - 1)) & !(align - 1)
    }

    pub fn alloc_bytes(&mut self, n: usize, align: usize) -> Option<&mut [u8]> {
        if n == 0 {
            return Some(&mut []);
        }
        if align == 0 || !align.is_power_of_two() {
            return None;
        }

        let start = Self::align_up(self.off, align);
        let end = start.checked_add(n)?;

        if end > self.capacity() {
            return None;
        }

        self.off = end;

        let base = self.buf.as_mut_ptr();
        unsafe {
            /* Safety:
               - buf is a valid allocation of capacity bytes and lives as long as self.
               - start..end is within bounds (checked).
               - start was aligned to `align`.
               - returned slice is unique because we advance `off` and never hand out
               ↪  overlapping regions.
            */
            let p = base.add(start);
            Some(std::slice::from_raw_parts_mut(p, n))
        }
    }

    pub fn reset(&mut self) {
        self.off = 0;
    }
}
```

16.5.3 Using the arena: building typed allocations (conceptual)

To allocate typed values, you would:

- request bytes with correct alignment for `T`,
- then write `T` into that memory,
- and return a reference with careful lifetime rules.

A safe typed interface is more advanced because it must ensure:

- proper initialization,
- correct drop behavior (or restrict to `Copy` / `Pod`-like types),
- no use-after-reset.

Here is a restricted typed interface for `Copy` values only (to avoid drop complexity):

```
impl Arena {
    pub fn alloc_copy<T: Copy>(&mut self, v: T) -> Option<&mut T> {
        let align = std::mem::align_of::<T>();
        let size = std::mem::size_of::<T>();
        let bytes = self.alloc_bytes(size, align)?;
        unsafe {
            /* Safety:
               - bytes is aligned and sized for T.
               - we write a fully initialized T.
               - returning &mut T is safe because bytes is unique.
            */
            let p = bytes.as_mut_ptr() as *mut T;
            p.write(v);
            Some(&mut *p)
        }
    }
}
```

16.5.4 Tests

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

    #[test]
    fn arena_alloc_basic() {
        let mut a = Arena::with_capacity(64);
        let x = a.alloc_bytes(8, 8).unwrap();
        assert_eq!(x.len(), 8);
        assert!(a.used() >= 8);

        let y = a.alloc_bytes(16, 16).unwrap();
        assert_eq!(y.len(), 16);
        assert!(a.used() >= 24);
    }

    #[test]
    fn arena_out_of_space() {
        let mut a = Arena::with_capacity(16);
        assert!(a.alloc_bytes(8, 8).is_some());
        assert!(a.alloc_bytes(16, 8).is_none());
    }

    #[test]
    fn arena_alloc_copy() {
        let mut a = Arena::with_capacity(64);
        let p = a.alloc_copy::<u64>(0x1122334455667788).unwrap();
        assert_eq!(*p, 0x1122334455667788);
    }

    #[test]
    fn arena_reset_reuses_space() {
        let mut a = Arena::with_capacity(32);
        assert!(a.alloc_bytes(16, 8).is_some());
        a.reset();
        assert!(a.alloc_bytes(16, 8).is_some());
    }
}
```

16.5.5 Lab extension: Vec-like structure (outline)

A minimal Vec-like structure requires unsafe for:

- managing uninitialized capacity,
- growing the buffer,
- moving elements,
- dropping initialized elements safely.

A correct design typically tracks:

- `ptr`: pointer to allocated memory,
- `len`: number of initialized elements,
- `cap`: allocated capacity.

Then:

- `push` writes into uninitialized slot,
- growth reallocates and moves initialized elements,
- `Drop` drops only initialized elements (0..len) and frees allocation.

16.5.6 Unsafe checklist for your own code

Before accepting unsafe code as “done”, verify:

- pointer validity and alignment are proven,
- bounds are proven before any `add` or deref,
- you never create two live `&mut` references to the same memory,
- lifetimes of returned references are correct and do not outlive storage,
- you do not read uninitialized memory,
- you document invariants next to each unsafe block,
- unsafe surface area is minimal and wrapped by safe APIs.

Unsafe Rust is the place where Rust becomes as powerful as C/C++ at the machine level. The difference is that Rust forces you to make this power explicit, local, and auditable.

Part V

Concurrency Without Pain

Threading Fundamentals

Rust's threading model is built on a simple but powerful idea: **threads are cheap to create, but sharing data is explicit and controlled**. Unlike many languages where data races are discovered late through testing or production failures, Rust moves most concurrency correctness checks to compile time.

This chapter establishes solid foundations:

- how threads are created and managed,
- how ownership is transferred safely into threads,
- how `JoinHandle` models thread lifecycle and results,
- and a hands-on lab that builds a simple but correct worker thread pool.

The goal is not to hide concurrency, but to make it **predictable, explicit, and scalable**.

17.1 Thread creation and management

17.1.1 Spawning a thread

The basic unit of concurrency in Rust is the OS thread, created via `std::thread::spawn`.

```
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        println!(”Hello from a thread”);
    });

    handle.join().unwrap();
}
```

Key observations:

- the closure passed to `spawn` runs in a new thread,
- the parent thread continues immediately,
- joining is explicit and optional.

17.1.2 Threads and OS reality

Rust threads are thin wrappers over native OS threads:

- they have real stacks,
- they are scheduled by the operating system,
- creating too many threads can hurt performance.

This is why thread pools and async models exist, but raw threads are still essential for:

- CPU-bound parallelism,
- long-running background workers,
- low-level system components.

17.1.3 Naming threads and configuration

Threads can be configured using `thread::Builder`.

```
use std::thread;

fn main() {
    let handle = thread::Builder::new()
        .name("worker-1".to_string())
        .spawn(|| {
            println!("running in named thread");
        })
        .unwrap();

    handle.join().unwrap();
}
```

Naming threads is extremely helpful for debugging and profiling.

17.1.4 Detached threads (fire-and-forget)

If you drop the `JoinHandle` without calling `join`, the thread becomes detached.

```
use std::thread;

fn main() {
    thread::spawn(|| {
        println!("background work");
    });

    /* main may exit before the thread finishes */
}
```

Professional rule:

- avoid detached threads unless their lifetime is truly independent,
- prefer explicit joining or structured concurrency patterns.

17.2 Ownership transfer into threads

17.2.1 Why ownership matters

Threads run concurrently and may outlive their creator. Rust therefore requires that data moved into a thread is:

- owned by the thread, or
- safely shared using synchronization primitives.

This is enforced by the type system.

17.2.2 Move closures

By default, closures borrow variables. Threads require ownership, so `move` is commonly used.

```
use std::thread;

fn main() {
    let msg = String::from(”hello”);

    let handle = thread::spawn(move || {
        println!(”{}”, msg);
    });

    handle.join().unwrap();
    /* msg is no longer accessible here */
}
```

The move keyword:

- transfers ownership into the closure,
- prevents accidental sharing of stack data,
- makes thread lifetimes explicit.

17.2.3 What can be sent to threads: Send

A type can be moved into another thread only if it implements the Send trait.
Most standard types are Send:

- integers, floats,
- String, Vec,
- Box<T> if T: Send.

Types that are **not** thread-safe (for example, unsynchronized interior mutability) are not Send.

17.2.4 Sharing read-only data

Immutable data can often be shared by cloning cheap handles or references.

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

fn main() {
    let data = Arc::new(vec![1, 2, 3]);

    let mut handles = Vec::new();
    for _ in 0..3 {
        let d = Arc::clone(&data);
        handles.push(thread::spawn(move || {
            println!("{:?}", d);
        }));
    }

    for h in handles {
        h.join().unwrap();
    }
}
```

17.2.5 The role of Arc

`Arc<T>` enables:

- shared ownership across threads,
- atomic reference counting,
- immutable or synchronized interior data.

Without `Arc`, shared ownership across threads is impossible in safe Rust.

17.3 JoinHandle

17.3.1 What JoinHandle represents

`JoinHandle<T>` represents:

- ownership of a spawned thread,
- the ability to wait for its completion,
- the ability to retrieve its result.

```
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        42
    });

    let result = handle.join().unwrap();
    println!("result = {}", result);
}
```

17.3.2 Error handling with join

If a thread panics, `join` returns an error.

```
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        panic!("something went wrong");
    });

    match handle.join() {
        Ok(_) => println!("completed"),
        Err(_) => println!("thread panicked"),
    }
}
```

This allows parent threads to:

- detect failures,
- restart workers,
- propagate errors upward.

17.3.3 Structured waiting

Joining threads explicitly creates a clear structure:

- threads start in known places,
- threads finish in known places,
- resource lifetimes are predictable.

This discipline is essential in large systems.

17.4 Lab: worker thread pool

This lab builds a minimal worker thread pool using:

- threads,
- channels for task distribution,
- ownership transfer for work items.

The pool will:

- start a fixed number of worker threads,
- accept jobs as closures,
- execute jobs concurrently,
- shut down cleanly.

17.4.1 Design overview

Key design choices:

- jobs are boxed closures: `Box<dyn FnOnce() + Send>`,
- a channel distributes jobs to workers,
- dropping the sender signals shutdown.

17.4.2 Job type

```
type Job = Box<dyn FnOnce() + Send + 'static>;
```

17.4.3 Worker implementation

```
use std::thread;
use std::sync::mpsc::{Receiver};

struct Worker {
    id: usize,
    handle: thread::JoinHandle<()>,
}

impl Worker {
    fn new(id: usize, rx: Receiver<Job>) -> Self {
        let handle = thread::spawn(move || {
            while let Ok(job) = rx.recv() {
                job();
            }
        });

        Self { id, handle }
    }
}
```

17.4.4 Thread pool implementation

```
use std::sync::mpsc::{self, Sender};

pub struct ThreadPool {
    workers: Vec<Worker>,
    tx: Sender<Job>,
}

impl ThreadPool {
    pub fn new(size: usize) -> Self {
        assert!(size > 0);

        let (tx, rx) = mpsc::channel::<Job>();

        let mut workers = Vec::with_capacity(size);
        for id in 0..size {
            workers.push(Worker::new(id, rx.clone()));
        }

        Self { workers, tx }
    }

    pub fn execute<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
        let job = Box::new(f);
        self.tx.send(job).unwrap();
    }
}
```

17.4.5 Graceful shutdown

When `ThreadPool` is dropped, the sender is dropped, causing all receivers to exit their loop.

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        drop(&self.tx);
        for worker in &mut self.workers {
            let _ = worker.handle.join();
        }
    }
}
```

17.4.6 Using the thread pool

```
fn main() {
    let pool = ThreadPool::new(4);

    for i in 0..8 {
        pool.execute(move || {
            println!("job {} executed", i);
        });
    }
}
```

17.4.7 What this lab demonstrates

This lab shows:

- ownership transfer of jobs into threads,

- controlled sharing via channels,
- explicit thread lifecycle management,
- no data races by construction.

17.4.8 Extensions for practice

To deepen understanding:

- add job results using a response channel,
- add graceful shutdown signals,
- add worker identification in logs,
- measure throughput under different pool sizes.

17.4.9 Key takeaways from this chapter

- Threads are explicit and powerful; Rust does not hide them.
- Ownership transfer into threads prevents accidental shared mutation.
- `JoinHandle` makes thread lifecycle and failure explicit.
- Correct thread pools can be built with small, understandable components.

Rust's threading model removes most of the fear from concurrency by making illegal states unrepresentable. The result is not only safer code, but code that scales with confidence.

Channels and Message Passing

Message passing is one of Rust's most practical concurrency tools because it naturally matches how systems are built:

- one part produces work,
- another part consumes and transforms it,
- ownership moves with the message,
- shared mutable state becomes optional rather than default.

Rust channels are designed to make data-race-free designs easy:

- send moves ownership into the channel,
- receivers own the message they receive,
- types enforce thread-safety constraints (`Send`).

In this chapter you will learn:

- the standard `mpsc` channel model,
- producer/consumer patterns that scale,
- the concept of backpressure and why unbounded queues can become a performance bug,
- and a lab that builds a concurrent file-processing pipeline.

18.1 mpsc channels

18.1.1 The basic model

`mpsc` means **multiple-producer, single-consumer**. The standard library provides:

- `Sender<T>` for sending values,
- `Receiver<T>` for receiving values.

Creating a channel:

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

fn main() {
    let (tx, rx) = mpsc::channel::<i32>();

    tx.send(10).unwrap();
    let v = rx.recv().unwrap();

    println!("{}", v);
}
```

Properties:

- `send` transfers ownership of `T` into the channel,
- `recv` blocks until a value is available or the channel is closed,
- if all senders are dropped, `recv` eventually returns an error.

18.1.2 Non-blocking receive with `try_recv`

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

fn main() {
    let (tx, rx) = mpsc::channel::<i32>();
    tx.send(1).unwrap();

    match rx.try_recv() {
        Ok(v) => println!("got {}", v),
        Err(_) => println!("no message yet"),
    }
}
```

Use `try_recv` when you:

- want to poll without blocking,
- integrate channel checks into event loops,
- want to do other work if no message is available.

18.1.3 Iterating over a receiver

A receiver can be used as an iterator that ends when the channel closes:

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

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

    thread::spawn(move || {
        tx.send("a".to_string()).unwrap();
        tx.send("b".to_string()).unwrap();
    });

    for msg in rx {
        println!("{}", msg);
    }
}
```

This pattern is clean for consumers:

- a single loop,
- ends automatically when producers go away.

18.1.4 Multiple producers

You create multiple producers by cloning the sender:

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

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

    let mut handles = Vec::new();
    for id in 0..4 {
        let t = tx.clone();
        handles.push(thread::spawn(move || {
            t.send(id).unwrap();
        }));
    }

    drop(tx); /* close when clones finish */
    for h in handles {
        h.join().unwrap();
    }

    let mut v: Vec<usize> = rx.iter().collect();
    v.sort();
    println!("{:?}", v);
}
```

18.1.5 What you can send

Values sent across threads must be `Send`. Most standard data types satisfy this. If your type is not `Send`, it is a signal that sharing it across threads would not be safe without additional design.

18.2 Producer/consumer patterns

18.2.1 Single producer, single consumer

The simplest pipeline: one thread reads, another thread processes.

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

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

    let producer = thread::spawn(move || {
        for s in ["a", "bb", "ccc"] {
            tx.send(s.to_string()).unwrap();
        }
    });
    let consumer = thread::spawn(move || {
        let mut total = 0;
        for msg in rx {
            total += msg.len();
        }
        total
    });

    producer.join().unwrap();
    let total = consumer.join().unwrap();
    println!("{}", total);
}
```

18.2.2 Multiple producers, single consumer (fan-in)

Multiple threads can feed one consumer. This is useful when:

- you read from multiple sources,
- you have multiple workers producing results,
- you aggregate or serialize results in one place.

18.2.3 One producer, multiple consumers (work queue)

Standard `mpsc` is single-consumer, so “multiple consumers” requires a design choice:

- one consumer thread receives and dispatches work,
- or you share the receiver behind a synchronization mechanism,
- or you use a different queue/channel design for multi-consumer workloads.

For learning purposes, a common approach is:

- one dispatcher receives,
- then it distributes tasks to worker channels.

18.2.4 Request/response channels

A classic pattern: each request contains a channel sender where the worker will send the response.

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

struct Request {
    x: i32,
    reply: mpsc::Sender<i32>,
}

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

    std::thread::spawn(move || {
        for req in rx {
            let _ = req.reply.send(req.x * 2);
        }
    });

    let (reply_tx, reply_rx) = mpsc::channel::<i32>();
    tx.send(Request { x: 21, reply: reply_tx }).unwrap();

    let ans = reply_rx.recv().unwrap();
    println!("{}", ans);
}
```

This avoids shared state and makes the flow explicit.

18.3 Backpressure concepts

18.3.1 What backpressure means

Backpressure means: **when consumers cannot keep up, producers must slow down or block**. Without backpressure, unbounded queues can grow indefinitely:

- memory usage increases,
- latency grows because messages sit in the queue,
- cache locality degrades,
- the system becomes unstable under load.

18.3.2 Unbounded channel behavior

The default `mpsc::channel` is effectively unbounded. If producers send faster than consumers receive, the queue grows.

This can be correct for low-volume messaging, but dangerous for high-throughput pipelines.

18.3.3 Bounded channels as backpressure

A bounded channel blocks senders when the buffer is full, forcing the system to stabilize.

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

fn main() {
    let (tx, rx) = mpsc::sync_channel::<i32>(2);

    tx.send(1).unwrap();
    tx.send(2).unwrap();
    /* tx.send(3) would block until rx receives */

    drop(rx);
}
```

`sync_channel` provides backpressure by design.

18.3.4 Backpressure as a performance feature

Backpressure is not “slowing down.” It is:

- controlling memory growth,
- bounding latency,
- and protecting the system from overload.

A professional pipeline often uses bounded queues between stages.

18.4 Lab: file-processing pipeline

This lab builds a concurrent pipeline:

- Stage 1: file reader produces lines,
- Stage 2: workers parse and process,
- Stage 3: aggregator collects statistics.

We implement:

- bounded channels between stages (backpressure),
- ownership transfer of data,

- clean shutdown.

To keep the lab self-contained, we will simulate file input with an in-memory vector of lines, but the structure matches real file processing.

18.4.1 Pipeline goal

Input lines have the format:

- `OK <number>`

- `ERR <code>`

Output:

- sum of OK numbers,

- counts of ERR codes,

- number of malformed lines.

18.4.2 Message types

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

#[derive(Debug)]
enum LineMsg {
    Line(String),
    End,
}

#[derive(Debug)]
enum WorkResult {
    OkValue(i64),
    ErrCode(i32),
    BadLine,
}
```

18.4.3 Stage 1: producer

Uses a bounded channel to apply backpressure if workers are slow.

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

fn spawn_reader(lines: Vec<String>, tx: mpsc::SyncSender<LineMsg>) ->
↪   thread::JoinHandle<()> {
    thread::spawn(move || {
        for line in lines {
            if tx.send(LineMsg::Line(line)).is_err() {
                return;
            }
        }
        let _ = tx.send(LineMsg::End);
    })
}
```

18.4.4 Stage 2: worker threads

Workers receive lines, parse, and send results.

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

fn parse_line(line: &str) -> WorkResult {
    let line = line.trim();
    let mut it = line.split_whitespace();
    let head = match it.next() {
        Some(h) => h,
        None => return WorkResult::BadLine,
    };
    let payload = match it.next() {
        Some(p) => p,
        None => return WorkResult::BadLine,
    };
    if it.next().is_some() {
        return WorkResult::BadLine;
    }

    match head {
        "OK" => match payload.parse::<i64>() {
            Ok(v) => WorkResult::OkValue(v),
            Err(_) => WorkResult::BadLine,
        },
        "ERR" => match payload.parse::<i32>() {
            Ok(c) => WorkResult::ErrCode(c),
            Err(_) => WorkResult::BadLine,
        },
        _ => WorkResult::BadLine,
    }
}

fn spawn_workers(
    n: usize,
    rx: mpsc::Receiver<LineMsg>,
    tx_res: mpsc::SyncSender<WorkResult>,
) -> Vec<thread::JoinHandle<()>> {
    let mut hs = Vec::with_capacity(n);

    for _ in 0..n {
        let r = rx.clone();
        let t = tx_res.clone();
        hs.push(thread::spawn(move || {
            loop {
                match r.recv() {
                    Ok(LineMsg::Line(line)) => {
                        let res = parse_line(&line);
                        if t.send(res).is_err() {
                            return;
                        }
                    }
                    Ok(LineMsg::End) => {
                        let _ = t.send(WorkResult::BadLine); /* optional marker */
                        return;
                    }
                    Err(_) => return,
                }
            }
        }));
    }

    hs
}
```

Note: Standard `mpsc::Receiver` cannot be cloned. The code above shows intent, but the correct way is to have one receiver and distribute work using a shared queue design. To keep this lab correct with standard library only, we implement a dispatcher that receives from the reader and forwards to per-worker channels.

18.4.5 Correct std-only worker distribution: dispatcher + per-worker channels

We create:

- one reader channel `rx_in` (single consumer),
- N worker channels `tx_w[i]/rx_w[i]`,
- one result channel `tx_res/rx_res`.

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

fn spawn_dispatcher(
    rx_in: mpsc::Receiver<LineMsg>,
    worker_txs: Vec<mpsc::SyncSender<String>>,
) -> thread::JoinHandle<()> {
    thread::spawn(move || {
        let mut idx = 0usize;
        let n = worker_txs.len();

        for msg in rx_in {
            match msg {
                LineMsg::Line(line) => {
                    let _ = worker_txs[idx].send(line);
                    idx = (idx + 1) % n;
                }
                LineMsg::End => {
                    for tx in &worker_txs {
                        let _ = tx.send(String::new()); /* empty means shutdown */
                    }
                    return;
                }
            }
        }
    })
}

fn spawn_worker(
    rx: mpsc::Receiver<String>,
    tx_res: mpsc::SyncSender<WorkResult>,
) -> thread::JoinHandle<()> {
    thread::spawn(move || {
        for line in rx {
            if line.is_empty() {
                return;
            }
            let res = parse_line(&line);
            if tx_res.send(res).is_err() {
                return;
            }
        }
    })
}
```

18.4.6 Stage 3: aggregator

```
use std::collections::HashMap;

#[derive(Debug)]
struct Stats {
    ok_sum: i64,
    err_counts: HashMap<i32, usize>,
    bad: usize,
}
```

```
fn collect_results(rx: mpsc::Receiver<WorkResult>, expected: usize) -> Stats {
    let mut s = Stats { ok_sum: 0, err_counts: HashMap::new(), bad: 0 };
    let mut seen = 0usize;

    while seen < expected {
        let r = rx.recv().unwrap();
        match r {
            WorkResult::OkValue(v) => s.ok_sum += v,
            WorkResult::ErrCode(c) => *s.err_counts.entry(c).or_insert(0) += 1,
            WorkResult::BadLine => s.bad += 1,
        }
        seen += 1;
    }

    s
}
```

In real systems you would use:

- explicit end-of-stream markers per worker,
- or dropping senders and iterating until `rx` closes.

18.4.7 Putting it all together

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

fn main() {
    let lines: Vec<String> = vec![
        "OK 10".into(),
        "ERR 2".into(),
        "OK 5".into(),
        "BAD 1".into(),
        "ERR 2".into(),
        "OK x".into(),
    ];

    let workers = 3usize;

    let (tx_in, rx_in) = mpsc::sync_channel::<LineMsg>(64);
    let (tx_res, rx_res) = mpsc::sync_channel::<WorkResult>(64);

    /* worker channels */
    let mut worker_txs = Vec::new();
    let mut worker_handles = Vec::new();

    for _ in 0..workers {
        let (tx_w, rx_w) = mpsc::sync_channel::<String>(32);
        worker_txs.push(tx_w);
        worker_handles.push(spawn_worker(rx_w, tx_res.clone()));
    }

    let h_reader = spawn_reader(lines.clone(), tx_in);
    let h_disp = spawn_dispatcher(rx_in, worker_txs);

    h_reader.join().unwrap();
    h_disp.join().unwrap();

    /* close result senders */
    drop(tx_res);

    /* gather results until channel closes */
    let mut stats = Stats { ok_sum: 0, err_counts: std::collections::HashMap::new(), bad: 0
    ↪   };
    for r in rx_res {
        match r {
            WorkResult::OkValue(v) => stats.ok_sum += v,
            WorkResult::ErrCode(c) => *stats.err_counts.entry(c).or_insert(0) += 1,
            WorkResult::BadLine => stats.bad += 1,
        }
    }

    for h in worker_handles {
        let _ = h.join();
    }

    println!("sum={}", stats.ok_sum);
    println!("bad={}", stats.bad);
    println!("errs={:?}", stats.err_counts);
}
```

18.4.8 What the lab teaches

- **Ownership moves with messages:** no shared mutation is required.
- **Backpressure is deliberate:** bounded channels prevent unbounded memory growth.
- **Shutdown is explicit:** end messages and dropping senders close pipelines.
- **Architecture scales:** add stages, increase workers, change work distribution.

18.4.9 Professional extensions

To make this pipeline production-grade:

- avoid per-line `String` allocation by reading a large buffer and slicing,
- use a work item type that carries offsets into a shared buffer,
- batch messages to reduce channel overhead,

- add metrics (queue sizes, throughput, latency),
- add structured error reporting from workers.

Message passing is not only a safety strategy. It is an architectural tool that improves modularity, testability, and scalability while keeping concurrency predictable.

Arc, Mutex, and RwLock in Practice

Shared-state concurrency is sometimes the simplest correct design:

- many threads need access to the same structure,
- updates must be consistent,
- and message passing would add overhead or complexity.

Rust does not forbid shared state. It requires it to be **explicit, synchronized, and type-checked**. The common building blocks are:

- `Arc<T>` for shared ownership across threads,
- `Mutex<T>` for exclusive access to mutable state,
- `RwLock<T>` for many-readers / one-writer patterns.

This chapter is practical and engineering-oriented:

- how to choose the right primitive,
- how to avoid deadlocks and performance traps,
- and a lab that builds a multi-threaded cache with correct synchronization.

19.1 Choosing the right primitive

19.1.1 A decision framework

When you need shared data across threads, ask these questions:

- Do I need **shared ownership** of the data? If yes, use `Arc<T>`.
- Do I need **mutation**? If yes, wrap the data in `Mutex<T>` or `RwLock<T>`.
- Is the workload **read-heavy** with rare writes? Consider `RwLock<T>`.
- Can I avoid sharing by message passing? If yes, that can reduce contention.

19.1.2 `Arc<T>` by itself is not mutability

`Arc<T>` only provides shared ownership. It does not allow mutation unless `T` itself provides interior mutability or synchronization.
Correct mental model:

- `Arc<T>` solves lifetime and ownership,
- `Mutex`/`RwLock` solve synchronized mutation.

19.1.3 Arc + Mutex: the standard shared mutable pattern

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

fn main() {
    let shared = Arc::new(Mutex::new(0_i64));

    let mut hs = Vec::new();
    for _ in 0..4 {
        let s = Arc::clone(&shared);
        hs.push(thread::spawn(move || {
            for _ in 0..1000 {
                let mut g = s.lock().unwrap();
                *g += 1;
            }
        }));
    }

    for h in hs {
        h.join().unwrap();
    }

    println!("{}", *shared.lock().unwrap());
}
```

This pattern is correct, but may suffer from contention if the critical section is too small and too frequent.

19.1.4 Arc + RwLock: many readers, occasional writers

If many threads mostly read and rarely write, RwLock can increase throughput.

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

fn main() {
    let cfg = Arc::new(RwLock::new(String::from("v1")));

    let readers: Vec<_> = (0..4).map(|_| {
        let c = Arc::clone(&cfg);
        thread::spawn(move || {
            for _ in 0..1000 {
                let g = c.read().unwrap();
                let _ = g.len();
            }
        })
    }).collect();

    let writer = {
        let c = Arc::clone(&cfg);
        thread::spawn(move || {
            let mut g = c.write().unwrap();
            g.push_str("-updated");
        })
    };

    for r in readers {
        r.join().unwrap();
    }
    writer.join().unwrap();

    println!("{}", *cfg.read().unwrap());
}
```

19.1.5 When `Mutex` beats `RwLock`

A common myth is “RwLock is always faster for reads.” Not necessarily:

- if writes are frequent, readers may starve or be blocked often,
- read locks still cost synchronization,
- some workloads benefit from a simpler `Mutex` due to lower overhead.

A professional rule:

- choose based on access pattern and measurement, not ideology.

19.1.6 Lock poisoning and panic behavior

If a thread panics while holding a lock, Rust may mark it as poisoned. Locking returns an error to signal potential invariant breakage.
Basic handling:

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

fn main() {
    let m = Mutex::new(1_i32);

    let guard = match m.lock() {
        Ok(g) => g,
        Err(poisoned) => poisoned.into_inner(),
    };

    println!("{}", *guard);
}
```

In high-reliability systems, poisoning strategy is a policy decision:

- crash fast if invariants are critical,

- recover if the state can be reconstructed safely.

19.2 Avoiding deadlocks

19.2.1 What deadlock is

Deadlock occurs when:

- thread A holds lock 1 and waits for lock 2,
- thread B holds lock 2 and waits for lock 1,
- and neither can proceed.

Rust prevents data races, but **cannot** prevent deadlocks automatically because deadlocks are a runtime ordering problem.

19.2.2 Deadlock avoidance rules

A practical set of rules that scale:

Rule 1: Keep lock scope small

Hold locks only for:

- the minimal critical section,
- quick state changes,
- copying needed data out.

Then release and do expensive work outside the lock.

Rule 2: Avoid calling user code while holding a lock

Do not call closures, callbacks, logging that might lock, or external code while holding locks.

Rule 3: Establish a global lock ordering

If you must hold multiple locks, always acquire them in the same order everywhere.
Example: always lock `A` then `B`, never the opposite.

Rule 4: Prefer one lock over many

If two structures are tightly coupled, one lock might be safer than two.

Rule 5: Use `try_lock` for contention-sensitive paths

`try_lock` fails rather than blocking. This can prevent lock chains and reduce worst-case latency.

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

fn maybe_update(m: &Mutex<i32>) {
    if let Ok(mut g) = m.try_lock() {
        *g += 1;
    } else {
        /* skip or defer work */
    }
}
```

19.2.3 Common deadlock pattern and fix

Bad: nested locks with unknown order.

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

fn main() {
    let a = Arc::new(Mutex::new(0_i32));
    let b = Arc::new(Mutex::new(0_i32));

    let a1 = Arc::clone(&a);
    let b1 = Arc::clone(&b);
    let t1 = thread::spawn(move || {
        let _ga = a1.lock().unwrap();
        let _gb = b1.lock().unwrap();
    });

    let a2 = Arc::clone(&a);
    let b2 = Arc::clone(&b);
    let t2 = thread::spawn(move || {
        let _gb = b2.lock().unwrap();
        let _ga = a2.lock().unwrap();
    });

    let _ = t1.join();
    let _ = t2.join();
}
```

Fix: enforce a global ordering (always lock `a` then `b`) or combine into one lock.

19.3 Lab: multi-threaded cache

We will build a concurrent cache:

- shared across threads,

- supports get-or-compute,

- avoids holding locks while doing expensive computation,

- uses `Arc<RwLock<HashMap<K, V»>` for read-heavy workloads.

The cache design will emphasize correctness and performance.

19.3.1 Cache requirements

- Many reads, occasional insertions.

- If a key is missing, compute outside locks.
- Avoid returning references tied to lock guards (keep API simple).

19.3.2 Implementation

```
use std::collections::HashMap;
use std::hash::Hash;
use std::sync::{Arc, RwLock};

pub struct Cache<K, V> {
    inner: Arc<RwLock<HashMap<K, V>>>,
}

impl<K, V> Cache<K, V>
where
    K: Eq + Hash + Clone,
    V: Clone,
{
    pub fn new() -> Self {
        Self {
            inner: Arc::new(RwLock::new(HashMap::new())),
        }
    }

    pub fn shared(&self) -> Arc<RwLock<HashMap<K, V>>> {
        Arc::clone(&self.inner)
    }

    pub fn get(&self, k: &K) -> Option<V> {
        let g = self.inner.read().unwrap();
        g.get(k).cloned()
    }
    pub fn insert(&self, k: K, v: V) {
        let mut g = self.inner.write().unwrap();
        g.insert(k, v);
    }

    pub fn get_or_compute<F>(&self, k: K, compute: F) -> V
    where
        F: FnOnce(&K) -> V,
    {
        /* Fast path: read lock */
        {
            let g = self.inner.read().unwrap();
            if let Some(v) = g.get(&k) {
                return v.clone();
            }
        }

        /* Slow path: compute outside lock */
        let v = compute(&k);

        /* Insert with write lock (double-check) */
        let mut g = self.inner.write().unwrap();
        if let Some(existing) = g.get(&k) {
            return existing.clone();
        }
        g.insert(k, v.clone());
        v
    }
}
```

Key performance property:

- computation happens outside the lock,

- the write lock is held only briefly for insertion,
- readers scale well in read-heavy usage.

19.3.3 Multi-threaded usage example

```
use std::thread;

fn expensive(k: &u64) -> u64 {
    /* placeholder for real work */
    k.wrapping_mul(10).wrapping_add(1)
}

fn main() {
    let cache: Cache<u64, u64> = Cache::new();

    let mut hs = Vec::new();
    for i in 0..8u64 {
        let c = cache.shared();
        hs.push(thread::spawn(move || {
            /* build a thin wrapper around shared lock for demonstration */
            let cache = Cache { inner: c };

            for _ in 0..1000 {
                let _v = cache.get_or_compute(i % 4, expensive);
            }
        }));
    }

    for h in hs {
        h.join().unwrap();
    }
    /* final cache size should be small */
    let g = cache.shared().read().unwrap();
    println!("cache size = {}", g.len());
}
```

19.3.4 Why this is correct

- all shared mutation is protected by `RwLock`,
- the API returns owned `V` to avoid lifetime coupling to lock guards,
- the "double-check" prevents overwriting values inserted by other threads,
- lock scope is minimal and computation is outside locks.

19.3.5 Lab extensions

To deepen your understanding:

- switch `RwLock` to `Mutex` and compare behavior conceptually for write-heavy loads,
- add an eviction policy (LRU-like) and observe how it complicates locking,
- add `try_read` / `try_write` in latency-sensitive contexts,
- add a sharded cache: multiple locks, each guarding a subset of keys (reduces contention).

19.3.6 Deadlock and contention checklist for real systems

When using `Arc`, `Mutex`, and `RwLock` in production:

- keep critical sections short,
- avoid nested locks unless you have a strict global ordering,

- never perform expensive IO while holding a lock,
- do not call unknown user code while holding locks,
- prefer coarse locks first, then shard only if measurement shows contention,
- handle poisoning according to your reliability policy.

Rust’s synchronization primitives give you controlled shared-state concurrency. The “pain” disappears when you treat locking as architecture: explicit boundaries, documented invariants, and minimal lock scope.

Atomics and Memory Ordering

Atomics are the lowest-level synchronization tool most systems programmers will ever use. They are powerful, fast, and dangerous when used without a correct mental model.
Rust makes atomic operations **explicit** and ties them to a small vocabulary of memory orderings. This helps you write code that is:

- correct on all CPU architectures (not just on your development machine),
- efficient (avoids unnecessary fences),
- and maintainable (orderings are documented at the call site).

This chapter focuses on practical engineering:

- memory ordering in real terms (what it prevents and what it allows),
- safe atomic patterns you can reuse confidently,
- and a lab that builds a lock-free counter as a controlled learning exercise.

20.1 Ordering in practical terms

20.1.1 What an atomic operation guarantees

An atomic operation guarantees that the read-modify-write is indivisible with respect to other atomic operations on the same location. It does **not** automatically guarantee:

- visibility of other non-atomic memory writes,
- ordering of operations across different memory locations,
- absence of higher-level logic races.

That is what memory ordering is for.

20.1.2 Two separate concerns

Memory ordering answers two questions:

- **Atomicity:** is the operation indivisible? (Yes, by definition for atomics.)
- **Ordering/visibility:** what other memory operations can be reordered around it, and what becomes visible to other threads?

20.1.3 The Rust memory ordering vocabulary

Rust exposes these orderings (via `std::sync::atomic::Ordering`):

- `Relaxed`
- `Acquire`
- `Release`

- `AcqRel`

- `SeqCst`

You should treat them as: **a contract with the compiler and the CPU**.

20.1.4 `Relaxed`: atomicity without synchronization

`Relaxed` means:

- operations are atomic,

- but they do not establish ordering constraints with other memory operations.

Use case:

- counters, statistics, metrics where exact ordering with other data is not required.

```rust
use std::sync::atomic::{AtomicU64, Ordering};

static HITS: AtomicU64 = AtomicU64::new(0);

fn hit() {
    HITS.fetch_add(1, Ordering::Relaxed);
}
```

This is correct for “counting events” even under high contention, but it does not synchronize any other data.

20.1.5 `Release` and `Acquire`: publish and observe

A common practical pattern is **publish/subscribe**:

- Writer initializes data, then performs a `store(..., Release)` on a flag.

- Reader performs `load(..., Acquire)` on the flag, and once it observes the flag, it can safely read the published data.

This establishes a **happens-before** relationship:

- writes before the Release become visible after the Acquire in the other thread.

Conceptual example:

```rust
use std::sync::atomic::{AtomicBool, Ordering};

static READY: AtomicBool = AtomicBool::new(false);

fn producer(shared: &mut [u8]) {
    shared[0] = 42;
    /* publish: all writes before this become visible to acquiring readers */
    READY.store(true, Ordering::Release);
}

fn consumer(shared: &[u8]) -> Option<u8> {
    if READY.load(Ordering::Acquire) {
        Some(shared[0])
    } else {
        None
    }
}
```

Important:

- This illustrates ordering, not a complete safe shared-memory program by itself.
- In real Rust code, you avoid unsafely sharing `&mut` across threads; you use safe shared containers or build a safe abstraction.

20.1.6 `AcqRel`: read-modify-write with synchronization

Used for operations like `fetch_add` when you need both acquire and release semantics in one atomic RMW.

```rust
use std::sync::atomic::{AtomicUsize, Ordering};

static STATE: AtomicUsize = AtomicUsize::new(0);

fn transition() -> usize {
    STATE.fetch_add(1, Ordering::AcqRel)
}
```

20.1.7 `SeqCst`: the simplest mental model (and the most expensive)

`SeqCst` provides the strongest constraints:

- all sequentially-consistent atomic operations appear in one global total order.

Use `SeqCst` when:

- you want maximum clarity and correctness for complex interactions,
- you are not sure which weaker ordering is correct,
- performance cost is acceptable or not in the hot path.

```rust
use std::sync::atomic::{AtomicBool, Ordering};

static FLAG: AtomicBool = AtomicBool::new(false);

fn set() {
    FLAG.store(true, Ordering::SeqCst);
}
```

```rust
fn is_set() -> bool {
    FLAG.load(Ordering::SeqCst)
}
```

Professional approach:

- start with `SeqCst` for correctness,
- then relax orderings only when you can prove safety and measure benefit.

20.2 Safe atomic patterns

Atomics are often misused when developers treat them as “lock-free = easy”. The safe path is to use patterns that are known to be correct and to keep atomics small and local.

20.2.1 Pattern 1: Relaxed counters

Use `Relaxed` for monotonic counters and statistics.

```rust
use std::sync::atomic::{AtomicU64, Ordering};

pub struct Stats {
    bytes: AtomicU64,
    reqs: AtomicU64,
}

impl Stats {
    pub const fn new() -> Self {
        Self { bytes: AtomicU64::new(0), reqs: AtomicU64::new(0) }
    }
    pub fn on_request(&self, nbytes: u64) {
        self.reqs.fetch_add(1, Ordering::Relaxed);
        self.bytes.fetch_add(nbytes, Ordering::Relaxed);
    }

    pub fn snapshot(&self) -> (u64, u64) {
        let r = self.reqs.load(Ordering::Relaxed);
        let b = self.bytes.load(Ordering::Relaxed);
        (r, b)
    }
}
```

20.2.2 Pattern 2: One-time initialization flag (Acquire/Release)

A classic pattern: publish an initialized resource once. In real Rust, prefer safe one-time init mechanisms, but the ordering idea is essential.

```
use std::sync::atomic::{AtomicBool, Ordering};

static READY: AtomicBool = AtomicBool::new(false);

fn init_once(data: &mut [u8]) {
    if !READY.load(Ordering::Acquire) {
        data[0] = 7;
        READY.store(true, Ordering::Release);
    }
}
```

20.2.3 Pattern 3: State machine in an atomic integer

An atomic state variable is often safer than multiple flags.

```
use std::sync::atomic::{AtomicU8, Ordering};
```

```rust
const INIT: u8 = 0;
const RUNNING: u8 = 1;
const STOPPING: u8 = 2;
const STOPPED: u8 = 3;

pub struct Service {
    state: AtomicU8,
}

impl Service {
    pub const fn new() -> Self {
        Self { state: AtomicU8::new(INIT) }
    }

    pub fn start(&self) -> bool {
        self.state.compare_exchange(INIT, RUNNING, Ordering::AcqRel,
        ↪  Ordering::Acquire).is_ok()
    }

    pub fn stop(&self) -> bool {
        self.state.compare_exchange(RUNNING, STOPPING, Ordering::AcqRel,
        ↪  Ordering::Acquire).is_ok()
    }

    pub fn is_running(&self) -> bool {
        self.state.load(Ordering::Acquire) == RUNNING
    }
}
```

This uses compare_exchange to enforce legal transitions.

20.2.4 Pattern 4: Reference counting is not your job

Do not implement your own lock-free reference counting without deep expertise. Use `Arc` or established data structures. Atomics are not a replacement for ownership modeling; they are a low-level tool used by such models.

20.2.5 Pattern 5: Do not mix atomics and non-atomics on the same data

If a memory location is accessed concurrently:

- either all accesses are synchronized,
- or you have a data race, which is undefined behavior.

Rule of thumb:

- do not read/write shared data from multiple threads without a clear synchronization plan.

20.3 Lab: a simple lock-free counter

This lab builds a correct lock-free counter using `AtomicU64` and compares it conceptually with a mutex counter.
Goal:

- build a thread-safe counter with atomics,
- understand which ordering is sufficient,
- practice multi-threaded increment and final read.

20.3.1 Atomic counter

```rust
use std::sync::atomic::{AtomicU64, Ordering};
use std::sync::Arc;
use std::thread;

#[derive(Default)]
pub struct AtomicCounter {
    v: AtomicU64,
}

impl AtomicCounter {
    pub const fn new() -> Self {
        Self { v: AtomicU64::new(0) }
    }

    pub fn inc(&self) {
        self.v.fetch_add(1, Ordering::Relaxed);
    }

    pub fn get(&self) -> u64 {
        self.v.load(Ordering::Relaxed)
    }
}

fn main() {
    let c = Arc::new(AtomicCounter::new());

    let mut hs = Vec::new();
    for _ in 0..8 {
        let cc = Arc::clone(&c);
        hs.push(thread::spawn(move || {
            for _ in 0..100_000 {
                cc.inc();
            }
        }));
    }

    for h in hs {
        h.join().unwrap();
    }

    println!("final = {}", c.get());
}
```

Why `Relaxed` is enough here:

- we only need atomicity of the counter itself,
- we do not use the counter to publish or protect other data,
- after joining all threads, we are in a single thread and the final load is sufficient.

20.3.2 Counter with `SeqCst` (clarity-first variant)

If you want the simplest mental model for learning:

```
use std::sync::atomic::{AtomicU64, Ordering};

pub struct CounterSeq {
    v: AtomicU64,
}

impl CounterSeq {
    pub const fn new() -> Self {
        Self { v: AtomicU64::new(0) }
    }
    pub fn inc(&self) {
        self.v.fetch_add(1, Ordering::SeqCst);
    }

    pub fn get(&self) -> u64 {
        self.v.load(Ordering::SeqCst)
    }
}
```

This is often slower under contention but is harder to misuse.

20.3.3 Mutex counter (comparison baseline)

A mutex-based counter is also correct but has higher overhead due to lock acquisition under contention.

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

pub struct MutexCounter {
    v: Mutex<u64>,
}

impl MutexCounter {
    pub fn new() -> Self {
        Self { v: Mutex::new(0) }
    }

    pub fn inc(&self) {
        let mut g = self.v.lock().unwrap();
        *g += 1;
    }
    pub fn get(&self) -> u64 {
        *self.v.lock().unwrap()
    }
}

fn main() {
    let c = Arc::new(MutexCounter::new());

    let mut hs = Vec::new();
    for _ in 0..8 {
        let cc = Arc::clone(&c);
        hs.push(thread::spawn(move || {
            for _ in 0..100_000 {
                cc.inc();
            }
        }));
    }

    for h in hs {
        h.join().unwrap();
    }

    println!("final = {}", c.get());
}
```

20.3.4 Lab questions

To ensure the mental model is correct, answer these after implementing:

- Why is `Relaxed` sufficient for counting but not for publishing other data?
- What changes if another thread reads the counter and uses it as a signal to read other

shared memory?

- Under heavy contention, why might a counter become a bottleneck even if it is lock-free?

20.3.5 Next step: from counters to lock-free structures

A counter is the simplest lock-free structure. Real lock-free designs require:

- careful state machines,
- correct ordering proofs,
- avoidance of ABA problems in pointer-based structures,
- memory reclamation strategies.

The engineering rule:

- use atomics directly only when the structure is simple and well understood,
- otherwise rely on proven synchronization primitives or established lock-free libraries.

Atomics are part of “concurrency without pain” only when you treat them with respect: explicit ordering, reusable patterns, and minimal surface area.

Part VI

Async Rust in Production

Understanding `async/await`

Rust `async/await` is not “magic threads” and not “hidden parallelism”. It is a disciplined model for **cooperative concurrency**:

- `async fn` produces a **Future** (a value representing an in-progress computation),
- the Future makes progress only when it is **polled** by an **executor**,
- when the Future cannot make progress, it returns **Pending** and arranges to be polled again via a **Waker**.

This chapter builds the correct mental model from first principles and then moves to practical engineering:

- Futures explained (what they are and how they run),
- runtime concepts (executor, tasks, reactor, timers, and scheduling),
- pinning from introductory to advanced (why `Pin` exists and how it relates to `async`),
- a lab that runs basic async tasks (both with a runtime and with a tiny educational executor).

21.1 Futures explained

21.1.1 The core contract: `Future::poll`

A Future is a state machine. Conceptually, it exposes one method:

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

pub trait Future {
    type Output;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}
```

Interpretation:

- `Poll::Ready(value)` means the future is complete and produced its output.
- `Poll::Pending` means it cannot complete *right now*.
- When returning `Pending`, the future must ensure that it will eventually call `cx.waker().wake()` (or equivalent) when it can make progress again.

This is the entire mechanism behind `await`.

21.1.2 `async fn` becomes a state machine

An `async fn` compiles into a hidden type that implements `Future`. Each `.await` point becomes a suspension point in the state machine.

```rust
async fn read_then_process() -> usize {
    let a = step1().await;
    let b = step2(a).await;
    b.len()
}
```

Mentally translate to:

- “Start step1; if pending, yield.”
- “When step1 ready, start step2; if pending, yield.”
- “When step2 ready, return result.”

21.1.3 Why `await` is not blocking

Blocking means: the OS thread sleeps/waits. Awaiting means: the current task yields control to the executor so other tasks can run on the same OS thread.
The executor decides when to poll your task again.

21.1.4 Wakers: how futures resume

When an async operation depends on an external event (socket readable, timer fired, file ready), the future:

- registers interest in that event,
- stores the `Waker` (or a clone),
- returns `Pending`,
- later, when the event occurs, the runtime calls `wake()` which schedules a new poll.

Conceptual pattern inside a future:

```
use std::task::{Context, Poll};

fn poll_something(cx: &mut Context<'_>) -> Poll<u32> {
    if is_ready_now() {
        Poll::Ready(123)
    } else {
        register_waker_somewhere(cx.waker().clone());
        Poll::Pending
    }
}
```

21.1.5 A minimal custom future (educational)

Here is a future that returns ready only after it has been polled n times. This is not useful in production, but it illustrates the polling model.

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

pub struct PollNTimes {
    remaining: u32,
}

impl PollNTimes {
    pub fn new(n: u32) -> Self {
        Self { remaining: n }
    }
}

impl Future for PollNTimes {
    type Output = ();
    fn poll(mut self: Pin<&mut Self>, _cx: &mut Context<'_>) -> Poll<()> {
        if self.remaining == 0 {
            Poll::Ready(())
        } else {
            self.remaining -= 1;
            Poll::Pending
        }
    }
}
```

Notice:

- returning `Pending` without arranging a wake means the executor would have to poll you again by some other mechanism;
- real futures must use the waker to avoid busy-polling.

21.2 Runtime concepts

Rust provides the **language** features (`async`/`await`) and the **Future** trait, but it does **not** ship a general-purpose production executor in `std`. In practice, you run async Rust with a runtime that provides:

- an **executor** (polls tasks),
- a **task system** (spawning, scheduling, cancellation policy),
- a **reactor/event loop** (IO readiness via OS mechanisms),
- **timers** and time drivers,
- often a **work-stealing scheduler** for multi-thread execution.

21.2.1 Executor vs reactor (practical mental model)

Executor: decides *which future to poll next* and on which thread.

Reactor: waits for OS events (socket readable, timer fired) and wakes the appropriate tasks.

A future that performs IO typically:

- tries a non-blocking operation,
- if it would block, registers interest in the reactor and returns `Pending`,
- later the reactor wakes the task, and the executor polls again.

21.2.2 Tasks vs threads

A **task** is a scheduled future. Thousands of tasks can be multiplexed onto a small number of OS threads.

Key implications:

- tasks must not block the thread (no long CPU loops without yielding, no blocking IO),
- CPU-heavy work typically goes to dedicated threads or special blocking facilities,
- shared state must still be synchronized (async does not remove concurrency hazards).

21.2.3 Structured concurrency mindset

A production-grade async design prefers:

- explicit task ownership and lifetimes,
- bounded queues and backpressure,
- cancellation-aware loops,
- timeouts and deadlines as first-class behavior.

21.2.4 Minimal runtime example (using a common runtime)

The following shows the shape of a runtime-driven async program:

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

async fn child(id: u32) -> u32 {
    /* pretend this is IO; in a real runtime you'd await a timer or socket */
    id * 2
}

async fn parent() -> u32 {
    let a = child(10).await;
    let b = child(20).await;
    a + b
}

/* In a real project, the runtime macro starts the executor and reactor. */
fn main() {
    /* placeholder: run parent() on your runtime */
    /* e.g., runtime.block_on(parent()) */
}
```

The important point is not the macro or the crate name. The point is: **a runtime repeatedly polls your futures until completion**.

21.3 Pinning (introductory to advanced)

Pinning is one of the most misunderstood parts of async Rust, mainly because you can be productive with `async`/`await` for a long time before you ever need to write `Pin` yourself. You still must understand it, because it explains:

- why `Future::poll` takes `Pin<&mut Self>`,

- why some futures are `!Unpin`,
- why certain self-referential patterns are forbidden in safe Rust without pinning.

21.3.1 What pinning actually guarantees

`Pin<P>` is a wrapper that means:

- the value behind pointer `P` will not be moved *by safe code*,
- unless the value's type implements `Unpin`.

Important nuance:

- "not moved" means its memory address remains stable,
- this is about *movement of the value*, not about mutability.

21.3.2 `Unpin`: movable even when pinned

If `T: Unpin`, then pinning does not impose restrictions; you can still move `T` safely.
Most ordinary types are `Unpin`.
The problematic class is **self-referential** data, where the value contains a pointer/reference into itself.

21.3.3 Why futures often need pinning

Async state machines may hold references to their own internal fields across suspension points.
If that future were moved in memory after such internal references are created, those references would become invalid.
Therefore, `poll` requires:

```
fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
```

So the executor can pin the future in memory and safely poll it across time.

21.3.4 How pinning shows up in everyday async code

In typical async code:

- `await` pins internal temporaries for you,
- spawning a task pins the future internally inside the runtime,
- you rarely write `Pin` unless you implement custom futures, self-referential structs, or certain combinators.

21.3.5 Pinning on the stack: `pin!` and `Box::pin`

Two common ways to pin a future:

- pin on the stack for a local scope,
- pin on the heap for long-lived storage or trait objects.

```
use std::pin::Pin;

fn example_local_pin<F>(mut f: F)
where
    F: std::future::Future<Output = ()>,
{
    /* f is pinned for the duration of this scope (conceptually) */
    let _pf: Pin<&mut F> = unsafe { Pin::new_unchecked(&mut f) };
}
```

In production code, you usually avoid manual `unsafe` pin creation and use safer helpers or store futures in `Pin<Box<...»`:

```
use std::future::Future;
use std::pin::Pin;
```

```
fn heap_pin<F>(f: F) -> Pin<Box<F>>
where
    F: Future,
{
    Box::pin(f)
}
```

21.3.6 Advanced: pinning and self-referential invariants

Pinning is only half the story:

- pinning provides a stable address,
- you must still ensure you never create invalid internal references,
- and you must enforce that the value is pinned before those references are relied upon.

This is why self-referential safe types are usually implemented with:

- careful construction APIs,
- internal raw pointers,
- unsafe blocks with strict invariants,
- and a safe public surface.

21.3.7 Pinning and trait objects

Sometimes you want to store heterogeneous futures behind a trait object:

```
use std::future::Future;
use std::pin::Pin;

type BoxFuture<T> = Pin<Box<dyn Future<Output = T> + Send + 'static>>;
```

Why `Pin<Box<dyn Future...»` and not just `Box<dyn Future...>`? Because polling requires pinning, and trait objects must support being polled safely without moving the future.

21.4 Lab: basic async tasks

This lab gives you two complementary experiences:

- running async code on a production-style runtime (how real programs are built),
- running async code on a tiny educational executor (how the mechanism works).

21.4.1 Lab A: spawn a few tasks (runtime-style)

Goal:

- spawn multiple tasks,
- await their completion,
- understand ownership transfer into tasks,
- observe that tasks are lightweight compared to OS threads.

```
/* Cargo.toml (conceptual)
[dependencies]
tokio = { version = "1", features = ["rt-multi-thread", "macros", "time"] }
*/

use std::time::Duration;

async fn work(id: u32) -> u32 {
    tokio::time::sleep(Duration::from_millis(10)).await;
    id * 10
}

#[tokio::main]
async fn main() {
    let a = tokio::spawn(async { work(1).await });
    let b = tokio::spawn(async { work(2).await });
    let c = tokio::spawn(async { work(3).await });

    let ra = a.await.unwrap();
    let rb = b.await.unwrap();
    let rc = c.await.unwrap();

    println!(”sum = {}”, ra + rb + rc);
}
```

Exercises:

- Add a loop that spawns 10,000 tiny tasks and measure total runtime.
- Move an owned `String` into each task with `move` and confirm ownership rules.
- Introduce one task that panics and observe join error behavior.

21.4.2 Lab B: a tiny educational executor (block_on)

Goal:

- understand polling, wakers, and why futures must arrange wakeups,
- run a future to completion with a minimal `block_on`.

This executor is intentionally minimal and not production-ready. It is a learning tool.

```rust
use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll, RawWaker, RawWakerVTable, Waker};
use std::thread;
use std::time::Duration;

/* A waker that simply does nothing. This works only for futures that
   eventually return Ready without needing real wakeups. */
fn dummy_waker() -> Waker {
    unsafe fn clone(_: *const ()) -> RawWaker { RawWaker::new(std::ptr::null(), &VTABLE) }
    unsafe fn wake(_: *const ()) {}
    unsafe fn wake_by_ref(_: *const ()) {}
    unsafe fn drop(_: *const ()) {}

    static VTABLE: RawWakerVTable = RawWakerVTable::new(clone, wake, wake_by_ref, drop);

    unsafe { Waker::from_raw(RawWaker::new(std::ptr::null(), &VTABLE)) }
}

pub fn block_on<F: Future>(mut fut: F) -> F::Output {
    let waker = dummy_waker();
    let mut cx = Context::from_waker(&waker);

    /* Pin the future on the stack for this function’s scope */
    let mut fut = unsafe { Pin::new_unchecked(&mut fut) };

    loop {
        match fut.as_mut().poll(&mut cx) {
            Poll::Ready(v) => return v,
            Poll::Pending => {
                /* In a real executor, we'd park and rely on wakeups.
                   Here we just sleep briefly and poll again. */
                thread::sleep(Duration::from_millis(1));
            }
        }
    }
}

async fn demo() -> u32 {
    7
}

fn main() {
    let v = block_on(demo());
    println!("{}", v);
}
```

Exercises:

- Replace `demo()` with a custom future that returns `Pending` a few times and then `Ready`.
- Observe that without real wakeups, the executor must poll repeatedly (inefficient).
- Extend the executor with a channel-based wake mechanism (advanced).

21.4.3 Reality check: why real runtimes matter

Production async needs:

- IO readiness integration (reactor),
- efficient scheduling and wakeups (executor),
- timers, cancellation, and backpressure,
- careful separation of CPU-bound work from async tasks.

21.4.4 Key takeaways

- A Future is a state machine driven by `poll`.
- `await` is cooperative: it yields, it does not block the OS thread.
- A runtime is required to poll futures and to drive IO and timers.
- `Pin` exists to guarantee stable addresses for futures and other self-referential patterns.
- Most developers rarely write `Pin` directly, but understanding it prevents subtle design mistakes and unlocks advanced systems work.

The Async Ecosystem

Rust provides the `async/await` language feature and the `Future` trait, but production async systems depend on an ecosystem of runtimes, I/O drivers, and structured concurrency utilities. In practice, **Tokio** is the dominant general-purpose async runtime for high-performance networked services. Its design centers on:

- an executor that polls tasks,
- an I/O driver that integrates with OS readiness notifications,
- a scheduling model that scales from single-thread to multi-thread runtimes,
- a set of ergonomic primitives for building real servers (TCP/UDP, timers, synchronization, cancellation).

This chapter is intentionally practical:

- Tokio fundamentals (how the runtime, tasks, and time driver fit together),
- TCP/UDP async I/O (correct patterns, buffering, and backpressure),
- structured concurrency (making task lifetimes explicit and leak-resistant),
- a lab that builds a TCP server with graceful shutdown.

Tokio evolves continuously; as of early 2026, the Tokio `1.49.0` release line is current, and Tokio also publishes LTS branches for long-lived production deployments. :contentReference[oaicite:0]index=0

22.1 Tokio fundamentals

22.1.1 What Tokio provides (in practical terms)

Tokio supplies the missing production pieces around `Future`:

- **Runtime:** an executor plus I/O driver plus timers.
- **Tasks:** lightweight, scheduled futures (`tokio::spawn`).
- **Async I/O:** `TcpListener`, `TcpStream`, `UdpSocket`, with readiness-driven non-blocking operations.
- **Time:** `sleep`, `interval`, timeouts.
- **Sync primitives:** async-aware mutexes, semaphores, channels (Tokio's own and std).

A task is a **future managed by the runtime**. Spawning a task returns a join handle, and Tokio schedules the task until completion. :contentReference[oaicite:1]index=1

22.1.2 Runtime flavors and why they matter

Tokio runtimes commonly come in two flavors:

- **Multi-thread runtime:** runs tasks across a thread pool (good default for servers).
- **Current-thread runtime:** runs tasks on a single OS thread (useful for embedded-like constraints, deterministic execution, or specialized setups).

A common production rule:

- use multi-thread runtime for I/O-heavy servers,
- isolate CPU-heavy work with `spawn_blocking` or dedicated worker pools,
- keep async tasks non-blocking and cooperative.

22.1.3 Project setup (minimal)

Tokio is feature-flag driven. For a typical networked service you enable:

- runtime + macros (for `#[tokio::main]`),
- net (TCP/UDP),
- time (timers),
- optionally io-util (buffered utilities) and sync.

```
[dependencies]
tokio = { version = "1.49", features = ["rt-multi-thread", "macros", "net", "time",
↪  "io-util", "sync"] }
tokio-util = { version = "0.7", features = ["sync"] }
```

22.1.4 The non-blocking rule (the #1 async performance killer)

Async tasks must not block the runtime threads. Common mistakes:

- using blocking filesystem APIs in hot paths,
- running CPU-heavy loops without yielding,
- holding locks across `.await`,

- doing long computations inside request handlers.

Tokio offers a clear escape hatch for blocking work:

```rust
use tokio::task;

fn heavy_cpu(x: u64) -> u64 {
    (0..5_000_000).fold(x, |a, i| a ^ (i as u64))
}

#[tokio::main]
async fn main() {
    let x = task::spawn_blocking(|| heavy_cpu(123)).await.unwrap();
    println!("{}", x);
}
```

22.2 TCP/UDP async I/O

22.2.1 TCP fundamentals: stream, framing, and buffering

TCP is a byte stream:

- no message boundaries,

- reads may return partial data,

- writes may not send everything at once.

Therefore, production TCP protocols need **framing**. Common patterns:

- length-prefix framing (4-byte length + payload),

- delimiter framing (newline-delimited),

- fixed-size frames (rare; used in specialized protocols).

Tokio provides AsyncRead / AsyncWrite traits and utilities in tokio::io to build correct buffering and framing.

22.2.2 Minimal TCP accept loop (the skeleton)

```rust
use tokio::net::TcpListener;

#[tokio::main]
async fn main() -> anyhow::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:9000").await?;

    loop {
        let (stream, addr) = listener.accept().await?;
        println!("accepted {}", addr);

        tokio::spawn(async move {
            let _ = handle_client(stream).await;
        });
    }
}

async fn handle_client(_stream: tokio::net::TcpStream) -> anyhow::Result<()> {
    Ok(())
}
```

This is only the starting point. Real servers must also:

- bound concurrency,
- implement timeouts,
- implement backpressure,

- support graceful shutdown.

22.2.3 UDP fundamentals: datagrams, not streams

UDP is message-based:

- each receive returns one datagram,
- datagrams can be lost, duplicated, or reordered,
- there is no built-in connection handshake.

Tokio's `UdpSocket` supports both:

- one-to-many via `send_to`/`recv_from`,
- one-to-one via `connect` then `send`/`recv`.

:contentReference[oaicite:2]index=2

22.2.4 UDP example: echo datagrams back

```rust
use tokio::net::UdpSocket;

#[tokio::main]
async fn main() -> anyhow::Result<()> {
    let sock = UdpSocket::bind("127.0.0.1:7000").await?;
    let mut buf = [0u8; 2048];

    loop {
        let (n, peer) = sock.recv_from(&mut buf).await?;
        sock.send_to(&buf[..n], &peer).await?;
    }
}
```

Engineering note:

- for high throughput, consider buffering strategies and bounded queues,
- for reliability, build an application-level protocol (sequence numbers, retransmit policy, timeouts).

22.2.5 Timeouts and cancellation around I/O

Timeouts are essential to avoid resource starvation.

```
use tokio::net::TcpStream;
use tokio::time::{timeout, Duration};

#[tokio::main]
async fn main() -> anyhow::Result<()> {
    let connect = TcpStream::connect("127.0.0.1:9000");
    let stream = timeout(Duration::from_secs(2), connect).await??;
    drop(stream);
    Ok(())
}
```

22.3 Structured concurrency patterns

Unstructured concurrency is when tasks are spawned and then forgotten. It often leads to:

- "task leaks" (background tasks outliving the subsystem),
- shutdown bugs (tasks keep running after services should stop),
- error invisibility (panics/errors are never observed),
- unbounded resource growth (too many tasks, sockets, or buffers).

Structured concurrency means:

- tasks have a clear owner,
- completion (or cancellation) is awaited,
- shutdown is explicit and predictable.

22.3.1 Pattern 1: JoinSet as a task group

Tokio provides JoinSet to manage a collection of spawned tasks and await them as they finish. :contentReference[oaicite:3]index=3

```
use tokio::task::JoinSet;

async fn run_group() -> anyhow::Result<()> {
    let mut set = JoinSet::new();

    for i in 0..10u32 {
        set.spawn(async move {
            i * 2
        });
    }

    let mut sum = 0u32;
    while let Some(res) = set.join_next().await {
        sum += res?;
    }

    println!("sum = {}", sum);
    Ok(())
}
```

This creates an ownership boundary:

- the group owns the tasks,
- the group drains results,
- errors are observed.

22.3.2 Pattern 2: `select!` for "whichever happens first"

`select!` is the async equivalent of multiplexing:

- read from socket OR shutdown signal,
- timer tick OR new request,
- background worker completion OR deadline.

```rust
use tokio::time::{sleep, Duration};

async fn one() -> u32 { sleep(Duration::from_millis(50)).await; 1 }
async fn two() -> u32 { sleep(Duration::from_millis(10)).await; 2 }

async fn race() -> u32 {
    tokio::select! {
        a = one() => a,
        b = two() => b,
    }
}
```

22.3.3 Pattern 3: graceful shutdown with cancellation tokens

A widely used approach is a shared cancellation signal. Tokio-Util provides `CancellationToken` which tasks can await to know when to stop. :contentReference[oaicite:4]index=4

```rust
use tokio_util::sync::CancellationToken;
use tokio::time::{sleep, Duration};

async fn worker(token: CancellationToken) {
    loop {
        tokio::select! {
            _ = token.cancelled() => {
                break;
            }
            _ = sleep(Duration::from_millis(100)) => {
                /* periodic work */
            }
        }
    }
}

async fn run() {
    let token = CancellationToken::new();

    let t1 = tokio::spawn(worker(token.clone()));
    let t2 = tokio::spawn(worker(token.clone()));

    token.cancel();

    let _ = t1.await;
    let _ = t2.await;
}
```

Engineering rules that prevent most shutdown bugs:

- every long-lived task must be cancel-aware,
- owners must await their spawned tasks (or drain JoinSet),
- shutdown must have deadlines (avoid hanging forever).

22.3.4 Pattern 4: bounding concurrency (do not spawn infinitely)

Spawning tasks per connection is fine, but you must bound concurrency for safety. Tokio's semaphore is a common tool:

```
use std::sync::Arc;
use tokio::net::TcpListener;
use tokio::sync::Semaphore;

#[tokio::main]
async fn main() -> anyhow::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:9000").await?;
    let limit = Arc::new(Semaphore::new(200)); /* max concurrent connections */

    loop {
        let (stream, _) = listener.accept().await?;
        let permit = limit.clone().acquire_owned().await?;

        tokio::spawn(async move {
            let _permit = permit; /* held until end of task */
            let _ = handle(stream).await;
        });
    }
}

async fn handle(_s: tokio::net::TcpStream) -> anyhow::Result<()> {
    Ok(())
}
```

This prevents load spikes from turning into memory exhaustion.

22.4 Lab: building a TCP server

This lab builds a real TCP server skeleton with:

- per-connection tasks,
- bounded concurrency,
- newline-delimited framing,
- graceful shutdown via cancellation token,
- clean error handling.

We implement an **echo server** with a tiny protocol:

- client sends lines of UTF-8 text terminated by `\n`,
- server echoes each line back,
- server supports a shutdown signal.

22.4.1 Step 1: connection handler with buffered lines

```
use tokio::net::TcpStream;
use tokio::io::{AsyncBufReadExt, AsyncWriteExt, BufReader};

async fn handle_connection(mut stream: TcpStream) -> anyhow::Result<()> {
    let (r, mut w) = stream.split();
    let mut reader = BufReader::new(r);
    let mut line = String::new();

    loop {
        line.clear();
        let n = reader.read_line(&mut line).await?;
        if n == 0 {
            /* client closed */
            return Ok(());
        }

        /* echo back exactly what we received */
        w.write_all(line.as_bytes()).await?;
        w.flush().await?;
    }
}
```

Notes:

- `read_line` is a framing strategy (delimiter framing).
- For binary protocols, you typically implement length-prefix framing instead.

22.4.2 Step 2: accept loop with bounded concurrency

```
use std::sync::Arc;
use tokio::net::TcpListener;
use tokio::sync::Semaphore;

async fn run_server(listener: TcpListener, limit: Arc<Semaphore>) -> anyhow::Result<()> {
    loop {
        let (stream, addr) = listener.accept().await?;
        let permit = limit.clone().acquire_owned().await?;

        tokio::spawn(async move {
            let _permit = permit;
            let _ = handle_connection(stream).await;
            let _ = addr; /* keep addr available for logging if needed */
        });
    }
}
```

22.4.3 Step 3: graceful shutdown with select!

We integrate a cancellation token and stop accepting connections when shutdown is requested.

```
use std::sync::Arc;
use tokio::net::TcpListener;
use tokio::sync::Semaphore;
use tokio_util::sync::CancellationToken;

async fn run_server_with_shutdown(
    listener: TcpListener,
    limit: Arc<Semaphore>,
    token: CancellationToken,
) -> anyhow::Result<()> {
    loop {
        tokio::select! {
            _ = token.cancelled() => {
                break;
            }
            res = listener.accept() => {
                let (stream, _addr) = res?;
                let permit = limit.clone().acquire_owned().await?;

                let t = token.clone();
                tokio::spawn(async move {
                    let _permit = permit;

                    tokio::select! {
                        _ = t.cancelled() => { /* stop early */ }
                        res = handle_connection(stream) => { let _ = res; }
                    }
                });
            }
        }
    }
    Ok(())
}
```

22.4.4 Step 4: main function (wiring it together)

This version also demonstrates a simple shutdown trigger using Ctrl+C.

```
use std::sync::Arc;
use tokio::net::TcpListener;
use tokio::sync::Semaphore;
use tokio_util::sync::CancellationToken;

#[tokio::main]
async fn main() -> anyhow::Result<()> {
    let listener = TcpListener::bind(”127.0.0.1:9000”).await?;
    let limit = Arc::new(Semaphore::new(200));
    let token = CancellationToken::new();

    let server_token = token.clone();
    let server = tokio::spawn(async move {
        run_server_with_shutdown(listener, limit, server_token).await
    });

    tokio::select! {
        _ = tokio::signal::ctrl_c() => {
            token.cancel();
        }
        res = server => {
            let _ = res?;
        }
    }

    let _ = server.await?;
    Ok(())
}
```

22.4.5 Lab exercises (to make it production-grade)

Improve the server in these steps:

Exercise A: add timeouts

Add an idle timeout so clients cannot hold connections forever:

- per read: if no line arrives within N seconds, drop the connection,
- per request: enforce maximum line length.

Exercise B: add backpressure and bounded output

If clients read slowly, writes can accumulate. Implement:

- a bounded per-connection output queue,
- or explicit limits on pending bytes.

Exercise C: add a simple command protocol

Support commands:

- `PING` → `PONG`

- `ECHO <text>` → `<text>`
- `QUIT` → close connection

Exercise D: structured task ownership

Replace unstructured spawning with a `JoinSet` owned by the server loop:

- insert each connection task into the set,
- drain completed tasks periodically,
- on shutdown: cancel token and then drain remaining tasks with a deadline.

22.4.6 Key takeaways

- Tokio is not just syntax support; it is a production runtime with scheduling, I/O, and time drivers. :contentReference[oaicite:5]index=5
- TCP requires explicit framing; UDP is datagram-based and needs protocol discipline. :contentReference[oaicite:6]index=6
- Structured concurrency is the difference between stable services and “mysterious background tasks”.
- A real server must bound concurrency, implement timeouts, and shut down gracefully with clear task ownership.

Building a Web Service

A web service is an applied test of everything you learned so far: types, ownership boundaries, error design, concurrency, performance discipline, and (in async Rust) task structure.
In this chapter we build a small REST service using a modern Rust async stack:

- type-safe routing and request extraction,
- explicit state injection (`State`),
- middleware as layered services,
- and a test strategy that validates handlers without spinning up a real TCP server.

The examples are written in a production-oriented style:

- clear request/response models,
- explicit error mapping,
- bounded resource usage where appropriate,
- and testable architecture.

23.1 Routing fundamentals

23.1.1 What “routing” really is

Routing is the mapping from an incoming HTTP request to a handler:

- match on method (GET/POST/PUT/DELETE),
- match on path pattern (e.g., `/items/:id`),
- extract typed data (path parameters, query, headers, JSON body),
- call a handler function that returns a response.

In strongly-typed Rust routing, the goal is:

- make invalid requests fail early and predictably,
- keep handlers small and testable,
- avoid stringly-typed parsing scattered across the codebase.

23.1.2 Minimal service skeleton (axum style)

A good default architecture:

- `main.rs` wires the router and server,
- `routes.rs` defines routing table,
- `handlers.rs` contains request handlers,
- `domain.rs` contains business types and validation,
- `errors.rs` contains an application error type mapped to HTTP responses.

Minimal dependencies:

```
[dependencies]
tokio = { version = "1", features = ["rt-multi-thread", "macros", "net", "time"] }
axum = "0.8"
serde = { version = "1", features = ["derive"] }
serde_json = "1"
tower-http = { version = "0.6", features = ["trace", "timeout", "request-id", "cors"] }
```

Minimal router:

```
use axum::{
    routing::{get, post},
    Router,
};

fn routes() -> Router {
    Router::new()
        .route("/health", get(health))
        .route("/echo", post(echo))
}

async fn health() -> &'static str {
    "ok"
}

async fn echo(body: String) -> String {
    body
}
```

Key idea:

- routing should be declarative and boring,
- business logic belongs in handlers and domain modules.

23.1.3 Extractors: turning HTTP into typed inputs

Extractors convert request parts into typed parameters:

- Path<T> for path parameters,
- Query<T> for query strings,
- Json<T> for JSON body,
- State<T> for application state injection.

Example: /items/:id with typed id:

```
use axum::{extract::Path, routing::get, Router};
use serde::Serialize;

#[derive(Serialize)]
struct Item {
    id: u64,
    name: String,
}

async fn get_item(Path(id): Path<u64>) -> axum::Json<Item> {
    axum::Json(Item { id, name: format!("item-{id}") })
}

fn routes() -> Router {
    Router::new().route("/items/:id", get(get_item))
}
```

23.1.4 Returning responses

In production you want:

- simple success returns (Json<T>, String, status codes),
- a single application error type that implements IntoResponse.

Example: return status plus JSON:

```
use axum::{http::StatusCode, response::IntoResponse, Json};
use serde::Serialize;

#[derive(Serialize)]
struct Created {
    id: u64,
}

async fn create() -> impl IntoResponse {
    (StatusCode::CREATED, Json(Created { id: 1 }))
}
```

23.2 Middleware concepts

23.2.1 Middleware as layers

Middleware is a function (or service) that wraps your handler:

- inspect or modify the request before it reaches the handler,
- inspect or modify the response after the handler,
- enforce cross-cutting policies: timeouts, tracing, auth, CORS, compression, request IDs.

In the Tower-based model, middleware is expressed as **layers** that transform a Service into another Service.

23.2.2 Global middleware vs route middleware

Two common deployment styles:

- apply middleware globally to the whole router (observability, request IDs, global timeouts),
- apply middleware to a sub-router or single route (auth for protected endpoints).

23.2.3 Practical middleware stack (recommended baseline)

A production baseline:

- request ID: correlate logs across services,
- tracing/logging: request span, latency, status codes,
- timeout: protect runtime threads and downstream resources,
- CORS (if this is a public HTTP API),
- body size limits (prevent abuse and accidental overload),
- optional rate limiting at edge or app layer.

Example: attach tower-http layers:

```
use axum::Router;
use std::time::Duration;
use tower_http::{
    cors::{Any, CorsLayer},
    request_id::{MakeRequestUuid, PropagateRequestIdLayer, SetRequestIdLayer},
    timeout::TimeoutLayer,
    trace::TraceLayer,
};
```

```
fn with_middleware(app: Router) -> Router {
    let cors = CorsLayer::new()
        .allow_origin(Any)
        .allow_methods(Any)
        .allow_headers(Any);

    app.layer((
        SetRequestIdLayer::x_request_id(MakeRequestUuid),
        PropagateRequestIdLayer::x_request_id(),
        TraceLayer::new_for_http(),
        TimeoutLayer::new(Duration::from_secs(10)),
        cors,
    ))
}
```

23.2.4 The most important async safety rule for middleware

Never hold a blocking lock across `.await`. Common mistakes:

- lock a `std::sync::Mutex` and then await I/O,
- hold an `RwLock` guard while awaiting database calls.

If you must lock around shared data in async code:

- keep lock scopes tiny,
- copy/clone only what is necessary out of the lock,
- then drop the guard before awaiting.

23.2.5 Writing a small custom middleware

Sometimes you want an app-specific guard such as “reject requests missing a header”. Axum offers middleware utilities; a simple pattern is:

- read headers,
- return an early error response if policy fails,
- otherwise forward to the next service.

```rust
use axum::{
    http::{Request, StatusCode},
    middleware::Next,
    response::Response,
};

pub async fn require_api_key<B>(req: Request<B>, next: Next<B>) -> Result<Response,
↪   StatusCode> {
    let ok = req.headers()
        .get("x-api-key")
        .and_then(|v| v.to_str().ok())
        .map(|s| !s.is_empty())
        .unwrap_or(false);

    if !ok {
        return Err(StatusCode::UNAUTHORIZED);
    }

    Ok(next.run(req).await)
}
```

Attach it to a protected router:

```
use axum::{middleware, routing::get, Router};

async fn secret() -> &'static str { "classified" }

fn protected() -> Router {
    Router::new()
        .route("/secret", get(secret))
        .route_layer(middleware::from_fn(require_api_key))
}
```

23.3 Lab: REST API with tests

This lab builds a small REST service: an in-memory task manager.
Endpoints:

- `POST /tasks` create a task,
- `GET /tasks/:id` read one,
- `GET /tasks` list tasks,
- `DELETE /tasks/:id` delete one.

We also add:

- a clean application error type mapped to HTTP,
- middleware baseline,
- and tests that call the router directly using `ServiceExt::oneshot`.

23.3.1 Domain types

```
use serde::{Deserialize, Serialize};

#[derive(Clone, Serialize)]
pub struct Task {
    pub id: u64,
    pub title: String,
    pub done: bool,
}

#[derive(Deserialize)]
pub struct CreateTask {
    pub title: String,
}
```

23.3.2 State and storage

For learning we use an in-memory `HashMap`. The goal is:

- keep lock scope minimal,
- ensure every handler is deterministic and testable.

```
use std::{collections::HashMap, sync::Arc};
use tokio::sync::RwLock;

#[derive(Default)]
pub struct Store {
    next_id: u64,
    tasks: HashMap<u64, Task>,
}

#[derive(Clone, Default)]
```

```
pub struct AppState {
    inner: Arc<RwLock<Store>>,
}
```

23.3.3 Application error mapped to HTTP

```
use axum::{
    http::StatusCode,
    response::{IntoResponse, Response},
    Json,
};
use serde::Serialize;

#[derive(Debug)]
pub enum AppError {
    NotFound,
    BadRequest(&'static str),
    Internal,
}

#[derive(Serialize)]
struct ErrBody {
    error: String,
}

impl IntoResponse for AppError {
    fn into_response(self) -> Response {
        let (code, msg) = match self {
            AppError::NotFound => (StatusCode::NOT_FOUND, "not found"),
            AppError::BadRequest(m) => (StatusCode::BAD_REQUEST, m),
            AppError::Internal => (StatusCode::INTERNAL_SERVER_ERROR, "internal error"),
        };
        (code, Json(ErrBody { error: msg.to_string() })).into_response()
    }
}

type AppResult<T> = Result<T, AppError>;
```

23.3.4 Handlers

```
use axum::{
    extract::{Path, State},
    http::StatusCode,
    response::IntoResponse,
    Json,
};

pub async fn create_task(
    State(st): State<AppState>,
    Json(req): Json<CreateTask>,
) -> AppResult<impl IntoResponse> {
    let title = req.title.trim();
    if title.is_empty() {
        return Err(AppError::BadRequest(”title must not be empty”));
    }

    let mut g = st.inner.write().await;
    g.next_id += 1;
    let id = g.next_id;

    let task = Task { id, title: title.to_string(), done: false };
    g.tasks.insert(id, task.clone());
    drop(g);

    Ok((StatusCode::CREATED, Json(task)))
```

```rust
}

pub async fn get_task(
    State(st): State<AppState>,
    Path(id): Path<u64>,
) -> AppResult<Json<Task>> {
    let g = st.inner.read().await;
    let t = g.tasks.get(&id).cloned().ok_or(AppError::NotFound)?;
    Ok(Json(t))
}

pub async fn list_tasks(State(st): State<AppState>) -> AppResult<Json<Vec<Task>>> {
    let g = st.inner.read().await;
    let mut v: Vec<Task> = g.tasks.values().cloned().collect();
    v.sort_by_key(|t| t.id);
    Ok(Json(v))
}

pub async fn delete_task(
    State(st): State<AppState>,
    Path(id): Path<u64>,
) -> AppResult<StatusCode> {
    let mut g = st.inner.write().await;
    let existed = g.tasks.remove(&id).is_some();
    drop(g);

    if existed {
        Ok(StatusCode::NO_CONTENT)
    } else {
        Err(AppError::NotFound)
    }
}
```

23.3.5 Router wiring

```
use axum::{
    routing::{delete, get, post},
    Router,
};

pub fn app(state: AppState) -> Router {
    let api = Router::new()
        .route("/tasks", post(create_task).get(list_tasks))
        .route("/tasks/:id", get(get_task).delete(delete_task))
        .with_state(state);

    with_middleware(api)
}
```

23.3.6 Main entry point

```
use tokio::net::TcpListener;

#[tokio::main]
async fn main() -> anyhow::Result<()> {
    let state = AppState::default();
    let app = app(state);

    let listener = TcpListener::bind("127.0.0.1:8080").await?;
    axum::serve(listener, app).await?;
    Ok(())
}
```

23.3.7 Testing the REST API without a real server

The best learning test style:

- build the router,
- call it as a service with oneshot,
- assert on status codes and JSON bodies.

```
#[cfg(test)]
mod tests {
    use super::*;
    use axum::{body::Body, http::{Request, StatusCode}};
    use tower::ServiceExt;

    fn req_json(method: &str, path: &str, json: &str) -> Request<Body> {
        Request::builder()
            .method(method)
            .uri(path)
            .header("content-type", "application/json")
            .body(Body::from(json.to_string()))
            .unwrap()
    }

    #[tokio::test]
    async fn create_and_get_task() {
        let state = AppState::default();
        let app = app(state);

        let r1 = app.clone().oneshot(req_json("POST", "/tasks",
        ↪  r#"{"title":"hello"}"#)).await.unwrap();
        assert_eq!(r1.status(), StatusCode::CREATED);

        let body = axum::body::to_bytes(r1.into_body(), usize::MAX).await.unwrap();
        let created: Task = serde_json::from_slice(&body).unwrap();
        assert_eq!(created.title, "hello");
        assert_eq!(created.done, false);
        let r2 = app.oneshot(Request::builder().method("GET").uri(format!("/tasks/{}",
         ↪   created.id)).body(Body::empty()).unwrap())
            .await.unwrap();
        assert_eq!(r2.status(), StatusCode::OK);

        let body2 = axum::body::to_bytes(r2.into_body(), usize::MAX).await.unwrap();
        let fetched: Task = serde_json::from_slice(&body2).unwrap();
        assert_eq!(fetched.id, created.id);
        assert_eq!(fetched.title, "hello");
    }

    #[tokio::test]
    async fn delete_task_then_404() {
        let state = AppState::default();
        let app = app(state);

        let r1 = app.clone().oneshot(req_json("POST", "/tasks",
         ↪   r#"{"title":"x"}"#)).await.unwrap();
        assert_eq!(r1.status(), StatusCode::CREATED);

        let body = axum::body::to_bytes(r1.into_body(), usize::MAX).await.unwrap();
        let created: Task = serde_json::from_slice(&body).unwrap();

        let r2 = app.clone()
            .oneshot(Request::builder().method("DELETE").uri(format!("/tasks/{}",
            ↪   created.id)).body(Body::empty()).unwrap())
            .await.unwrap();
        assert_eq!(r2.status(), StatusCode::NO_CONTENT);

        let r3 = app
            .oneshot(Request::builder().method("GET").uri(format!("/tasks/{}",
            ↪   created.id)).body(Body::empty()).unwrap())
```

```
            .await.unwrap();
        assert_eq!(r3.status(), StatusCode::NOT_FOUND);
    }

    #[tokio::test]
    async fn validation_rejects_empty_title() {
        let state = AppState::default();
        let app = app(state);

        let r = app.oneshot(req_json("POST", "/tasks",
        ↪  r#"{"title":"   "}"#)).await.unwrap();
        assert_eq!(r.status(), StatusCode::BAD_REQUEST);
    }
}
```

23.3.8 Lab extensions (production realism)

After the basic lab passes, upgrade it:

Extension A: add pagination

- support `GET /tasks?limit=50&offset=0`,
- validate bounds to prevent abuse.

Extension B: add structured logging and request IDs

- ensure every response includes request ID,
- log latency and status codes.

Extension C: add timeouts and body limits

- enforce an overall request timeout,

- reject bodies larger than a maximum size.

Extension D: refactor into library plus binary

- move `app(state)` into `lib.rs`,
- keep `main.rs` as wiring only,
- keep tests in the library so they are fast and stable.

23.3.9 Key takeaways

- Routing should be declarative and type-driven.
- Middleware should enforce cross-cutting policies without leaking into business logic.
- Tests should validate behavior by calling the router as a service, not by relying on fragile integration setups.
- Good Rust web services are built from small, explicit, testable parts.

Part VII

Rust in the Real World

C Interoperability (FFI)

Rust is designed to interoperate with existing C ecosystems: operating-system APIs, vendor SDKs, legacy libraries, embedded firmware, and performance-critical codebases that already expose C ABIs. This chapter focuses on writing **correct, stable, and auditable** FFI layers: you explicitly model calling conventions, data layout, ownership, lifetimes, and error behavior, then wrap the unsafe boundary behind safe Rust APIs.

24.1 extern `"C"`

24.1.1 What `extern "C"` means

In Rust, the ABI (Application Binary Interface) controls how function arguments are passed, how return values are produced, which registers/stack locations are used, and which symbol names are expected by linkers and debuggers. The C ABI is the lingua franca: if you can speak `extern "C"`, you can interoperate with most systems libraries.
There are two common directions:

- **Rust calling C**: declare C functions in an `extern "C"` block (no body).
- **C calling Rust**: export Rust functions using `extern "C"` and usually `#[no_mangle]`.

24.1.2 Calling C from Rust: external blocks

The following declares functions implemented in a C library:

```rust
use std::ffi::{c_char, c_int, c_void};

extern "C" {
    fn strlen(s: *const c_char) -> usize;

    fn memcpy(dst: *mut c_void, src: *const c_void, n: usize) -> *mut c_void;

    fn puts(s: *const c_char) -> c_int;
}
```

Why these declarations are `unsafe`

Calling an `extern` function is `unsafe` because Rust cannot verify:

- the pointer arguments are non-null, properly aligned, and valid for the required size,
- the function obeys the declared signature,
- the function respects Rust aliasing assumptions (it may mutate through `*const` pointers),
- the function's error behavior and side effects.

A safe wrapper must impose and document invariants that make the call sound.

24.1.3 Exporting Rust to C: `#[no_mangle]` and stable signatures

To expose a Rust function as a C symbol, use:

```rust
use std::ffi::{c_char, c_int};
```

```
#[no_mangle]
pub extern "C" fn add_i32(a: i32, b: i32) -> i32 {
    a + b
}
```

The `panic` rule: do not unwind across a C ABI

Rust panics must not unwind through a non-unwinding ABI such as `"C"`. Treat all exported FFI functions as **panic-free** boundaries. Common patterns:

- Write code that cannot panic (no indexing without checks, no `unwrap`, no overflow panics).
- Convert panics into error codes using `catch_unwind` at the boundary (shown later).

24.1.4 Function pointers and callbacks

C libraries often accept callbacks. Rust can pass an `extern "C"` function pointer, plus a user-data pointer (an opaque `void*`):

```
use std::ffi::{c_int, c_void};

pub type CCallback = Option<extern "C" fn(code: c_int, user: *mut c_void)>;

extern "C" {
    fn register_cb(cb: CCallback, user: *mut c_void) -> c_int;
}
```

Rule: never pass a Rust closure directly as a C callback. Instead, store the closure in a heap allocation, pass the pointer as `user`, and use a thin `extern "C"` trampoline that restores the pointer and calls the closure inside Rust (with strict lifetime rules).

24.2 Data layout and `repr(C)`

24.2.1 Why Rust layout is not stable by default

Rust's default representation (`repr(Rust)`) does *not* promise stable field order, padding, or enum layout across compiler versions and targets. Therefore, for types shared with C you must opt into explicit representations.

24.2.2 `#[repr(C)]` for structs

`#[repr(C)]` makes a struct layout compatible with what a platform C compiler expects for a corresponding C `struct`.

```
use std::ffi::{c_char, c_int};

#[repr(C)]
pub struct CPoint {
    pub x: c_int,
    pub y: c_int,
}

#[repr(C)]
pub struct CName {
    /* C string pointer: points to NUL-terminated bytes */
    pub ptr: *const c_char,
}
```

Struct rules for FFI

- Only include **FFI-safe fields**: integers of fixed width, `*const T`/`*mut T`, `f32`/`f64`, other `repr(C)` structs.

- Avoid bool across FFI unless you control both sides and document the representation.
- Never place a Rust reference (&T, &mut T) in an FFI struct: references have Rust-specific validity and aliasing requirements.
- Be careful with usize/isize: they are pointer-sized and match C size_t/ssize_t, but this must be intended.

24.2.3 #[repr(C)] for enums

C enums are typically integer constants. Rust enums can carry payloads and have layout optimizations. When interoperating with C:

- Use a **fieldless** enum with an explicit integer representation, or
- Use a tagged-union pattern explicitly modeled as repr(C) (advanced).

A safe and common pattern for error codes:

```
#[repr(i32)]
pub enum CStatus {
    Ok = 0,
    NullArg = 1,
    OutOfMemory = 2,
    InvalidUtf8 = 3,
    InternalError = 1000,
}
```

24.2.4 #[repr(transparent)] for newtype wrappers

When you want a Rust type to have the exact ABI/layout of a single field (often for handle types or strongly-typed IDs), use repr(transparent):

```
use std::ffi::c_int;

#[repr(transparent)]
#[derive(Copy, Clone)]
pub struct FileDesc(pub c_int);
```

24.2.5 Opaque types and handles

Many C libraries expose opaque pointers:

```
/* C header style */
typedef struct foo foo_t;
foo_t* foo_new(void);
void   foo_free(foo_t*);
```

In Rust, model this as an opaque `repr(C)` zero-sized type and use raw pointers to it:

```
#[repr(C)]
pub struct foo_t {
    _private: [u8; 0],
}

extern "C" {
    fn foo_new() -> *mut foo_t;
    fn foo_free(p: *mut foo_t);
}
```

Important: the opaque Rust type never gets constructed. It only exists to type-check pointers.

24.2.6 Strings: `char*`, `CString`, and `CStr`

C strings are NUL-terminated byte sequences. Rust provides:

- CString: owned C string (guarantees exactly one terminating NUL and no interior NUL bytes).
- CStr: borrowed view of a NUL-terminated sequence.

Passing a string from Rust to C:

```
use std::ffi::{CString, CStr};
use std::ffi::c_char;

extern "C" {
    fn puts(s: *const c_char) -> i32;
}

pub fn print_line(s: &str) -> Result<(), ()> {
    let cs = CString::new(s).map_err(|_| ())?;
    unsafe {
        puts(cs.as_ptr());
    }
    Ok(())
}
```

Reading a borrowed string from C (no ownership transfer):

```
use std::ffi::{CStr, c_char};

pub unsafe fn read_c_str(p: *const c_char) -> Result<String, ()> {
    if p.is_null() {
        return Err(());
    }
    let s = CStr::from_ptr(p);
    s.to_str().map(|x| x.to_owned()).map_err(|_| ())
}
```

Ownership warning: never free C-allocated memory using Rust deallocators, and never free Rust-allocated memory using C `free`, unless both sides explicitly agree on the same allocator and the API is designed for it.

24.3 Ownership across language boundaries

FFI correctness is primarily an ownership contract problem. The best FFI layer is explicit about:

- **Who allocates?**
- **Who frees?**
- **What allocator is used?**
- **What is the lifetime and aliasing model?**

24.3.1 Rule 1: One side owns allocation and also provides the deallocation function

If C allocates memory, C must provide a `free`-like function in the same library:

```
char* lib_make_message(void);  /* caller must call lib_free */
void  lib_free(void* p);
```

Rust wrapper:

```
use std::ffi::{CStr, c_char, c_void};

extern "C" {
    fn lib_make_message() -> *mut c_char;
    fn lib_free(p: *mut c_void);
}
```

```
pub fn make_message() -> Result<String, ()> {
    unsafe {
        let p = lib_make_message();
        if p.is_null() {
            return Err(());
        }
        /* Borrow as CStr, copy into Rust-owned String, then free via C */
        let msg = CStr::from_ptr(p).to_str().map_err(|_| ())?.to_owned();
        lib_free(p as *mut c_void);
        Ok(msg)
    }
}
```

24.3.2 Rule 2: If Rust allocates and C must free, export a Rust deallocator

A safe pattern is to expose paired functions:

```
use std::ffi::{CString, c_char};
use std::ptr;

#[no_mangle]
pub extern "C" fn rust_string_new(s: *const c_char) -> *mut c_char {
    /* returns a Rust-allocated C string that C must free using rust_string_free */
    unsafe {
        if s.is_null() {
            return ptr::null_mut();
        }
        let in_s = std::ffi::CStr::from_ptr(s);
        match CString::new(in_s.to_bytes()) {
            Ok(cs) => cs.into_raw(), /* transfers ownership to caller */
            Err(_) => ptr::null_mut(),
        }
    }
}

#[no_mangle]
pub extern "C" fn rust_string_free(p: *mut c_char) {
    unsafe {
        if p.is_null() {
            return;
        }
        /* Rebuild and drop: frees using Rust allocator */
        drop(CString::from_raw(p));
    }
}
```

Contract: `rust_string_free` must only be called on pointers returned by `rust_string_new` (or equivalent Rust allocator functions).

24.3.3 Rule 3: Prefer opaque handles over sharing complex ownership graphs

Avoid sharing:

- Rust containers (`Vec`, `String`, `HashMap`) across FFI,
- trait objects, generics, or lifetimes across FFI,
- structs that contain pointers to internal buffers unless you define full invariants.

Instead, expose opaque handles and explicit operations:

- create/destroy,
- query size/capacity,
- get/set by index with bounds checks,
- copy into user-provided buffers.

24.3.4 Rule 4: Make aliasing explicit

In C, aliasing rules are weaker and code may keep multiple pointers to the same object. In Rust, aliasing is central to soundness. Your wrapper must enforce:

- Either expose only `*mut T` and treat all calls as potentially aliased (unsafe),
- Or enforce unique access by creating safe Rust objects that ensure no concurrent mutable aliasing.

24.3.5 Rule 5: Threading and reentrancy must be stated

Many C libraries are:

- not thread-safe unless you create separate contexts,
- thread-safe only after global initialization,
- reentrant only for some functions.

Reflect this in your Rust types using `Send`/`Sync` decisions:

- If the underlying handle is not thread-safe, do **not** implement `Send` or `Sync`.
- If it is thread-safe by contract, you may implement them, but document why.

24.3.6 Rule 6: Errors cross the boundary as data, not as panics

Common patterns:

- Return integer status codes and write output into out-parameters.
- Return nullable pointers (`NULL` means error).
- Provide a function to get last error message (thread-local if needed).

A robust Rust-to-C boundary with panic containment:

```
use std::ffi::{c_int};
use std::panic::{catch_unwind, AssertUnwindSafe};

#[repr(i32)]
pub enum CStatus {
    Ok = 0,
    Panic = 1000,
}

#[no_mangle]
pub extern ”C” fn do_work(out: *mut c_int) -> CStatus {
    if out.is_null() {
        return CStatus::Panic;
    }

    let r = catch_unwind(AssertUnwindSafe(|| {
        /* must not unwind across C ABI */
        let v: i32 = 40 + 2;
        unsafe { *out = v; }
    }));

    match r {
        Ok(()) => CStatus::Ok,
        Err(_) => CStatus::Panic,
    }
}
```

24.4 Lab: wrapping a C library safely

This lab walks through wrapping a tiny C library that manages an opaque calculator context. The C API is intentionally minimal but includes the patterns found in real libraries: opaque handles, error codes, out-parameters, and explicit destroy functions.

24.4.1 Step 1: The C API

Assume the library provides this header (`calc.h`):

```
#ifndef CALC_H
#define CALC_H

#include <stddef.h>

#ifdef __cplusplus
extern "C" {
#endif

typedef struct calc calc_t;

typedef enum calc_status {
    CALC_OK = 0,
    CALC_NULL = 1,
    CALC_OVERFLOW = 2
} calc_status_t;

calc_t*        calc_new(void);
void           calc_free(calc_t* c);

calc_status_t  calc_add(calc_t* c, int a, int b, int* out);
calc_status_t  calc_accumulate(calc_t* c, int v);
calc_status_t  calc_total(calc_t* c, int* out);

#ifdef __cplusplus
}
#endif
```

```
#endif
```

Behavior contract (typical C style):

- `calc_new` returns `NULL` on allocation failure.
- All functions return a `calc_status_t`.
- `calc_free(NULL)` is allowed (no-op).
- `out` pointers must be non-null for functions that write results.
- Overflow is reported as `CALC_OVERFLOW`; the function does not write to `out` on overflow.

24.4.2 Step 2: Raw Rust bindings

We first write the minimal *unsafe* binding layer. This layer should be as small as possible and very close to the C API.

```rust
use std::ffi::{c_int};

#[repr(C)]
pub struct calc_t {
    _private: [u8; 0],
}

#[repr(i32)]
#[derive(Copy, Clone, Debug, PartialEq, Eq)]
pub enum calc_status_t {
    CALC_OK = 0,
    CALC_NULL = 1,
    CALC_OVERFLOW = 2,
}
```

```
extern "C" {
    pub fn calc_new() -> *mut calc_t;
    pub fn calc_free(c: *mut calc_t);

    pub fn calc_add(c: *mut calc_t, a: c_int, b: c_int, out: *mut c_int) -> calc_status_t;
    pub fn calc_accumulate(c: *mut calc_t, v: c_int) -> calc_status_t;
    pub fn calc_total(c: *mut calc_t, out: *mut c_int) -> calc_status_t;
}
```

24.4.3 Step 3: Define safe Rust errors

Map C status codes to a Rust error type that is ergonomic and preserves meaning:

```
#[derive(Debug)]
pub enum CalcError {
    Null,
    Overflow,
    Unknown,
}

impl From<calc_status_t> for Result<(), CalcError> {
    fn from(s: calc_status_t) -> Self {
        match s {
            calc_status_t::CALC_OK => Ok(()),
            calc_status_t::CALC_NULL => Err(CalcError::Null),
            calc_status_t::CALC_OVERFLOW => Err(CalcError::Overflow),
        }
    }
}
```

24.4.4 Step 4: Create a safe RAII wrapper

The goal: a Rust type that:

- owns the C handle,
- frees it automatically,
- prevents use-after-free by construction,
- exposes safe methods that validate arguments and convert errors.

```
use std::ptr;

pub struct Calc {
    raw: *mut calc_t,
}

impl Calc {
    pub fn new() -> Result<Self, CalcError> {
        let p = unsafe { calc_new() };
        if p.is_null() {
            return Err(CalcError::Null);
        }
        Ok(Self { raw: p })
    }

    pub fn add(&self, a: i32, b: i32) -> Result<i32, CalcError> {
        let mut out: i32 = 0;
        let st = unsafe { calc_add(self.raw, a, b, &mut out as *mut i32) };
        match st {
            calc_status_t::CALC_OK => Ok(out),
            _ => Result::<(), CalcError>::from(st).and_then(|_| unreachable!()),
        }
    }

    pub fn accumulate(&mut self, v: i32) -> Result<(), CalcError> {
        let st = unsafe { calc_accumulate(self.raw, v) };
        Result::<(), CalcError>::from(st)
    }

    pub fn total(&self) -> Result<i32, CalcError> {
        let mut out: i32 = 0;
        let st = unsafe { calc_total(self.raw, &mut out as *mut i32) };
        match st {
            calc_status_t::CALC_OK => Ok(out),
            _ => Result::<(), CalcError>::from(st).and_then(|_| unreachable!()),
        }
    }

    pub fn as_ptr(&self) -> *mut calc_t {
        self.raw
    }
}

impl Drop for Calc {
    fn drop(&mut self) {
        unsafe { calc_free(self.raw) };
        self.raw = ptr::null_mut();
    }
}
```

Soundness notes

This wrapper is safe if:

- The C library honors its contract about `calc_free` and all function signatures.

- The handle can be shared immutably (`&self`) without internal mutation that violates thread safety assumptions.

If the C library is not thread-safe, you should keep `Calc` *not* `Send`/`Sync` by default (raw pointers typically prevent auto-derivation of `Send`/`Sync` unless wrapped). If you *know* it is thread-safe, you may add explicit unsafe impls with a strong justification:

```
/* Only if the C library documents thread-safety for calc_t */
unsafe impl Send for Calc {}
unsafe impl Sync for Calc {}
```

24.4.5 Step 5: Add a higher-level safe API (optional)

A more idiomatic interface can combine operations while maintaining explicit error paths:

```
impl Calc {
    pub fn accumulate_many<I>(&mut self, it: I) -> Result<(), CalcError>
    where
        I: IntoIterator<Item = i32>,
    {
        for v in it {
            self.accumulate(v)?;
        }
        Ok(())
    }

    pub fn checked_add_then_accumulate(&mut self, a: i32, b: i32) -> Result<(), CalcError> {
        let s = self.add(a, b)?;
        self.accumulate(s)?;
        Ok(())
    }
}
```

24.4.6 Step 6: Testing the wrapper

Even for small wrappers, tests should:

- validate null checks (simulate `calc_new` failure if possible),
- validate overflow paths,
- validate that `Drop` frees resources (often by observing resource counters in a test build),
- fuzz C boundary inputs if the library is exposed to untrusted data.

Example unit test skeleton:

```
#[test]
fn basic_flow() {
    let mut c = Calc::new().unwrap();
    assert_eq!(c.add(1, 2).unwrap(), 3);
    c.accumulate_many([10, 20, 30]).unwrap();
    let t = c.total().unwrap();
    let _ = t; /* validate based on C implementation */
}
```

24.4.7 Step 7: Header generation and binding generation (practical notes)

In real projects:

- Use **bindgen** to generate Rust declarations from C headers (reduces drift).
- Use **cbindgen** (or manual headers) to generate C headers for Rust exports.
- Keep the **raw FFI module** separate from the **safe wrapper module**.
- Pin or test compiler/library versions and validate ABI in CI when shipping binaries.

24.5 Summary: a mental checklist for safe C FFI

Before you consider an FFI boundary safe, ensure you can answer all of the following:

- ABI: Is every exported/imported function marked with the correct `extern "..."` ABI?
- Layout: Are all shared structs/enums annotated with `repr(C)` (or `repr(transparent)`) and restricted to FFI-safe fields?
- Ownership: For each pointer, who allocates, who frees, and via which function?
- Lifetimes: Are borrowed pointers valid for the duration of the call? Are returned pointers tied to a handle lifetime?
- Aliasing: Can two pointers alias the same object? Does the API require unique mutable access?
- Panics: Is the boundary panic-proof (no unwinding through `"C"`)?
- Errors: Are failures communicated as explicit values (status codes / null pointers), never as exceptions or panics?
- Threads: Is the handle thread-safe? If not, is your Rust wrapper preventing `Send`/`Sync`?

Interfacing with C++

Rust can interoperate with C++ effectively, but unlike C, C++ is **not** a stable ABI language. C++ interoperability therefore requires deliberate design choices: selecting a stable ABI surface, controlling name mangling, containing exceptions, and constructing a safe boundary that respects Rust's aliasing and lifetime rules. This chapter focuses on *production-grade* Rust–C++ interop patterns used in systems software, engines, embedded runtimes, and performance-critical libraries.

ABI, name mangling, exceptions

25.0.1 Why C++ has no stable ABI

Unlike C, C++ does not define a single universal ABI. ABI details depend on:

- the compiler (GCC, Clang, MSVC),
- the standard library implementation (libstdc++, libc++, MSVC STL),
- compiler flags (RTTI, exceptions, alignment, calling conventions),
- target platform and OS.

Consequences:

- C++ symbols are name-mangled in compiler-specific ways,
- class layout, vtables, and exception handling mechanisms vary,
- binaries compiled with different toolchains are often incompatible.

Therefore, **Rust should never bind directly to arbitrary C++ class layouts or templates**. Instead, interoperability must be mediated through a stable ABI layer.

25.0.2 Name mangling and symbol visibility

C++ encodes function names, namespaces, parameter types, and overloads into mangled symbols. For example, the C++ function:

```
namespace math {
    int add(int a, int b);
}
```

may produce a mangled symbol such as:

```
_ZN4math3addEii
```

Rust cannot rely on such symbols. The solution is to expose **C ABI wrappers** from the C++ side:

```
extern "C" int math_add(int a, int b) {
    return math::add(a, b);
}
```

This wrapper:

- disables name mangling,
- uses the C calling convention,
- produces a stable symbol usable by Rust.

25.0.3 Calling conventions and extern "C"

Rust interoperates with C++ through the "C" ABI:

```
extern "C" {
    fn math_add(a: i32, b: i32) -> i32;
}
```

Important constraints:

- Only plain data types with C-compatible layout may cross the boundary,
- No references, templates, classes, or STL types,
- No exceptions may propagate across this boundary.

25.0.4 Exception handling across the boundary

C++ exceptions and Rust panics are **incompatible mechanisms**.
Rules:

- A C++ exception must never cross into Rust,
- A Rust panic must never unwind into C++,
- All error propagation across the boundary must be explicit.

C++ side: catch all exceptions at the ABI boundary.

```
extern "C" int safe_divide(int a, int b, int* out) {
    try {
        if (!out) return -1;
        *out = a / b;
        return 0;
    } catch (...) {
        return -2;
    }
}
```

Rust side: treat all foreign calls as `unsafe` and error-coded.

```
extern "C" {
    fn safe_divide(a: i32, b: i32, out: *mut i32) -> i32;
}
```

25.0.5 RTTI, destructors, and object lifetime

C++ object destruction depends on:

- virtual destructors,
- RTTI and vtables,
- allocator symmetry.

Rust must never:

- allocate a C++ object and destroy it itself,
- invoke C++ destructors implicitly,
- assume layout compatibility with C++ classes.

All C++ object lifetimes must be controlled via explicit C-style functions.

Designing safe boundaries

The core principle of Rust–C++ interop is:

> **Use C as the ABI, not C++.**

25.0.6 The three-layer model

A robust architecture separates concerns into three layers:

1. **C++ implementation layer** Full C++ features, classes, templates, RAII, exceptions.
2. **C ABI facade layer** Flat functions, opaque handles, error codes, no exceptions.
3. **Rust safe wrapper layer** Ownership, lifetimes, RAII, error types, thread-safety.

25.0.7 Opaque handles instead of classes

C++ side:

```
class Engine {
public:
    void push(int v);
    int total() const;
};
```

Expose only opaque handles:

```
extern "C" {

struct engine_handle;

engine_handle* engine_new();
void engine_free(engine_handle*);

int engine_push(engine_handle*, int);
int engine_total(engine_handle*, int* out);

}
```

Implementation:

```
struct engine_handle {
    Engine impl;
};
```

25.0.8 Error modeling

Avoid:

- throwing exceptions across FFI,
- returning sentinel values without documentation.

Preferred patterns:

- explicit status codes,
- out-parameters for results,
- separate error retrieval functions if needed.

Rust wrapper:

```
#[derive(Debug)]
pub enum EngineError {
    Null,
    Internal,
}

impl EngineError {
    fn from_code(c: i32) -> Result<(), Self> {
        match c {
            0 => Ok(()),
            -1 => Err(EngineError::Null),
            _ => Err(EngineError::Internal),
        }
    }
}
```

25.0.9 Ownership and aliasing

Rules:

- C++ allocates ⇒ C++ frees,
- Rust allocates ⇒ Rust frees,
- raw pointers crossing the boundary are never assumed unique.

Rust wrapper enforces safety:

```
pub struct Engine {
    raw: *mut engine_handle,
}

impl Drop for Engine {
    fn drop(&mut self) {
        unsafe { engine_free(self.raw) };
    }
}
```

25.0.10 Thread-safety contracts

C++ libraries often have undocumented threading assumptions.
Rust must encode them explicitly:

- if not thread-safe: do nothing (not Send/Sync),

- if thread-safe by contract: add unsafe impls with justification.

```
/* Only if documented thread-safe */
unsafe impl Send for Engine {}
unsafe impl Sync for Engine {}
```

Lab: consuming a Rust library from C++

This lab demonstrates exporting a Rust library with a stable C ABI and consuming it safely from C++.

25.0.11 Step 1: Define a Rust library crate

```
[lib]
crate-type = ["cdylib"]
```

25.0.12 Step 2: Rust API with explicit ABI

```
use std::ffi::c_int;
use std::panic::{catch_unwind, AssertUnwindSafe};

#[repr(i32)]
pub enum Status {
    Ok = 0,
    Panic = 1,
}

#[no_mangle]
pub extern "C" fn rust_add(a: c_int, b: c_int, out: *mut c_int) -> Status {
    if out.is_null() {
        return Status::Panic;
    }
    let r = catch_unwind(AssertUnwindSafe(|| {
        unsafe { *out = a + b; }
    }));

    match r {
        Ok(_) => Status::Ok,
        Err(_) => Status::Panic,
    }
}
```

Key properties:

- no name mangling,
- no unwinding across the boundary,
- explicit error signaling.

25.0.13 Step 3: C-compatible header

Manually written header for C++:

```
#ifndef RUST_API_H
#define RUST_API_H

#ifdef __cplusplus
extern "C" {
#endif

typedef enum Status {
    Status_Ok = 0,
    Status_Panic = 1
} Status;
Status rust_add(int a, int b, int* out);

#ifdef __cplusplus
}
#endif

#endif
```

25.0.14 Step 4: Consuming from C++

```
#include "rust_api.h"
#include <iostream>

int main() {
    int result = 0;
    Status s = rust_add(40, 2, &result);

    if (s == Status_Ok) {
        std::cout << "Result = " << result << "\n";
    } else {
        std::cerr << "Rust error\n";
    }
}
```

25.0.15 Step 5: Linking

- Build Rust as a shared library,
- Link against it from C++,
- Ensure runtime loader can locate the library.

25.0.16 What this lab demonstrates

- Rust is safely consumable from C++ when using a C ABI,
- exceptions and panics are fully contained,
- ownership and lifetime are explicit,
- the boundary is testable and auditable.

Key takeaways

- C++ is not a stable ABI language; never bind directly to its internals.
- Use `extern "C"` as the only supported Rust–C++ boundary.
- Catch exceptions and panics at the boundary and convert them to data.
- Design the ABI as if it were a network protocol: minimal, explicit, and versioned.
- Wrap all unsafe foreign calls behind safe Rust abstractions.

Systems Programming with Rust

Systems programming is the art of writing software that speaks directly to the operating system and hardware-adjacent abstractions: files, processes, sockets, memory maps, and low-level performance tradeoffs. Rust is especially well-suited here because it combines:

- direct access to OS primitives (via `std` and platform APIs),
- explicit control over allocation and data layout,
- a type system that makes whole classes of memory and concurrency bugs structurally harder to write.

This chapter is practical and example-driven: first we build mental models for file I/O, sockets, and memory-mapped I/O; then we connect classic C failure modes to Rust solutions; finally we implement a lab project that exercises real OS interfaces with performance in mind.

Files, sockets, memory-mapped I/O

26.0.1 Files: descriptors, handles, and the OS contract

A file in systems programming is not merely "a sequence of bytes"; it is a kernel object accessed through a handle:

- POSIX: file descriptor (`int fd`)

- Windows: handle (`HANDLE`)

Rust standard library models these resources with RAII: objects close themselves automatically.

26.0.2 Basic file I/O with `std::fs`

The simplest pattern: open, read, process, write.

```rust
use std::fs::File;
use std::io::{self, Read, Write, BufReader, BufWriter};

fn copy_file(src: &str, dst: &str) -> io::Result<u64> {
    let in_f  = File::open(src)?;
    let out_f = File::create(dst)?;

    let mut r = BufReader::new(in_f);
    let mut w = BufWriter::new(out_f);

    let mut buf = [0u8; 64 * 1024];
    let mut total: u64 = 0;

    loop {
        let n = r.read(&mut buf)?;
        if n == 0 { break; }
        w.write_all(&buf[..n])?;
        total += n as u64;
    }

    w.flush()?;
    Ok(total)
}
```

Key systems notes:

- Reads and writes can be partial; use `read` in a loop and `write_all`.
- Buffering reduces syscall overhead.
- Error handling is explicit and composable via `Result`.

26.0.3 Random access and file positioning

Random access relies on a file offset maintained by the kernel for each open handle.

```
use std::fs::File;
use std::io::{self, Read, Seek, SeekFrom};

fn read_at(path: &str, offset: u64, n: usize) -> io::Result<Vec<u8>> {
    let mut f = File::open(path)?;
    f.seek(SeekFrom::Start(offset))?;

    let mut buf = vec![0u8; n];
    let got = f.read(&mut buf)?;
    buf.truncate(got);
    Ok(buf)
}
```

Important:

- A shared file handle with multiple threads needs coordination if the file offset is shared.
- Prefer independent handles per thread or OS primitives for positioned I/O when available.

26.0.4 Metadata: size, type, permissions

File metadata is a performance and correctness tool, especially for indexing and walking directories.

```rust
use std::fs;
use std::io;

fn print_meta(path: &str) -> io::Result<()> {
    let m = fs::metadata(path)?;
    println!("is_file={} is_dir={} len={}",
             m.is_file(), m.is_dir(), m.len());
    Ok(())
}
```

26.0.5 Directory walking at scale

Directory traversal is mostly syscall-bound; you need to minimize allocations and format work.

```rust
use std::fs;
use std::io;
use std::path::{Path, PathBuf};

fn walk(root: &Path, out: &mut Vec<PathBuf>) -> io::Result<()> {
    let mut stack = vec![root.to_path_buf()];

    while let Some(dir) = stack.pop() {
        for e in fs::read_dir(&dir)? {
            let e = e?;
            let p = e.path();
            let ft = e.file_type()?;
            if ft.is_dir() {
                stack.push(p);
            } else if ft.is_file() {
                out.push(p);
            }
        }
    }
    Ok(())
}
```

Engineering notes:

- Avoid recursion to prevent deep stack usage.
- Consider filtering by extension early to reduce work.
- For huge trees, streaming results (write to output as you go) is better than storing everything.

26.0.6 Sockets: blocking vs non-blocking, streams vs datagrams

Sockets are kernel objects representing network endpoints.

- TCP: stream semantics, reliable, ordered.
- UDP: datagram semantics, message boundaries preserved, unreliable.

Rust provides blocking sockets in `std::net`. For non-blocking high-scale servers you typically use an async runtime; however, understanding the blocking model is foundational.

26.0.7 TCP client and server (blocking)

Server:

```
use std::io::{self, Read, Write};
use std::net::{TcpListener, TcpStream};
use std::thread;

fn handle(mut s: TcpStream) -> io::Result<()> {
    let mut buf = [0u8; 4096];
    loop {
        let n = s.read(&mut buf)?;
        if n == 0 { break; }
        s.write_all(&buf[..n])?;
    }
    Ok(())
}

fn main() -> io::Result<()> {
    let lis = TcpListener::bind(”127.0.0.1:9000”)?;
    for c in lis.incoming() {
        let s = c?;
        thread::spawn(move || {
            let _ = handle(s);
        });
    }
    Ok(())
}
```

Client:

```
use std::io::{self, Read, Write};
use std::net::TcpStream;

fn main() -> io::Result<()> {
    let mut s = TcpStream::connect(”127.0.0.1:9000”)?;
    s.write_all(b”hello”)?;
    s.shutdown(std::net::Shutdown::Write)?;

    let mut r = Vec::new();
    s.read_to_end(&mut r)?;
    println!(”{:?}”, r);
    Ok(())
}
```

Systems notes:

- TCP reads are not message-oriented: you must define your own framing protocol.
- Always handle partial reads/writes and `EINTR`-like interruptions (Rust converts these to retriable errors on many platforms, but you must still code robustly).

26.0.8 Memory-mapped I/O: when and why

Memory mapping maps a file's contents into a process address space:

- Reads can become page faults + memory loads rather than explicit `read` syscalls.
- The OS page cache backs the mapping, enabling fast random access.
- You can avoid copying bytes into user buffers for some workloads.

Typical use-cases:

- fast indexing and scanning of large files,
- random lookups (offset-based formats),
- shared read-only data across processes.

Caveats:

- Accessing unmapped pages triggers faults; poor locality can be slower than buffered streaming.
- Mapping extremely large files can stress address space on 32-bit systems.
- File truncation or concurrent modification can invalidate assumptions (and in some OSes may cause SIGBUS-like faults).

26.0.9 Memory mapping in Rust

The standard library does not expose portable mmap directly, so system programming uses platform APIs (unsafe) or a well-audited crate. The conceptual pattern is always:

- open file,
- map region,
- treat mapping as &[u8] or &mut [u8],
- unmap on drop.

Below is a minimal POSIX mapping example to show the mechanics. This is unsafe and platform-specific.

```
use std::fs::File;
use std::io;
use std::os::fd::AsRawFd;
use std::ptr;

type c_void = core::ffi::c_void;
type size_t = usize;
type off_t = isize;
type c_int = i32;

const PROT_READ: c_int = 0x1;
const MAP_PRIVATE: c_int = 0x02;

extern "C" {
    fn mmap(addr: *mut c_void,
            len: size_t,
            prot: c_int,
            flags: c_int,
            fd: c_int,
            off: off_t) -> *mut c_void;

    fn munmap(addr: *mut c_void, len: size_t) -> c_int;
}

pub struct Mmap {
    p: *mut u8,
    len: usize,
}

impl Mmap {
    pub fn map_readonly(f: &File, len: usize) -> io::Result<Self> {
        let fd = f.as_raw_fd();
        unsafe {
            let r = mmap(ptr::null_mut(), len, PROT_READ, MAP_PRIVATE, fd, 0);
            if r as isize == -1 {
                return Err(io::Error::last_os_error());
            }
            Ok(Self { p: r as *mut u8, len })
        }
    }

    pub fn as_bytes(&self) -> &[u8] {
        unsafe { std::slice::from_raw_parts(self.p as *const u8, self.len) }
    }
}

impl Drop for Mmap {
    fn drop(&mut self) {
        unsafe {
            if !self.p.is_null() && self.len != 0 {
                let _ = munmap(self.p as *mut c_void, self.len);
            }
        }
    }
}
```

Safety boundary: the mapping object ensures unmap on drop, and provides a slice view. But correctness still depends on:

- mapping length matching file size constraints,
- file not being truncated in ways that violate access assumptions,
- not using the slice after drop.

Classic C systems bugs and Rust solutions

Classic systems bugs arise from undefined behavior, unchecked assumptions, and manual lifetime management. Rust does not eliminate all bugs, but it changes the default failure mode: many memory and aliasing errors become compile-time failures or are forced into narrow `unsafe` regions.

26.0.10 Use-after-free

C pattern:

```
char* p = malloc(16);
free(p);
p[0] = 'X'; /* use-after-free */
```

Rust prevents this in safe code because ownership moves and drops invalidate access:

```
fn demo() {
    let v = vec![0u8; 16];
    drop(v);
    /* v[0] = 1; */ /* compile error: use of moved value */
}
```

If you *choose* to do raw-pointer work, Rust forces it into `unsafe`, making review focused.

26.0.11 Double free

C pattern:

```
void* p = malloc(32);
free(p);
free(p); /* double free */
```

Rust RAII + move semantics prevent double-drop in safe code.

```
fn safe() {
    let v = vec![0u8; 32];
    /* drop(v); drop(v); */ /* compile error: moved value */
}
```

26.0.12 Buffer overflow / out-of-bounds

C pattern:

```
char buf[8];
strcpy(buf, "this string is too long"); /* overflow */
```

Rust slices know their length; indexing checks bounds:

```
fn bounds() {
    let a = [0u8; 8];
    /* let x = a[999]; */ /* panic (checked) rather than UB */
    let _ = a.get(999); /* None */
}
```

For high-performance code, you can choose `get_unchecked` in `unsafe`, but that is an explicit opt-in.

26.0.13 Uninitialized memory

C pattern:

```
int x;
printf("%d\n", x); /* uninitialized read */
```

Rust requires definite initialization:

```
fn init() {
    /* let x: i32; println!("{}", x); */ /* compile error */
    let x: i32 = 0;
    println!("{}", x);
}
```

26.0.14 Integer overflow pitfalls

C signed overflow is undefined behavior; C unsigned overflow wraps. In Rust:

- debug builds check and panic on overflow for many integer ops,
- release builds wrap for two's complement targets, but you should use explicit APIs for intent.

```
fn overflow_examples(a: u32, b: u32) {
    let _wrap = a.wrapping_add(b);
    let _sat  = a.saturating_add(b);
    let _chk  = a.checked_add(b);  /* Option<u32> */
    let _ovf  = a.overflowing_add(b); /* (u32, bool) */
}
```

Systems code should encode the intended arithmetic semantics explicitly.

26.0.15 Data races and thread unsafety

In C, data races are easy to write and often non-reproducible. Rust's type system requires thread-safety traits and forbids shared mutable aliasing in safe code.

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

fn threaded_counter() {
    let c = Arc::new(Mutex::new(0u64));

    let mut th = Vec::new();
    for _ in 0..4 {
        let c2 = Arc::clone(&c);
        th.push(thread::spawn(move || {
            for _ in 0..1000 {
                *c2.lock().unwrap() += 1;
            }
        }));
    }

    for t in th { t.join().unwrap(); }
    println!("{}", *c.lock().unwrap());
}
```

Rust does not magically make logic correct, but it removes huge classes of undefined behavior and accidental races by default.

26.0.16 Resource leaks and error paths

C systems code often leaks resources on early returns.
Rust's Drop ensures deterministic cleanup on all paths:

```rust
use std::fs::File;
use std::io;

fn open_two(a: &str, b: &str) -> io::Result<(File, File)> {
    let fa = File::open(a)?;
    let fb = File::open(b)?;
    Ok((fa, fb))
}
```

If the second open fails, the first file handle is automatically closed.

Lab: mini shell or fast file indexer

This lab gives you two project options. Both are classic systems exercises that force you to engage with:

- process spawning,
- I/O redirection,
- directory walking and metadata,
- performance bottlenecks and measurement,
- robust error handling.

26.0.17 Option A: Mini shell (processes, pipes, redirection)

Goal: implement a minimal interactive shell with:

- command parsing (basic whitespace splitting),
- builtins: `cd`, `exit`,
- external commands using `std::process::Command`,
- optional output redirection: `cmd > file`.

Core loop

```
use std::io::{self, Write};
use std::process::{Command, Stdio};
use std::fs::File;
use std::env;

fn main() -> io::Result<()> {
    let stdin = io::stdin();
    loop {
        print!("rshell> ");
        io::stdout().flush()?;

        let mut line = String::new();
        if stdin.read_line(&mut line)? == 0 {
            break;
        }
        let line = line.trim();
        if line.is_empty() {
            continue;
        }
        if line == "exit" {
            break;
        }

        if let Some(rest) = line.strip_prefix("cd ") {
            let target = rest.trim();
            if let Err(e) = env::set_current_dir(target) {
                eprintln!("cd: {}", e);
            }
            continue;
        }

        /* Parse optional redirection: cmd ... > file */
        let mut parts: Vec<&str> = line.split_whitespace().collect();
        let mut redirect: Option<&str> = None;

        if parts.len() >= 3 {
            let n = parts.len();
            if parts[n - 2] == ">" {
                redirect = Some(parts[n - 1]);
                parts.truncate(n - 2);
            }
        }

        let prog = parts[0];
        let args = &parts[1..];

        let mut cmd = Command::new(prog);
        cmd.args(args);

        if let Some(path) = redirect {
            let f = File::create(path)?;
            cmd.stdout(Stdio::from(f));
        }

        match cmd.status() {
            Ok(st) => {
                if !st.success() {
                    eprintln!("exit status: {}", st);
                }
            }
            Err(e) => eprintln!("exec error: {}", e),
        }
    }

    Ok(())
}
```

Extensions:

- add `2> file` for stderr redirection,
- add pipelines `cmd1 | cmd2` using `Stdio::piped()`,
- improve parsing to handle quotes.

26.0.18 Option B: Fast file indexer (I/O, metadata, scanning)

Goal: build a tool that walks a directory tree and produces an index:

- path,
- size,
- last modified timestamp (platform dependent precision),
- fast content hash (optional).

Index structure

```
use std::path::PathBuf;

#[derive(Debug)]
pub struct Entry {
    pub path: PathBuf,
    pub size: u64,
}
```

Fast traversal with streaming output

Instead of storing all results, write as you go:

```
use std::fs;
use std::io::{self, Write};
use std::path::Path;

fn index(root: &Path, mut out: impl Write) -> io::Result<u64> {
    let mut stack = vec![root.to_path_buf()];
    let mut count: u64 = 0;

    while let Some(dir) = stack.pop() {
        for e in fs::read_dir(&dir)? {
            let e = e?;
            let p = e.path();
            let ft = e.file_type()?;

            if ft.is_dir() {
                stack.push(p);
                continue;
            }

            if ft.is_file() {
                let m = e.metadata()?;
                let size = m.len();
                writeln!(out, "{}\t{}", p.display(), size)?;
                count += 1;
            }
        }
    }

    Ok(count)
}
```

Hashing: choose a strategy

Hashing file contents is often the bottleneck. You can:

- avoid hashing for speed and index only metadata,
- hash only first N bytes as a fingerprint,
- hash fully but do it in parallel with a worker pool.

A simple streaming hash skeleton (placeholder function):

```
use std::fs::File;
use std::io::{self, Read, BufReader};

fn hash_file(path: &std::path::Path) -> io::Result<u64> {
    let f = File::open(path)?;
    let mut r = BufReader::new(f);
    let mut buf = [0u8; 64 * 1024];

    /* Example: a tiny rolling hash (not cryptographic) */
    let mut h: u64 = 1469598103934665603;
    loop {
        let n = r.read(&mut buf)?;
        if n == 0 { break; }
        for &b in &buf[..n] {
            h ^= b as u64;
            h = h.wrapping_mul(1099511628211);
        }
    }
    Ok(h)
}
```

If you need cryptographic hashes or modern fast hashes for deduplication, use a well-audited hashing implementation and clearly document the security/performance intent.

26.0.19 Performance checklist (applies to both labs)

- Avoid tiny reads/writes: buffer aggressively.
- Minimize allocations inside hot loops.
- Use iterative traversal (stack) rather than recursion.
- Do less formatting: defer string formatting or write raw fields.
- Measure: time wall-clock, count syscalls, and test on SSD vs HDD.

26.0.20 Safety checklist

- Never assume UTF-8 for paths; treat OS strings carefully.
- Handle permission errors gracefully: skip and continue.
- Avoid TOCTOU traps when using metadata then opening files; if security matters, open first then query via the handle.

- If using `unsafe` (for memory maps or platform APIs), isolate it in a tiny module and document invariants.

Summary

- Files and sockets are kernel objects; performance is often syscall-bound.
- Rust makes resource management deterministic and robust via RAII and `Result`.
- Most classic C memory bugs are structurally prevented in safe Rust, or forced into narrow `unsafe` boundaries.
- Memory-mapped I/O can be a major speed win for random access and indexing, but it requires careful invariants.
- A mini shell or file indexer is an ideal capstone to practice real OS interactions with correctness and speed.

Professional CLI Tools

Professional CLI tools are not just “a main function that reads args”. They are small products: they must be discoverable (`−help`), predictable (stable UX and exit codes), observable (logging and diagnostics), configurable (files + env + CLI precedence), and releasable (repeatable builds, artifacts, checksums, and changelogs). Rust’s ecosystem has converged on a set of reliable building blocks for these concerns, with patterns that scale from a one-file utility to a multi-binary workspace.

27.1 Argument parsing

27.1.1 Design goals for a real CLI

A production CLI should make these decisions explicit:

- **Command topology**: single-command tool vs subcommands (`git`-style).
- **UX contract**: stable flags, stable output formats, stable exit codes.
- **Precedence**: CLI overrides config file overrides environment overrides defaults.
- **Error surface**: validation errors are human-friendly; internal failures have diagnostics.

27.1.2 A modern pattern: derive-based parsing with subcommands

A strongly-typed CLI definition keeps parsing rules next to documentation.

```
use clap::{Parser, Subcommand, Args, ValueEnum};

#[derive(Parser, Debug)]
#[command(name = "procli")]
#[command(version, about = "Production-grade CLI example")]
#[command(propagate_version = true)]
struct Cli {
    /// Increase verbosity (-v, -vv, -vvv)
    #[arg(short, long, action = clap::ArgAction::Count)]
    verbose: u8,

    /// Optional path to a config file (TOML/YAML/JSON by loader choice)
    #[arg(long)]
    config: Option<std::path::PathBuf>,

    /// Select output format for human or tooling
    #[arg(long, value_enum, default_value_t = Output::Human)]
    output: Output,

    #[command(subcommand)]
    cmd: Command,
}

#[derive(Subcommand, Debug)]
enum Command {
    /// Index files under a directory
    Index(IndexArgs),

    /// Show resolved configuration (after precedence)
    ConfigShow,
    /// Generate shell completions/man pages (release engineering)
    Gen(GenArgs),
}

#[derive(Args, Debug)]
struct IndexArgs {
    /// Root directory to scan
    root: std::path::PathBuf,

    /// Filter by extension (repeatable: --ext rs --ext toml)
    #[arg(long, action = clap::ArgAction::Append)]
    ext: Vec<String>,

    /// Max depth (0 = only root). Omit for unlimited.
    #[arg(long)]
    max_depth: Option<usize>,
}

#[derive(Args, Debug)]
struct GenArgs {
    /// Where to write generated artifacts
    #[arg(long)]
    out_dir: std::path::PathBuf,

    /// Which artifacts to generate
    #[arg(long, value_enum, default_value_t = GenWhat::All)]
    what: GenWhat,
}

#[derive(Copy, Clone, Debug, ValueEnum)]
enum Output {
    Human,
    Json,
    Tsv,
}

#[derive(Copy, Clone, Debug, ValueEnum)]
enum GenWhat {
    All,
    Completions,
    Manpages,
}
```

27.1.3 Recommended UX rules

- **Use subcommands** when the tool has distinct modes (`index`, `query`, `serve`).
- **Keep global flags global**: `-v/–verbose`, `–config`, `–output`.
- **Avoid ambiguous positional arguments**. Prefer named options for values that may be optional.
- **Make repeatable flags explicit**: `–ext` repeated, `-v` counted.
- **Prefer enums** for constrained choices (validated with `ValueEnum`).

27.1.4 Validation: fail early with clear messages

Do not let invalid input reach core logic. Validate at parse time where possible, or immediately after parsing.

```
fn validate(cli: &Cli) -> Result<(), String> {
    match &cli.cmd {
        Command::Index(args) => {
            if args.max_depth == Some(0) && !args.root.exists() {
                return Err("root does not exist".into());
            }
            for e in &args.ext {
                if e.is_empty() || e.contains('/') {
                    return Err(format!("invalid extension: {e:?}"));
                }
            }
        }
        _ => {}
    }
    Ok(())
}
```

27.1.5 Exit codes as part of the API

Define and document exit codes; scripts rely on them.

```
#[repr(i32)]
enum ExitCode {
    Ok = 0,
    BadArgs = 2,
    Config = 3,
    Io = 4,
    Internal = 1,
}

fn main() {
    let cli = Cli::parse();
    if let Err(msg) = validate(&cli) {
        eprintln!("error: {msg}");
        std::process::exit(ExitCode::BadArgs as i32);
    }
    if let Err(code) = run(cli) {
        std::process::exit(code as i32);
    }
}
```

27.2 Logging

27.2.1 Why structured logging matters

In production, logs are often consumed by systems (journald, OpenTelemetry pipelines, log aggregation). A robust strategy is:

- Use **structured** events (`key=value`) rather than only formatted strings.
- Use **spans** to represent operations that have a duration (request, indexing run, query).
- Configure **filters** at runtime (environment, config, CLI).

27.2.2 The modern Rust stack: `tracing` + `tracing-subscriber`

Instrumentation is done via `tracing` macros, and output/filtering is installed by a subscriber.

```
use tracing::{info, warn, error, debug, instrument};

#[instrument(level = "info", skip_all, fields(root = %root.display()))]
fn do_index(root: &std::path::Path) -> std::io::Result<u64> {
    info!("starting index");
    let mut count = 0u64;

    for entry in std::fs::read_dir(root)? {
        let entry = entry?;
        let ft = entry.file_type()?;
        if ft.is_file() {
            count += 1;
            debug!(path = %entry.path().display(), "file");
        }
    }

    info!(count, "index complete");
    Ok(count)
}
```

27.2.3 Initialize logging once, early

A good initializer supports:

- human text output for terminals,
- JSON output for production,
- filtering via RUST_LOG-style environment strings,
- optional timestamps and target/module info.

```
use tracing_subscriber::{fmt, EnvFilter};

#[derive(Copy, Clone, Debug)]
enum LogFormat { Human, Json }

fn init_logging(verbose: u8, format: LogFormat) {
    let base = match verbose {
        0 => "info",
        1 => "debug",
        _ => "trace",
    };
    let filter = EnvFilter::try_from_default_env()
        .unwrap_or_else(|_| EnvFilter::new(base));

    let builder = fmt()
        .with_env_filter(filter)
        .with_target(true);

    match format {
        LogFormat::Human => {
            builder.compact().init();
        }
        LogFormat::Json => {
            builder.json().init();
        }
    }
}
```

27.2.4 Bridge legacy `log` users

Many crates still emit `log::info!`. Ensure they show up.

```
/* Cargo.toml: tracing-log = "0.2" */
fn init_log_bridge() {
    let _ = tracing_log::LogTracer::init();
}
```

27.2.5 Operational rules for logging

- **No secrets**: never log tokens, private keys, raw credentials.
- **Use fields**: `info!(user_id, req_id, ...)` beats string concatenation.
- **Make errors explicit**: `error!(error = %e, "message")`.

- **Avoid noisy logs by default**: `info` should be high-value.

27.3 Configuration management

27.3.1 Configuration layering and precedence

A professional CLI typically supports these layers (lowest to highest priority):

1. built-in defaults
2. config file (system/user/project)
3. environment variables
4. command-line arguments

Rule: always provide a way to print the resolved configuration (`config-show`) so users can debug precedence.

27.3.2 Define a typed configuration model

Use a typed struct as the single source of truth for configuration, and keep it separate from the CLI struct.

```
use serde::Deserialize;

#[derive(Debug, Clone, Deserialize)]
struct AppConfig {
    /* logging */
    log_format: Option<String>,     /* "human" | "json" */
    log_filter: Option<String>,     /* "info,mycrate=debug" */
    /* indexing */
    follow_symlinks: Option<bool>,
    max_depth: Option<usize>,
    extensions: Option<Vec<String>>,
}

impl Default for AppConfig {
    fn default() -> Self {
        Self {
            log_format: Some(”human”.into()),
            log_filter: None,
            follow_symlinks: Some(false),
            max_depth: None,
            extensions: Some(vec![]),
        }
    }
}
```

27.3.3 Option A: Layered config with the `config` crate

This approach is straightforward: merge sources and deserialize into your struct.

```
use std::path::PathBuf;

fn load_config(path: Option<PathBuf>) -> Result<AppConfig, String> {
    let mut b = config::Config::builder();

    /* 1) defaults */
    let defaults = AppConfig::default();
    b = b
        .set_default(”log_format”, defaults.log_format.clone().unwrap())?
        .set_default(”follow_symlinks”, defaults.follow_symlinks.unwrap())?
        .set_default(”extensions”, defaults.extensions.clone().unwrap())?;
    /* 2) optional file */
    if let Some(p) = path {
        b = b.add_source(config::File::from(p).required(false));
    } else {
        /* conventional names (optional) */
        b = b.add_source(config::File::with_name("procli").required(false));
    }

    /* 3) env vars: PROCLI_LOG_FORMAT, PROCLI_MAX_DEPTH, ... */
    b = b.add_source(config::Environment::with_prefix("PROCLI").separator("_"));

    let cfg = b.build().map_err(|e| e.to_string())?;
    cfg.try_deserialize::<AppConfig>().map_err(|e| e.to_string())
}
```

27.3.4 Option B: Provenance-aware config with `figment`

When you want better diagnostics about where each value came from, `figment` is a strong choice.

```
use figment::{
    Figment,
    providers::{Serialized, Env, Format, Toml, Json, Yaml},
};
use serde::Serialize;

#[derive(Debug, Clone, Deserialize, Serialize)]
struct AppConfig2 {
    log_format: String,
    follow_symlinks: bool,
    max_depth: Option<usize>,
}
```

```
fn load_config2(path: Option<std::path::PathBuf>) -> Result<AppConfig2, String> {
    let defaults = AppConfig2 {
        log_format: ”human”.into(),
        follow_symlinks: false,
        max_depth: None,
    };

    let mut f = Figment::from(Serialized::defaults(defaults))
        .merge(Env::prefixed(”PROCLI_”));

    if let Some(p) = path {
        if let Some(ext) = p.extension().and_then(|x| x.to_str()) {
            f = match ext {
                ”toml” => f.merge(Toml::file(p)),
                ”json” => f.merge(Json::file(p)),
                ”yaml” | ”yml” => f.merge(Yaml::file(p)),
                _ => return Err(”unsupported config format”.into()),
            };
        }
    }

    f.extract::<AppConfig2>().map_err(|e| e.to_string())
}
```

27.3.5 Overriding config with CLI values

Treat the CLI as the highest-priority layer by applying it last, explicitly.

```
#[derive(Debug, Clone)]
struct EffectiveConfig {
    log_format: LogFormat,
    max_depth: Option<usize>,
    extensions: Vec<String>,
}

fn merge(cli: &Cli, file_cfg: AppConfig) -> Result<EffectiveConfig, String> {
    let log_format = match file_cfg.log_format.as_deref().unwrap_or("human") {
        "human" => LogFormat::Human,
        "json" => LogFormat::Json,
        other => return Err(format!("invalid log_format: {other}")),
    };

    /* Start from config file values */
    let mut max_depth = file_cfg.max_depth;
    let mut extensions = file_cfg.extensions.unwrap_or_default();

    /* Apply CLI overrides */
    if let Command::Index(args) = &cli.cmd {
        if let Some(d) = args.max_depth {
            max_depth = Some(d);
        }
        if !args.ext.is_empty() {
            extensions = args.ext.clone();
        }
    }

    Ok(EffectiveConfig { log_format, max_depth, extensions })
}
```

27.3.6 A dedicated `config-show` command

This is a real-world support tool.

```
fn cmd_config_show(cfg: &EffectiveConfig) {
    /* Keep output stable; scripts may parse it */
    println!("log_format={:?}", cfg.log_format);
    println!("max_depth={:?}", cfg.max_depth);
    println!("extensions={:?}", cfg.extensions);
}
```

Lab: production-ready CLI with releases

This lab builds a shippable CLI called `procli` that:

- parses arguments with subcommands,
- uses structured logging (human and JSON),
- loads layered configuration (file/env/CLI),
- provides stable exit codes,
- ships releases with reproducible automation and generated artifacts.

27.3.7 Project layout

A maintainable layout for a single-binary CLI:

```
procli/
  Cargo.toml
  src/
    main.rs
    cli.rs
    config.rs
    logging.rs
    commands/
      mod.rs
      index.rs
      gen.rs
```

27.3.8 Cargo.toml (baseline dependencies)

```toml
[package]
name = "procli"
version = "0.1.0"
edition = "2021"

[dependencies]
clap = { version = "4", features = ["derive"] }
serde = { version = "1", features = ["derive"] }
config = "0.14"
tracing = "0.1"
tracing-subscriber = { version = "0.3", features = ["env-filter", "json"] }
tracing-log = "0.2"
```

27.3.9 src/main.rs (wiring)

```rust
mod cli;
mod config;
mod logging;
mod commands;

use clap::Parser;

#[repr(i32)]
enum ExitCode {
    Ok = 0,
    BadArgs = 2,
    Config = 3,
    Io = 4,
    Internal = 1,
}
```

```
fn main() {
    let cli = cli::Cli::parse();

    if let Err(msg) = cli::validate(&cli) {
        eprintln!("error: {msg}");
        std::process::exit(ExitCode::BadArgs as i32);
    }

    let file_cfg = match config::load(cli.config.clone()) {
        Ok(c) => c,
        Err(e) => {
            eprintln!("config error: {e}");
            std::process::exit(ExitCode::Config as i32);
        }
    };

    let eff = match config::merge(&cli, file_cfg) {
        Ok(c) => c,
        Err(e) => {
            eprintln!("config error: {e}");
            std::process::exit(ExitCode::Config as i32);
        }
    };

    logging::init(cli.verbose, eff.log_format);
    logging::init_log_bridge();

    let code = match commands::run(cli, eff) {
        Ok(()) => ExitCode::Ok,
        Err(ExitCode::Ok) => ExitCode::Ok,
        Err(c) => c,
    };
    std::process::exit(code as i32);
}
```

27.3.10 Index command: stable output formats

For tooling, define output formats that do not change casually. Example: TSV for pipelines.

```
use std::io::{self, Write};
use tracing::info;

pub enum OutputFmt { Human, Json, Tsv }

pub fn write_index_line(mut out: impl Write, fmt: OutputFmt, path: &str, size: u64) ->
↪  io::Result<()> {
    match fmt {
        OutputFmt::Human => writeln!(out, "{path} ({size} bytes)"),
        OutputFmt::Tsv => writeln!(out, "{path}\t{size}"),
        OutputFmt::Json => {
            /* Keep JSON stable: explicit keys */
            writeln!(out, "{{\"path\":\"{}\",\"size\":{}}}", escape_json(path), size)
        }
    }
}

fn escape_json(s: &str) -> String {
    s.chars().flat_map(|c| match c {
        '\\' => "\\\\".chars().collect::<Vec<_>>(),
        '"'  => "\\\"".chars().collect::<Vec<_>>(),
        '\n' => "\\n".chars().collect::<Vec<_>>(),
        '\r' => "\\r".chars().collect::<Vec<_>>(),
        '\t' => "\\t".chars().collect::<Vec<_>>(),
        _ => vec![c],
    }).collect()
```

```rust
}

pub fn run_index(root: &std::path::Path) -> io::Result<u64> {
    info!(root = %root.display(), "index start");
    let mut count = 0u64;

    for e in std::fs::read_dir(root)? {
        let e = e?;
        let p = e.path();
        let ft = e.file_type()?;
        if ft.is_file() {
            count += 1;
        }
    }

    info!(count, "index done");
    Ok(count)
}
```

27.3.11 Release engineering: versioning and artifacts

Versioning:

- Use semantic versioning (`MAJOR.MINOR.PATCH`) for the user-facing CLI.
- Treat changes to flags/output as API changes.
- Emit version via `–version` automatically from the CLI parser.

Artifacts you should ship:

- binaries for major platforms,
- checksums,

- changelog/release notes,
- shell completions,
- man pages (optional but professional).

27.3.12 Generate completions and man pages

Many teams generate these during CI and attach them to releases.

```rust
use clap::{CommandFactory};
use clap_complete::{generate_to, shells};
use std::fs;
use std::path::Path;

pub fn gen_all(out_dir: &Path) -> std::io::Result<()> {
    fs::create_dir_all(out_dir)?;

    let mut cmd = crate::cli::Cli::command();

    let _bash = generate_to(shells::Bash, &mut cmd, "procli", out_dir)?;
    let _zsh  = generate_to(shells::Zsh,  &mut cmd, "procli", out_dir)?;
    let _fish = generate_to(shells::Fish, &mut cmd, "procli", out_dir)?;
    let _ps   = generate_to(shells::PowerShell, &mut cmd, "procli", out_dir)?;

    Ok(())
}
```

27.3.13 Automating releases

A practical approach is to use a release automation tool that generates CI workflows to build and publish artifacts when you push a version tag. One commonly used workflow is:

- commit version bump,

- create a git tag like `v0.1.0`,
- push the tag,
- CI builds per-platform artifacts and publishes a GitHub Release.

Example release steps (human procedure):

```
# 1) update version in Cargo.toml
# 2) update CHANGELOG.md
git commit -am ”release: 0.1.0”
git tag v0.1.0
git push
git push --tags
```

27.3.14 Minimum quality gates before cutting a release

A release pipeline should at least run:

- `cargo fmt –check`
- `cargo clippy – -D warnings`
- `cargo test`
- build in `–release`

Optionally:

- `cargo audit` (dependency advisories),
- `cargo deny` (license and bans),
- reproducibility checks and checksums.

27.3.15 What “production-ready” means for this lab

Your CLI is production-ready when:

- `-help` is clear and examples exist,
- exit codes are stable and documented,
- logs are structured, filterable, and do not leak secrets,
- config precedence is deterministic and inspectable,
- releases are automated and repeatable from tags,
- outputs intended for tools are stable (JSON schema, TSV fields).

Part VIII

Testing, Maintenance, and Production Quality

Testing in Rust

Testing is not an optional activity in production Rust. It is a design tool that shapes APIs, forces explicit invariants, prevents regressions, and makes refactoring safe. Rust provides a strong built-in testing foundation via `cargo test` and the standard test harness, while the ecosystem adds high-leverage tools for fast, parallel execution and generative testing.

28.1 Unit Tests

28.1.1 What Unit Tests Validate

Unit tests validate *small, local* behavior:

- correctness of pure functions and small modules,
- invariants of internal data structures,
- boundary conditions (empty input, maximum sizes, overflow behavior if relevant),
- error paths and recovery behavior,
- performance-sensitive micro-behavior (as *behavioral* checks, not timing checks).

28.1.2 Where Unit Tests Live

Unit tests usually live in the same module/file as the code, inside a `cfg(test)` module. They can access private items in the same module, which is ideal for testing internal invariants without exposing internals publicly.

```
/* src/lib.rs */
pub fn clamp_i32(x: i32, lo: i32, hi: i32) -> i32 {
    debug_assert!(lo <= hi);
    if x < lo { lo } else if x > hi { hi } else { x }
}

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

    #[test]
    fn clamp_in_range_returns_same_value() {
        assert_eq!(clamp_i32(5, 0, 10), 5);
    }

    #[test]
    fn clamp_below_returns_lower_bound() {
        assert_eq!(clamp_i32(-5, 0, 10), 0);
    }

    #[test]
    fn clamp_above_returns_upper_bound() {
        assert_eq!(clamp_i32(50, 0, 10), 10);
    }
}
```

28.1.3 Assertion Style and Failure Diagnostics

Prefer assertions that communicate intent:

- `assert!` for boolean conditions,
- `assert_eq!` / `assert_ne!` for value expectations,
- custom messages for invariants that need context.

```rust
#[test]
fn parse_rejects_invalid_prefix() {
    let input = "XYZ:123";
    let err = parse_record(input).unwrap_err();
    assert!(err.is_prefix_error(), "unexpected error: {:?}", err);
}
```

28.1.4 Testing Errors Precisely

In production-quality systems, error checks must be specific and stable:

- validate the error *kind*, not the exact string,
- check important fields (e.g., index, reason code),
- avoid brittle matching on formatting.

```rust
#[derive(Debug, PartialEq)]
pub enum ParseError {
    Empty,
    BadPrefix,
    BadNumber,
}
```

```
pub fn parse_record(s: &str) -> Result<i32, ParseError> {
    if s.is_empty() { return Err(ParseError::Empty); }
    let (pfx, rest) = s.split_once(':').ok_or(ParseError::BadPrefix)?;
    if pfx != "OK" { return Err(ParseError::BadPrefix); }
    let n: i32 = rest.parse().map_err(|_| ParseError::BadNumber)?;
    Ok(n)
}

#[test]
fn parse_error_kinds_are_stable() {
    assert_eq!(parse_record(""), Err(ParseError::Empty));
    assert_eq!(parse_record("NOPE:1"), Err(ParseError::BadPrefix));
    assert_eq!(parse_record("OK:x"), Err(ParseError::BadNumber));
}
```

28.1.5 Tests Returning `Result`

For tests that do setup which can fail, returning `Result` keeps code clean:

```
use std::fs;

#[test]
fn loads_config_from_disk() -> Result<(), Box<dyn std::error::Error>> {
    fs::write("tmp.cfg", "mode=fast\n")?;
    let cfg = load_config("tmp.cfg")?;
    assert_eq!(cfg.mode, "fast");
    fs::remove_file("tmp.cfg")?;
    Ok(())
}
```

28.1.6 Panic Tests: Use Sparingly

If a panic is part of the contract (rare in production APIs), test it explicitly. Prefer error returns for production boundaries; panics are typically reserved for programmer errors.

```
#[test]
#[should_panic]
fn clamp_panics_in_debug_if_bounds_invalid() {
    /* This relies on debug_assert; may be optimized out in release. */
    let _ = clamp_i32(1, 10, 0);
}
```

28.1.7 Structuring Unit Tests for Large Modules

For non-trivial modules, group tests by behavior:

- tests::construction,
- tests::invariants,
- tests::serialization,
- tests::edge_cases.

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

        #[test]
        fn size_matches_internal_count() {
            let mut m = MetricSet::new();
            m.insert("a", 1);
            m.insert("b", 2);
            assert_eq!(m.len(), 2);
        }
    }

    mod edge_cases {
        use super::super::*;

        #[test]
        fn empty_is_valid_state() {
            let m = MetricSet::new();
            assert!(m.is_empty());
        }
    }
}
```

28.2 Integration Tests

28.2.1 What Integration Tests Validate

Integration tests validate *system behavior* through the public interface:

- public API correctness as consumed by external code,
- component wiring (parsing → validation → storage),
- correct error propagation across layers,
- behavior with realistic I/O boundaries (files, sockets) using test isolation.

28.2.2 Where Integration Tests Live

Integration tests live under `tests/` and compile as separate crates. They access only the public API, which prevents testing private internals and forces a consumer mindset.

```
/* tests/api_smoke.rs */
use mycrate::{parse_record, clamp_i32};

#[test]
fn api_smoke() {
    assert_eq!(clamp_i32(99, 0, 10), 10);
    assert_eq!(parse_record(”OK:42”).unwrap(), 42);
}
```

28.2.3 Sharing Helpers Across Integration Tests

Because each file under `tests/` is its own crate, share helpers via a module file.

```
/* tests/common/mod.rs */
use std::path::{Path, PathBuf};

pub fn test_data_path(name: &str) -> PathBuf {
    Path::new(”tests”).join(”data”).join(name)
}
```

```
/* tests/loads_data.rs */
mod common;

#[test]
fn loads_known_fixture() {
    let p = common::test_data_path(”sample.txt”);
    let s = std::fs::read_to_string(p).unwrap();
    assert!(s.contains(”hello”));
}
```

28.2.4 Testing With Realistic I/O Without Flakes

For production-quality tests:

- isolate filesystem tests using per-test temp directories,
- avoid tests that depend on network availability,
- avoid relying on wall-clock timing,
- design components so that I/O is injected (traits / interfaces) and deterministic in tests.

```rust
/* Example of dependency injection for deterministic integration tests */
pub trait Clock {
    fn now_ms(&self) -> u64;
}

pub struct SystemClock;

impl Clock for SystemClock {
    fn now_ms(&self) -> u64 {
        /* Replace with a real time source in production */
        0
    }
}

pub struct TokenIssuer<C: Clock> {
    clock: C,
}

impl<C: Clock> TokenIssuer<C> {
    pub fn new(clock: C) -> Self { Self { clock } }
    pub fn issue(&self) -> String {
        format!("t-{}", self.clock.now_ms())
    }
}

struct FakeClock(u64);
```

```
impl Clock for FakeClock {
    fn now_ms(&self) -> u64 { self.0 }
}

#[test]
fn token_is_deterministic_under_fake_clock() {
    let issuer = TokenIssuer::new(FakeClock(1234));
    assert_eq!(issuer.issue(), ”t-1234”);
}
```

28.2.5 End-to-End (E2E) Tests in Rust Projects

E2E tests often execute the produced binary and assert outputs. This is extremely effective for CLI tools and services.

```
/* tests/cli_e2e.rs */
use std::process::Command;

#[test]
fn cli_prints_help() {
    let out = Command::new(env!(”CARGO_BIN_EXE_mytool”))
        .arg(”--help”)
        .output()
        .expect(”run binary”);

    assert!(out.status.success());
    let s = String::from_utf8_lossy(&out.stdout);
    assert!(s.contains(”USAGE”) || s.contains(”Usage”));
}
```

28.3 Property-Based Testing

28.3.1 Why Property-Based Testing

Unit tests check a handful of examples. Property-based testing (PBT) checks *families of inputs* and validates invariants:

- serialization round-trips,
- ordering and set/map invariants,
- parser behavior under random valid/invalid inputs,
- arithmetic properties (monotonicity, idempotence, commutativity when applicable),
- state machine properties (sequences of operations).

When a counterexample is found, a good PBT framework *shrinks* it to a minimal failing case.

28.3.2 Choosing a PBT Tooling Style

Two common styles:

- **Strategy-based** (high control, strong shrinking): typically `proptest`.
- **Trait-based** (quick to start): a QuickCheck-family approach.

In high-assurance systems, strategy-based testing is often preferred because you can precisely define valid domains and shrinking behavior.

28.3.3 Core Pattern: Invariants Over Generated Inputs

Example property: *clamp always returns within bounds* and is *idempotent*.

```
/* Cargo.toml (dev-dependencies)
proptest = "..."
*/

#[cfg(test)]
mod pbt {
    use super::*;
    use proptest::prelude::*;

    proptest! {
        #[test]
        fn clamp_is_within_bounds(x in any::<i32>(), a in any::<i32>(), b in any::<i32>()) {
            /* Ensure lo <= hi by construction */
            let (lo, hi) = if a <= b { (a, b) } else { (b, a) };

            let y = clamp_i32(x, lo, hi);
            prop_assert!(y >= lo);
            prop_assert!(y <= hi);
        }

        #[test]
        fn clamp_is_idempotent(x in any::<i32>(), a in any::<i32>(), b in any::<i32>()) {
            let (lo, hi) = if a <= b { (a, b) } else { (b, a) };

            let y1 = clamp_i32(x, lo, hi);
            let y2 = clamp_i32(y1, lo, hi);
            prop_assert_eq!(y1, y2);
        }
    }
}
```

28.3.4 Round-Trip Properties

A classic production property is *encode then decode returns the original*. This catches subtle bugs that example-based tests miss.

```
#[derive(Debug, Clone, PartialEq, Eq)]
pub struct KeyVal {
    pub key: String,
    pub val: Vec<u8>,
}

/* A simple, explicit format: len(key) ':' key len(val) ':' val-hex */
pub fn encode(kv: &KeyVal) -> String {
    let hex = kv.val.iter().map(|b| format!("{:02x}", b)).collect::<String>();
    format!("{}:{}:{}:{}", kv.key.len(), kv.key, kv.val.len(), hex)
}

pub fn decode(s: &str) -> Result<KeyVal, ()> {
    let mut it = s.splitn(4, ':');
    let klen: usize = it.next().ok_or(())?.parse().map_err(|_| ())?;
    let key = it.next().ok_or(())?.to_string();
    let vlen: usize = it.next().ok_or(())?.parse().map_err(|_| ())?;
    let hex = it.next().ok_or(())?;

    if key.len() != klen { return Err(()); }
    if hex.len() != vlen * 2 { return Err(()); }

    let mut val = Vec::with_capacity(vlen);
    for i in (0..hex.len()).step_by(2) {
        let b = u8::from_str_radix(&hex[i..i+2], 16).map_err(|_| ())?;
        val.push(b);
    }
    Ok(KeyVal { key, val })
}

#[cfg(test)]
mod roundtrip {
    use super::*;
    use proptest::prelude::*;

    fn key_strategy() -> impl Strategy<Value = String> {
        /* Restrict to visible ASCII for simplicity; expand as needed */
        prop::collection::vec(prop::char::range('a','z'), 0..32)
            .prop_map(|v| v.into_iter().collect())
    }

    fn val_strategy() -> impl Strategy<Value = Vec<u8>> {
        prop::collection::vec(any::<u8>(), 0..64)
    }

    proptest! {
        #[test]
        fn encode_decode_roundtrip(key in key_strategy(), val in val_strategy()) {
            let kv = KeyVal { key, val };
            let s = encode(&kv);
            let out = decode(&s).map_err(|_| "decode failed")?;
            prop_assert_eq!(out, kv);
        }
    }
}
```

28.3.5 Stateful Properties: Sequences of Operations

For systems code, the most valuable properties often involve *sequences*: insert/remove/update/load/save and invariants that must always hold.

Below is a compact model-based check: compare your structure to a simple reference model.

```
use std::collections::BTreeMap;

#[derive(Clone, Debug)]
pub enum Op {
    Put(String, i32),
    Del(String),
}

pub struct Store {
    inner: BTreeMap<String, i32>,
}

impl Store {
    pub fn new() -> Self { Self { inner: BTreeMap::new() } }
    pub fn put(&mut self, k: String, v: i32) { self.inner.insert(k, v); }
    pub fn del(&mut self, k: &str) { self.inner.remove(k); }
    pub fn get(&self, k: &str) -> Option<i32> { self.inner.get(k).copied() }
}

#[cfg(test)]
mod stateful {
    use super::*;
    use proptest::prelude::*;

    fn small_key() -> impl Strategy<Value = String> {
        prop::collection::vec(prop::char::range('a','f'), 0..8)
            .prop_map(|v| v.into_iter().collect())
    }

    fn op_strategy() -> impl Strategy<Value = Op> {
        prop_oneof![
            (small_key(), any::<i32>()).prop_map(|(k,v)| Op::Put(k,v)),
            small_key().prop_map(Op::Del),
        ]
    }

    proptest! {
        #[test]
        fn store_matches_reference_model(ops in prop::collection::vec(op_strategy(),
        ↪   0..200)) {
            let mut s = Store::new();
            let mut refm: BTreeMap<String, i32> = BTreeMap::new();

            for op in ops {
                match op {
                    Op::Put(k,v) => { s.put(k.clone(), v); refm.insert(k, v); }
                    Op::Del(k) => { s.del(&k); refm.remove(&k); }
                }

                /* Invariant: all keys in reference must match store */
                for (k, v) in refm.iter() {
                    prop_assert_eq!(s.get(k), Some(*v));
                }
            }
        }
    }
}
```

28.3.6 Production Rules for Property-Based Testing

- Constrain domains to *valid inputs* when validating functional invariants; generate invalid inputs separately when validating error handling.

- Keep properties stable and meaningful; avoid properties that simply re-check internal implementation details.

- Use shrinking-friendly strategies; if a failure is found, you want a minimal reproducer.
- Ensure determinism: isolate randomness within the test framework; avoid time-based and external dependencies.
- Combine with classic unit tests: PBT finds corner cases; unit tests document exact known behaviors.

Lab: Full Project Coverage

This lab builds a small but realistic Rust library + CLI, then drives coverage using unit, integration, and property-based tests. The objective is not only “high coverage” but *high confidence*: invariants, boundaries, and end-to-end behavior.

28.3.7 Project Spec

Build a crate named `telemetry_kv`:

- A library that stores key/value metrics with basic parsing and validation.
- A CLI that can:
  - `put KEY VALUE`
  - `get KEY`
  - `del KEY`
  - `import FILE` (reads `KEY=VALUE` lines)
- Requirements:
  - keys: `[a-z0-9_]` length 1..32,

- values: signed 32-bit integers,
- import ignores blank lines and comments starting with #,
- invalid lines produce structured errors (line number + kind).

28.3.8 Library Skeleton

```rust
/* src/lib.rs */
use std::collections::BTreeMap;

#[derive(Debug, Clone, PartialEq, Eq)]
pub enum ImportErrorKind {
    EmptyKey,
    BadKeyChar,
    KeyTooLong,
    MissingEquals,
    BadValue,
}

#[derive(Debug, Clone, PartialEq, Eq)]
pub struct ImportError {
    pub line: usize,
    pub kind: ImportErrorKind,
}

pub struct TelemetryKv {
    map: BTreeMap<String, i32>,
}

impl TelemetryKv {
    pub fn new() -> Self { Self { map: BTreeMap::new() } }

    pub fn put(&mut self, key: String, value: i32) {
        self.map.insert(key, value);
    }

    pub fn get(&self, key: &str) -> Option<i32> {
        self.map.get(key).copied()
    }

    pub fn del(&mut self, key: &str) -> bool {
        self.map.remove(key).is_some()
    }

    pub fn import_lines<I: IntoIterator<Item = String>>(&mut self, lines: I) -> Result<(),
    ↪   ImportError> {
        for (idx, raw) in lines.into_iter().enumerate() {
            let line_no = idx + 1;
            let s = raw.trim();

            if s.is_empty() || s.starts_with('#') {
                continue;
            }

            let (k, v) = s.split_once('=').ok_or(ImportError { line: line_no, kind:
            ↪   ImportErrorKind::MissingEquals })?;
            let key = k.trim();
            validate_key(key).map_err(|kind| ImportError { line: line_no, kind })?;

            let val_str = v.trim();
            let value: i32 = val_str.parse().map_err(|_| ImportError { line: line_no, kind:
            ↪   ImportErrorKind::BadValue })?;

            self.put(key.to_string(), value);
        }
        Ok(())
    }
}

fn validate_key(key: &str) -> Result<(), ImportErrorKind> {
    if key.is_empty() { return Err(ImportErrorKind::EmptyKey); }
    if key.len() > 32 { return Err(ImportErrorKind::KeyTooLong); }
    for ch in key.chars() {
        let ok = ch.is_ascii_lowercase() || ch.is_ascii_digit() || ch == '_';
        if !ok { return Err(ImportErrorKind::BadKeyChar); }
    }
    Ok(())
}
```

28.3.9 Unit Tests: Internal Invariants and Error Kinds

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

    #[test]
    fn validate_key_rules() {
        assert_eq!(validate_key(""), Err(ImportErrorKind::EmptyKey));
        assert_eq!(validate_key("a_b9"), Ok(()));
        assert_eq!(validate_key("A"), Err(ImportErrorKind::BadKeyChar));
        assert_eq!(validate_key("a-b"), Err(ImportErrorKind::BadKeyChar));
        assert_eq!(validate_key(&"a".repeat(33)), Err(ImportErrorKind::KeyTooLong));
    }

    #[test]
    fn put_get_del_flow() {
        let mut db = TelemetryKv::new();
        db.put("cpu0".to_string(), 10);
        assert_eq!(db.get("cpu0"), Some(10));
        assert_eq!(db.del("cpu0"), true);
        assert_eq!(db.get("cpu0"), None);
        assert_eq!(db.del("cpu0"), false);
    }

    #[test]
    fn import_reports_line_and_kind() {
        let mut db = TelemetryKv::new();
        let lines = vec![
            "cpu0=10".to_string(),
            "bad-key=1".to_string(), /* '-' invalid */
        ];

        let err = db.import_lines(lines).unwrap_err();
        assert_eq!(err.line, 2);
        assert_eq!(err.kind, ImportErrorKind::BadKeyChar);
    }
}
```

28.3.10 Integration Tests: Public API + Realistic Fixtures

Create `tests/import_integration.rs` to test the crate as a consumer would.

```
/* tests/import_integration.rs */
use telemetry_kv::{TelemetryKv, ImportErrorKind};

#[test]
fn import_skips_comments_and_blanks() {
    let mut db = TelemetryKv::new();
    let lines = vec![
        "".to_string(),
        "   ".to_string(),
        "# comment".to_string(),
        ”cpu0=10”.to_string(),
        ”mem=20”.to_string(),
    ];
    db.import_lines(lines).unwrap();
    assert_eq!(db.get(”cpu0”), Some(10));
    assert_eq!(db.get(”mem”), Some(20));
}

#[test]
fn import_missing_equals_is_detected() {
    let mut db = TelemetryKv::new();
    let lines = vec![”cpu0:10”.to_string()];
    let err = db.import_lines(lines).unwrap_err();
    assert_eq!(err.line, 1);
    assert_eq!(err.kind, ImportErrorKind::MissingEquals);
}
```

28.3.11 Property-Based Tests: Invariants Over Many Inputs

Now add a property: after importing a set of valid KEY=VALUE lines, each key maps to the last value assigned to that key. This checks ordering, overwrites, and parsing robustness.

```
/* tests/property_import.rs */
use telemetry_kv::TelemetryKv;
use proptest::prelude::*;
use std::collections::BTreeMap;

fn key_strategy() -> impl Strategy<Value = String> {
    /* keys: [a-z0-9_] length 1..32 */
    let ch = prop_oneof![
        prop::char::range('a','z'),
        prop::char::range('0','9'),
        Just('_'),
    ];
    prop::collection::vec(ch, 1..33).prop_map(|v| v.into_iter().collect())
}

proptest! {
    #[test]
    fn import_last_write_wins(pairs in prop::collection::vec((key_strategy(), any::<i32>()),
    ↪   0..200)) {
        let mut db = TelemetryKv::new();

        let mut model: BTreeMap<String, i32> = BTreeMap::new();
        let mut lines: Vec<String> = Vec::new();

        for (k, v) in pairs {
            lines.push(format!("{}={}", k, v));
            model.insert(k, v);
        }

        db.import_lines(lines).unwrap();

        for (k, v) in model.iter() {
            prop_assert_eq!(db.get(k), Some(*v));
        }
    }
}
```

28.3.12 CLI Layer and E2E Tests

Add a minimal CLI that calls the library. Then write E2E tests that execute the binary. (Use a command-based test to validate output, exit code, and argument parsing.)

```
/* src/main.rs */
use telemetry_kv::TelemetryKv;
```

```
fn main() {
    /* Minimal example for lab; production CLI should have robust parsing and help output. */
    let mut db = TelemetryKv::new();
    let args: Vec<String> = std::env::args().collect();

    if args.len() < 2 {
        eprintln!("Usage: telemetry_kv <cmd> ...");
        std::process::exit(2);
    }

    match args[1].as_str() {
        "put" if args.len() == 4 => {
            let key = args[2].clone();
            let value: i32 = args[3].parse().unwrap_or_else(|_| {
                eprintln!("Bad value");
                std::process::exit(2);
            });
            db.put(key, value);
            println!("OK");
        }
        "get" if args.len() == 3 => {
            match db.get(&args[2]) {
                Some(v) => println!("{}", v),
                None => { eprintln!("Not found"); std::process::exit(1); }
            }
        }
        _ => {
            eprintln!("Unknown or invalid command");
            std::process::exit(2);
        }
    }
}
```

```
/* tests/cli_e2e.rs */
use std::process::Command;

#[test]
fn cli_usage_on_missing_args() {
    let out = Command::new(env!("CARGO_BIN_EXE_telemetry_kv"))
        .output()
        .expect("run binary");

    assert!(!out.status.success());
    let err = String::from_utf8_lossy(&out.stderr);
    assert!(err.contains("Usage"));
}
```

28.3.13 Coverage Discipline: What "Full Coverage" Means in Practice

A production-ready "full coverage" target is not just a percentage; it is a checklist:

- **API coverage:** every public function has direct or indirect tests.
- **Error coverage:** every error kind is triggered by at least one test.
- **Boundary coverage:** minimum/maximum sizes, empty cases, and parsing edges.
- **Invariant coverage:** key data-structure invariants validated repeatedly (unit + PBT).
- **E2E coverage:** at least one test executes the final binary with realistic scenarios.

28.3.14 Operational Testing Workflow

A pragmatic workflow used in real Rust teams:

- `cargo test` for local correctness.

- fast parallel test execution (next-generation runners) for CI speed and flaky detection.
- coverage runs in CI to prevent untested code from silently growing.
- separate “slow” test tier (I/O-heavy, many-case PBT) from quick PR checks.

28.3.15 Maintenance Rules: Keeping Tests Healthy Over Time

- Treat tests as product code: refactor them, keep them readable, remove duplication.
- Prefer deterministic tests; isolate randomness behind PBT frameworks.
- Avoid time-based assertions; test logical outcomes and invariants.
- Keep integration tests stable by designing APIs that allow dependency injection.
- When a bug is found in production: add a regression test *first*, then fix.

By combining unit tests (internal invariants), integration tests (public behavior), property-based tests (wide input space), and E2E tests (real binary execution), you get a testing strategy that supports aggressive optimization and refactoring without sacrificing correctness or reliability.

Documentation as Code

28.4 rustdoc

`rustdoc` is Rust's official documentation generator and one of the most important production-quality tools in the ecosystem. It is not a "pretty printer" for comments. It understands the Rust language model: modules, visibility, traits, generics, lifetimes, associated items, re-exports, and feature-gated APIs. The result is documentation that matches the compiled public surface, not a separate narrative that can silently drift.

28.4.1 What rustdoc Generates and Why It Matters

When you run `cargo doc` (which invokes `rustdoc`), you get:

- a crate landing page (crate-level docs),
- module pages and an API tree reflecting visibility and re-exports,
- per-item pages (types, traits, functions, constants, macros),
- rendered signatures with full generics and bounds,
- trait implementations and implementor lists,
- intra-doc links that are resolved by item identity,

- embedded, type-checked code blocks (and optionally executed as doctests).

In systems work, `rustdoc` is part of correctness:

- it clarifies invariants and preconditions (especially around `unsafe`),
- it documents performance characteristics and allocation behavior,
- it prevents "tribal knowledge" by making design intent explicit,
- it supports long-term maintenance by keeping the contract close to the code.

28.4.2 Doc Comment Forms and Placement

Rust documentation is written in doc comments:

- `//!` attaches to the enclosing module or crate (top-of-file narrative),
- `///` attaches to the item that follows (API reference contract),
- `#[doc = "..."]` is the attribute form (useful for generated docs or macros).

```
/* src/lib.rs */

/*!
Crate documentation: purpose, guarantees, non-goals, and quick start.

This crate provides a deterministic telemetry store:
- predictable iteration order,
- explicit key validation,
- stable error semantics.
*/

pub mod kv {
    //! Module documentation: scope, invariants, and intended usage.
    /// Stores integer metrics keyed by validated identifiers.
    pub struct TelemetryKv {
        /* ... */
    }
}
```

28.4.3 A Professional rustdoc Structure for Public Items

For production APIs, treat each doc comment as a contract. A robust structure is:

- **Summary line**: one sentence stating what the item does.
- **Behavior**: key semantics, edge cases, ordering, determinism.
- **Guarantees / Invariants**: facts callers can rely on.
- **Complexity / Allocation**: what matters for performance and predictability.
- **Errors**: stable, enumerated error meaning (avoid string-matching contracts).
- **Panics**: only if it can panic and under which conditions.
- **Safety**: for `unsafe`, a complete safety contract.
- **Examples**: minimal, correct, compile-checked code.

```
#[derive(Debug, Clone, PartialEq, Eq)]
pub enum ParseLineError {
    MissingEquals,
    BadKey,
    BadValue,
}
```

```
/// Parses a telemetry line of the form `KEY=VALUE`.
///
/// # Behavior
/// - Leading/trailing whitespace is ignored.
/// - Keys must match `[a-z0-9_]` and length `1..=32`.
/// - Values must parse as signed 32-bit integers.
///
/// # Guarantees
/// On success, the returned key is normalized exactly as input after trimming.
///
/// # Errors
/// - `MissingEquals` if `=` is absent.
/// - `BadKey` if the key violates validation rules.
/// - `BadValue` if the value cannot be parsed as `i32`.
///
/// # Examples
/// ```
/// # use telemetry_kv::{parse_line, ParseLineError};
/// assert_eq!(parse_line(”cpu0=10”).unwrap(), (”cpu0”.to_string(), 10));
/// assert_eq!(parse_line(”NO_EQUALS”).unwrap_err(), ParseLineError::MissingEquals);
/// assert_eq!(parse_line(”bad-key=1”).unwrap_err(), ParseLineError::BadKey);
/// assert_eq!(parse_line(”cpu0=NaN”).unwrap_err(), ParseLineError::BadValue);
/// ```
pub fn parse_line(s: &str) -> Result<(String, i32), ParseLineError> {
    let s = s.trim();
    let (k, v) = s.split_once('=').ok_or(ParseLineError::MissingEquals)?;
    let key = k.trim();
    let val = v.trim();

    if key.is_empty() || key.len() > 32 {
        return Err(ParseLineError::BadKey);
    }
    if !key.chars().all(|c| c.is_ascii_lowercase() || c.is_ascii_digit() || c == '_') {
```

```
        return Err(ParseLineError::BadKey);
    }

    let value: i32 = val.parse().map_err(|_| ParseLineError::BadValue)?;
    Ok((key.to_string(), value))
}
```

28.4.4 Intra-Doc Links for Refactor-Resistant Documentation

Rust docs can link to items by name. This is crucial for maintainability because it avoids duplicated explanations and stays stable across refactors.

```
/// Inserts or replaces a value.
///
/// See also: [`TelemetryKv::get`], [`TelemetryKv::del`].
pub fn put(&mut self, key: String, value: i32) -> Option<i32> {
    /* ... */
    None
}
```

In professional codebases, intra-doc links are used as a documentation graph:

- error types link to “how to handle errors” guidance,
- `unsafe` functions link to the safety model section,
- high-level APIs link to lower-level building blocks and invariants.

28.4.5 Documenting Performance, Allocation, and Determinism

For high-performance systems, “what it does” is incomplete without “how it behaves under load.” At minimum, public APIs should document:

- complexity (Big-O),

- allocation behavior (allocates? clones? reuses buffers?),
- determinism (ordering, stable iteration),
- concurrency expectations (thread-safety, internal locking, lock-free invariants).

```
use std::collections::BTreeMap;

/// A deterministic store with predictable iteration order.
pub struct TelemetryKv {
    map: BTreeMap<String, i32>,
}

impl TelemetryKv {
    /// Creates an empty store.
    ///
    /// # Allocation
    /// Does not allocate.
    pub fn new() -> Self {
        Self { map: BTreeMap::new() }
    }

    /// Returns the current value for `key`.
    ///
    /// # Complexity
    /// `O(log n)` comparisons.
    ///
    /// # Allocation
    /// Does not allocate.
    ///
    /// # Determinism
    /// Deterministic across runs for the same inserted keys.
    pub fn get(&self, key: &str) -> Option<i32> {
        self.map.get(key).copied()
    }

    /// Inserts or replaces the value associated with `key`.
    ///
    /// # Complexity
    /// `O(log n)` comparisons.
    ///
    /// # Allocation
    /// May allocate if `key` is a new string not previously stored.
    pub fn put(&mut self, key: String, value: i32) -> Option<i32> {
        self.map.insert(key, value)
    }

    /// Deletes `key` and returns whether it existed.
    ///
    /// # Complexity
    /// `O(log n)` comparisons.
    ///
    /// # Allocation
    /// Does not allocate.
    pub fn del(&mut self, key: &str) -> bool {
        self.map.remove(key).is_some()
    }
}
```

28.4.6 Documenting `unsafe`: The Safety Contract Is the API

For `unsafe` functions, the documentation must precisely state the obligations of the caller. If the safety contract is incomplete, the API is incomplete.

```
/// Views a raw memory region as a byte slice without copying.
///
/// # Safety
/// The caller must ensure all of the following:
/// - `ptr` is valid for reads of `len` bytes.
/// - The memory range `[ptr, ptr.add(len))` is not mutated for the lifetime `'a`.
/// - The returned slice does not outlive the allocation backing `ptr`.
/// - The pointer is non-null if `len != 0`.
///
/// Breaking these rules can cause Undefined Behavior.
pub unsafe fn view_bytes<'a>(ptr: *const u8, len: usize) -> &'a [u8] {
    if len == 0 {
        return &[];
    }
    debug_assert!(!ptr.is_null());
    std::slice::from_raw_parts(ptr, len)
}
```

A production-grade pattern is to include:

- explicit aliasing constraints (who may write?),
- lifetime constraints (how long must memory stay valid?),
- alignment constraints (especially for typed views),
- concurrency constraints (data races are UB even if “it works on my machine”).

28.4.7 Controlling Visibility and Public Surface in rustdoc

Documentation quality depends on controlling what is visible:

- hide internal modules and helpers from public docs,
- keep internal invariants internal while documenting public guarantees,
- re-export a curated API surface for end users.

```
#[doc(hidden)]
pub mod internal {
    pub fn internal_helper() {}
}

/* Curated public surface */
pub use kv::TelemetryKv;
```

28.4.8 Conditional Documentation for Feature-Gated APIs

Many systems crates expose features (fast hashing, OS-specific backends, SIMD paths). Documentation should make feature constraints explicit.

```
#[cfg(feature = "fast-hash")]
/// Fast hash backend enabled by the `fast-hash` feature.
///
/// # Notes
/// This backend may trade determinism across versions for speed.
/// Use only if your system does not require cross-version stable hashing.
pub struct FastHasher;
```

28.4.9 Doc Attributes for Precision and Maintainability

The attribute form is valuable in large or generated code:

- `#[doc = "..."]` for programmatic or macro-driven docs,
- `#[doc(hidden)]` to remove internal items from public docs,
- `#[deprecated]` to communicate migration paths to users.

```
#[deprecated(since = "2.0.0", note =
↪   "Use `TelemetryKv::put` which returns the previous value.")]
pub fn insert_legacy(/* ... */) {
    /* ... */
}

#[doc = "Parses input using the stable telemetry format."]
pub fn parse_stable(/* ... */) {
    /* ... */
}
```

28.4.10 Workflow: Building, Reviewing, and Treating Docs as Release Criteria

A professional workflow treats documentation as a release gate:

- **Build docs in CI**: generate docs for every change to detect breakage.
- **Enforce "no missing docs" on public items**: treat undocumented public APIs as defects.
- **Doctest critical examples**: examples must compile and remain correct.
- **Document changes with migrations**: deprecations include a path forward.

A practical enforcement approach is to deny missing documentation on public items during development for library crates:

```
/* src/lib.rs */
#![deny(missing_docs)]
```

This does not replace good judgment; it forces you to explicitly document the public contract. The goal is not verbosity. The goal is correctness: stable guarantees, explicit safety boundaries, and examples that remain accurate as the code evolves.

28.5 Doctests

Doctests are Rust code examples embedded directly inside documentation comments that are compiled (and often executed) by the standard Rust test harness. In production-quality Rust, doctests are not decorative snippets. They are executable specifications that prevent documentation drift, enforce correct usage patterns, and act as high-signal regression tests for the most important user-facing behaviors.

28.5.1 What a Doctest Is in Rust

A doctest is a fenced Rust code block inside a doc comment, for example inside `///` or `//!` documentation:

- When documentation is built and tested, each code block is extracted.
- The block is compiled in a controlled context.
- Depending on annotations, it may also be executed.
- Failures are reported as test failures, making broken documentation a build-quality problem.

````
/// Returns `true` if `x` is even.
///
/// ```
/// # use mycrate::is_even;
/// assert!(is_even(4));
/// assert!(!is_even(5));
/// ```
pub fn is_even(x: i32) -> bool {
    x % 2 == 0
}
````

28.5.2 Why Doctests Matter for Production Quality

Doctests provide benefits that classic unit tests often miss:

- They validate the *user story*: the public API as consumers will use it.
- They guarantee examples compile as the API evolves.
- They anchor correct usage patterns (ownership, borrowing, lifetimes, error handling).
- They reduce onboarding time by showing minimal working code.
- They enforce that the documentation and code are never out of sync.

28.5.3 Doctest Execution Model

Rust doctests are compiled as separate test units. Conceptually, the documentation snippet is wrapped into a synthetic context and compiled. Important implications:

- A snippet must import what it uses (unless you hide setup lines).
- Names must resolve as they would for an external consumer.
- The snippet should not rely on private internals.
- Side effects (filesystem, network, time) can cause flakes; doctests must be deterministic.

28.5.4 Minimal, Correct, and Deterministic Examples

Production doctests should be:

- minimal: show the intended usage with as few lines as possible,
- correct: represent the recommended approach, not “some approach,”

- deterministic: avoid time, network, random behavior, and global state.

```
/// Parses a line `KEY=VALUE` into `(String, i32)`.
///
/// ```
/// # use telemetry_kv::{parse_line, ParseLineError};
/// assert_eq!(parse_line("cpu0=10").unwrap(), ("cpu0".to_string(), 10));
/// assert_eq!(parse_line("NO_EQUALS").unwrap_err(), ParseLineError::MissingEquals);
/// ```
pub fn parse_line(s: &str) -> Result<(String, i32), ParseLineError> {
    /* ... */
    unimplemented!()
}
```

28.5.5 Hidden Lines: Compiled but Not Rendered

Doctests support “hidden” lines that begin with #. These lines are compiled, but not displayed in the rendered docs. This lets you keep examples clean while still providing imports and setup.

```
/// Demonstrates storing and retrieving a metric.
///
/// ```
/// # use telemetry_kv::TelemetryKv;
/// let mut db = TelemetryKv::new();
/// db.put("cpu0".to_string(), 10);
/// assert_eq!(db.get("cpu0"), Some(10));
/// ```
pub fn docs_only_example() {}
```

Best practice: hide only boilerplate. Never hide the key line that teaches usage.

28.5.6 Doctests for Error Contracts

A production API’s error behavior is part of its contract. Doctests should:

- demonstrate the main error types,
- show recommended handling patterns,
- test stable error kinds rather than brittle string messages.

```
#[derive(Debug, Clone, PartialEq, Eq)]
pub enum ParseLineError {
    MissingEquals,
    BadKey,
    BadValue,
}

/// Parses a line and returns structured errors.
///
/// ```
/// # use telemetry_kv::{parse_line, ParseLineError};
/// assert_eq!(parse_line("NO_EQUALS").unwrap_err(), ParseLineError::MissingEquals);
/// assert_eq!(parse_line("bad-key=1").unwrap_err(), ParseLineError::BadKey);
/// assert_eq!(parse_line("cpu0=NaN").unwrap_err(), ParseLineError::BadValue);
/// ```
pub fn parse_line(s: &str) -> Result<(String, i32), ParseLineError> {
    /* ... */
    unimplemented!()
}
```

28.5.7 Doctests That Should Compile but Not Run

Some examples are correct but should not be executed in CI (e.g., network I/O, infinite loops, environment-dependent code). For these, use:

- no_run: compile but do not execute,
- ignore: do not compile or run as part of doctests by default.

```
/// Example that is valid but should not run during tests.
///
/// ```no_run
/// # use std::net::TcpStream;
/// let _sock = TcpStream::connect("127.0.0.1:9000")?;
/// # Ok::<(), std::io::Error>(())
/// ```
pub fn network_doc_example() {}
```

Use `no_run` when the code is platform-stable and should still compile. Use `ignore` when compilation itself may fail across targets or feature sets.

28.5.8 Doctests with `Result`: Idiomatic Pattern

Many examples involve fallible operations. The cleanest doctest pattern is to end the snippet with a `Result` return so `?` can be used.

```
/// Loads configuration from a string.
///
/// ```
/// # use mycrate::Config;
/// # fn demo() -> Result<(), Box<dyn std::error::Error>> {
/// let cfg = Config::parse("mode=fast\nthreads=4\n")?;
/// assert_eq!(cfg.mode(), "fast");
/// assert_eq!(cfg.threads(), 4);
/// # Ok(())
/// # }
/// # demo().unwrap();
/// ```
pub fn config_docs() {}
```

This pattern avoids `unwrap()` on intermediate steps while keeping the example minimal and correct.

28.5.9 Doctests for Concurrency and Non-Determinism

Concurrency examples often become flaky when they depend on timing. A production doctest should:

- avoid sleeping and deadlines,
- avoid asserting exact interleavings,
- focus on invariants that must hold regardless of scheduling.

````
/// Demonstrates thread-safe shared counter.
///
/// ```
/// # use std::sync::{Arc, Mutex};
/// # use std::thread;
/// let x = Arc::new(Mutex::new(0u32));
/// let mut th = Vec::new();
///
/// for _ in 0..4 {
///     let x2 = Arc::clone(&x);
///     th.push(thread::spawn(move || {
///         let mut g = x2.lock().unwrap();
///         *g += 1;
///     }));
/// }
///
/// for t in th { t.join().unwrap(); }
/// assert_eq!(*x.lock().unwrap(), 4);
/// ```
pub fn concurrency_doc_example() {}
````

This is deterministic because it asserts a final invariant after all joins, not a schedule-dependent ordering.

28.5.10 Doctests for `unsafe`: Demonstrate the Boundary Clearly

If the API involves `unsafe`, doctests must demonstrate:

- what is safe to do,
- what the caller must guarantee,
- how to confine unsafe usage to a small boundary.

```
/// Views a byte slice without copying.
///
/// ```
/// # use mycrate::view_bytes;
/// let data = [1u8, 2, 3, 4];
/// let p = data.as_ptr();
/// let s = unsafe { view_bytes(p, data.len()) };
/// assert_eq!(s, &data);
/// ```
pub fn unsafe_doc_example() {}
```

28.5.11 Doctests and Feature-Gated APIs

When features change the public surface, doctests must remain valid across builds. Use one of these approaches:

- place feature-specific doctests in feature-gated modules,
- guard the doctest code with conditional compilation inside the snippet.

```
#[cfg(feature = ”fast-hash”)]
/// Uses the fast hashing backend.
///
/// ```
```

```
/// # #[cfg(feature = "fast-hash”)]
/// # {
/// # use mycrate::FastHasher;
/// let h = FastHasher::new();
/// let _ = h.hash(b”abc”);
/// # }
/// ```
pub struct FastHasher;
```

28.5.12 Common Doctest Failure Modes and Professional Fixes

Production teams most commonly break doctests due to:

- missing imports after refactoring module paths,
- renamed items that the docs still mention,
- examples relying on private internals,
- non-deterministic behavior (time, randomness, network),
- platform-specific behavior without guards.

Professional fixes:

- hide setup imports with #,
- keep examples focused on the public contract,
- prefer value-based assertions over output printing,
- isolate platform-specific code behind guards,
- treat doctest failures as API quality regressions.

28.5.13 Doctests as a Release Gate

For production libraries, doctests should run in CI:

- they ensure docs compile for every release,
- they validate that the “quick start” story still works,
- they reduce the risk of shipping misleading documentation.

A stable practice is to ensure key public items have at least one doctest demonstrating the recommended usage and at least one doctest covering the primary error behavior. The goal is not maximum coverage; the goal is maximum correctness of the public contract.

28.6 README-driven development

README-driven development (RDD) is the practice of designing a Rust crate by writing the README first and treating it as the authoritative user-facing specification. In production-quality Rust—especially systems and performance-critical code—the README is not marketing text; it is the adoption contract that defines how the crate is meant to be used, what it guarantees, and what it explicitly does not guarantee.

RDD enforces clarity early. If an API cannot be explained clearly in a short README with a minimal example, the API is likely too complex or poorly structured.

28.6.1 Core Principles

RDD is based on the following principles:

- The README defines the minimal public API surface.
- Examples represent supported and recommended usage.

- Guarantees written in the README are part of the compatibility contract.
- Changes to README semantics imply versioning consequences.

28.6.2 Why RDD Is Effective in Rust

Rust APIs are defined as much by constraints as by functionality. RDD forces early, explicit decisions about:

- ownership and borrowing expectations,
- structured error models and stable error kinds,
- determinism and ordering guarantees,
- performance characteristics and allocation behavior,
- safety boundaries and feature-flag trade-offs.

These properties are difficult to retrofit later without breaking users, making RDD particularly effective for long-lived Rust libraries.

28.6.3 Minimal Professional README Structure

A concise, production-grade README should include:

- a one-paragraph purpose statement,
- a minimal Quick Start example that compiles,
- explicit guarantees (correctness, determinism, complexity),
- a brief description of the error model,
- feature flags and their trade-offs (if any),
- clear non-goals to prevent misuse.

28.6.4 Quick Start Discipline

Quick Start examples must be:

- minimal and deterministic,
- free of hidden assumptions,
- representative of recommended usage,
- honest about error handling.

The Quick Start is the most copied code in the project; misleading examples inevitably lead to production bugs.

28.6.5 RDD as a Release Rule

In README-driven development:

- if the README example breaks, the release is broken,
- adding or weakening guarantees is a compatibility decision,
- undocumented behavior is not part of the public contract.

RDD turns documentation into a design tool. By defining behavior before implementation, it produces simpler APIs, fewer breaking changes, and safer adoption in production Rust systems.

Lab: Professional-Grade API Documentation

This lab upgrades a small Rust crate from “some docs exist” to professional-grade, production-quality API documentation. The goal is not verbosity; it is correctness and durability: documentation that remains accurate under refactoring, communicates safety and performance contracts precisely, and provides executable examples that enforce the user story.

28.6.6 Lab Outcomes

By the end of this lab, your crate will have:

- crate-level documentation that clearly states scope, intended audience, guarantees, and non-goals,
- consistent item-level docs for every public type/function/trait,
- doctests that compile (and run when appropriate) to prevent documentation drift,
- explicit documentation of performance characteristics (complexity, allocation, determinism),
- explicit safety contracts for any `unsafe` surfaces,
- a documentation quality gate suitable for CI and release readiness.

28.6.7 Project Setup

Create a small library crate named `telemetry_kv`. It stores validated key/value integer metrics with deterministic iteration.

```
cargo new telemetry_kv --lib
```

Create a minimal module layout:

- `src/lib.rs`: crate docs and re-exports,
- `src/kv.rs`: main type and API,
- `src/error.rs`: stable error kinds and error type.

28.6.8 Step 1: Crate-Level Documentation as the Contract

Write crate-level docs in `src/lib.rs` using `//!`. The crate docs must include:

- a one-paragraph purpose statement,
- a short Quick Start (kept minimal),
- guarantees and invariants,
- non-goals,
- high-level notes about performance and determinism.

```
/* src/lib.rs */

#![deny(missing_docs)]

/*!
A deterministic telemetry key/value store for systems programming.

This crate is designed for predictable behavior and stable contracts:
- keys are validated (`[a-z0-9_]`, length `1..=32`),
- iteration order is deterministic,
- lookups are `O(log n)` and do not allocate.

Non-goals:
- persistence,
- distributed storage,
- schema management.
*/

pub mod error;
pub mod kv;
```

```rust
pub use crate::error::{Error, ErrorKind};
pub use crate::kv::TelemetryKv;
```

Professional rule: every guarantee written here is part of your compatibility contract.

28.6.9 Step 2: Define a Stable Error Model

Professional APIs expose stable, matchable error kinds. Avoid string-only errors for public contracts.

```rust
/* src/error.rs */

/// Stable error categories for the public API.
#[derive(Debug, Clone, PartialEq, Eq)]
pub enum ErrorKind {
    /// Key violates validation rules.
    BadKey,
    /// Operation required an existing key but it was absent.
    NotFound,
}

/// Public error type with a stable error kind.
#[derive(Debug, Clone, PartialEq, Eq)]
pub struct Error {
    kind: ErrorKind,
}

impl Error {
    /// Returns the stable error kind.
    pub fn kind(&self) -> ErrorKind { self.kind.clone() }

    fn bad_key() -> Self { Self { kind: ErrorKind::BadKey } }
    fn not_found() -> Self { Self { kind: ErrorKind::NotFound } }

    pub(crate) fn bad_key_public() -> Self { Self::bad_key() }
    pub(crate) fn not_found_public() -> Self { Self::not_found() }
}
```

28.6.10 Step 3: Implement the Core Type with Documented Guarantees

The main type must document:

- invariants (key rules, determinism),
- complexity and allocation behavior,
- errors and when they occur,
- examples as doctests (minimal and stable).

```
/* src/kv.rs */

use std::collections::BTreeMap;
use crate::error::{Error, ErrorKind};

/// A deterministic telemetry key/value store.
///
/// # Guarantees
/// - Keys are validated: `[a-z0-9_]`, length `1..=32`.
/// - Iteration order is deterministic (sorted keys).
///
/// # Performance
/// - Lookups are `O(log n)` and do not allocate.
/// - Inserts are `O(log n)` and allocate only when a new key is stored.
pub struct TelemetryKv {
    map: BTreeMap<String, i32>,
}

impl TelemetryKv {
    /// Creates an empty store.
    ///
    /// # Allocation
    /// Does not allocate.
    ///
    /// # Examples
    /// ```
    /// # use telemetry_kv::TelemetryKv;
    /// let db = TelemetryKv::new();
    /// assert_eq!(db.len(), 0);
    /// ```
    pub fn new() -> Self {
        Self { map: BTreeMap::new() }
    }

    /// Returns the number of stored keys.
    ///
    /// # Complexity
    /// `O(1)`.
    pub fn len(&self) -> usize {
        self.map.len()
    }

    /// Inserts or replaces the value associated with `key`.
    ///
    /// # Errors
    /// Returns `ErrorKind::BadKey` if the key violates validation rules.
    ///
    /// # Examples
    /// ```
```

```
/// # use telemetry_kv::{TelemetryKv, ErrorKind};
/// let mut db = TelemetryKv::new();
/// db.put("cpu0", 10).unwrap();
/// assert_eq!(db.get("cpu0").unwrap(), Some(10));
/// assert_eq!(db.put("bad-key", 1).unwrap_err().kind(), ErrorKind::BadKey);
/// ```
pub fn put(&mut self, key: &str, value: i32) -> Result<(), Error> {
    validate_key(key)?;
    self.map.insert(key.to_string(), value);
    Ok(())
}

/// Returns the current value for `key`.
///
/// # Errors
/// Returns `ErrorKind::BadKey` if the key violates validation rules.
///
/// # Complexity
/// `O(log n)` comparisons.
///
/// # Allocation
/// Does not allocate.
///
/// # Examples
/// ```
/// # use telemetry_kv::TelemetryKv;
/// let mut db = TelemetryKv::new();
/// db.put("cpu0", 10).unwrap();
/// assert_eq!(db.get("cpu0").unwrap(), Some(10));
/// assert_eq!(db.get("missing").unwrap(), None);
/// ```
pub fn get(&self, key: &str) -> Result<Option<i32>, Error> {
    validate_key(key)?;
```

```
        Ok(self.map.get(key).copied())
    }

    /// Returns the value for `key` or an error if absent.
    ///
    /// # Errors
    /// - `ErrorKind::BadKey` if the key is invalid.
    /// - `ErrorKind::NotFound` if the key does not exist.
    ///
    /// # Examples
    /// ```
    /// # use telemetry_kv::{TelemetryKv, ErrorKind};
    /// let mut db = TelemetryKv::new();
    /// db.put("cpu0", 10).unwrap();
    /// assert_eq!(db.must_get("cpu0").unwrap(), 10);
    /// assert_eq!(db.must_get("missing").unwrap_err().kind(), ErrorKind::NotFound);
    /// ```
    pub fn must_get(&self, key: &str) -> Result<i32, Error> {
        match self.get(key)? {
            Some(v) => Ok(v),
            None => Err(Error::not_found_public()),
        }
    }

    /// Returns an iterator over `(key, value)` in deterministic order.
    ///
    /// # Determinism
    /// Keys are returned in sorted order.
    ///
    /// # Examples
    /// ```
    /// # use telemetry_kv::TelemetryKv;
    /// let mut db = TelemetryKv::new();
```

```
    /// db.put("b", 2).unwrap();
    /// db.put("a", 1).unwrap();
    /// let v: Vec<(&str, i32)> = db.iter().map(|(k, v)| (k.as_str(), *v)).collect();
    /// assert_eq!(v, vec![("a", 1), ("b", 2)]);
    /// ```
    pub fn iter(&self) -> impl Iterator<Item = (&String, &i32)> {
        self.map.iter()
    }
}

fn validate_key(key: &str) -> Result<(), Error> {
    if key.is_empty() || key.len() > 32 {
        return Err(Error::bad_key_public());
    }
    if !key.chars().all(|c| c.is_ascii_lowercase() || c.is_ascii_digit() || c == '_') {
        return Err(Error::bad_key_public());
    }
    Ok(())
}
```

28.6.11 Step 4: Doctest Quality Rules (Your “Documentation Test Policy”)

Establish a strict policy for doctests:

- Every public item should have at least one doctest that compiles and demonstrates the recommended usage.
- The doctest must be minimal; hide boilerplate with # lines, but never hide the key usage line.
- Avoid time, randomness, and network dependencies.
- Use structured error kind assertions (`ErrorKind`) instead of string comparisons.

- If an example should compile but not run, mark it `no_run`.

28.6.12 Step 5: Documentation of Performance, Allocation, and Determinism

Add explicit documentation notes to public APIs:

- Complexity for each operation (Big-O).
- Allocation behavior (allocating vs non-allocating calls).
- Determinism properties (ordering, stable behavior across runs).
- Any concurrency assumptions (if applicable).

Professional note: do not promise absolute timings. Promise properties you can maintain (allocation-free hot paths, deterministic order, stable error kinds).

28.6.13 Step 6: Build Docs and Run Doctests

Run:

```
cargo test
cargo doc --no-deps
```

Your acceptance criteria:

- `cargo test` passes (including doctests).
- Public items are documented (missing docs fail due to `deny(missing_docs)`).
- Docs describe behavior, errors, determinism, and performance in a way that is consistent with code.

28.6.14 Step 7: Professional Review Checklist (Release Gate)

Before release, verify:

- Crate docs contain purpose, guarantees, non-goals, and a minimal Quick Start.
- Every public item documents: behavior, errors, complexity, allocation, determinism (as relevant).
- Doctests exist for the main success path and primary error path.
- Error model is stable and matchable (no string-only contracts).
- No docs depend on non-deterministic or environment-specific behavior.
- No `unsafe` API exists without a complete safety contract.

A crate that satisfies this lab is “documentation-complete” in a production sense: users can adopt it safely, maintainers can refactor with confidence, and the documentation cannot silently drift away from reality because it is continuously compiled and tested.

Versioning and Package Architecture

28.7 Semantic versioning

Semantic Versioning (SemVer) is the discipline of communicating compatibility through version numbers. In Rust ecosystems, SemVer is not only a social convention; it is deeply operational because dependency resolution, transitive upgrades, and API stability assumptions depend on it.

28.7.1 Version meaning: `MAJOR.MINOR.PATCH`

SemVer assigns meaning to each component:

- **PATCH** (`X.Y.Z` → `X.Y.(Z+1)`): bug fixes only; no intended breaking changes.
- **MINOR** (`X.Y.Z` → `X.(Y+1).0`): backward-compatible additions (new functions, new types, new trait impls when safe).
- **MAJOR** (`X.Y.Z` → `(X+1).0.0`): breaking changes (API removals, signature changes, behavior that breaks consumers).

28.7.2 Rust-specific interpretation: what counts as "breaking"

In Rust libraries, "breaking" includes more than obvious signature changes:

- Removing or renaming public items (`pub` types, functions, modules, constants).

- Changing function signatures, trait bounds, lifetimes, or visibility.
- Tightening input requirements without preserving old behavior (e.g., rejecting values previously accepted).
- Changing error types or error variants in a way that breaks pattern matching.
- Changing semantics of existing functions in ways that violate documented guarantees.
- Changing trait implementations when downstream code relied on those impls (e.g., removing `Send`/`Sync` or removing an impl that enabled blanket behavior).
- Making previously public fields private, or changing struct layout if users relied on construction patterns.

A practical rule: if a reasonable downstream crate could stop compiling or could compile but break a documented contract, treat it as a breaking change.

28.7.3 The special case of `0.y.z` versions

SemVer allows rapid evolution before `1.0.0`. Many Rust crates follow these conventions:

- `0.y.z`: the API is not guaranteed stable in the strict SemVer sense.
- In practice, many projects treat `0.y` as a "major line," so `0.(y+1).0` may include breaking changes.
- For professional users, **document your policy**: whether `0.y` is treated like major, and how breaking changes are communicated.

28.7.4 Dependency constraints in `Cargo.toml`

Cargo uses version requirements to select compatible versions. Common patterns:

```
[dependencies]
/* Caret requirement (default style): compatible updates */
serde = "1.0"

/* More conservative: patch-only updates */
regex = "=1.10.2"       # exact version
log   = "0.4.21"        # caret still; patch updates are allowed within 0.4
anyhow = "~1.0"         # tilde: patch updates within 1.0.* (minor stays fixed)

/* Range: explicit bounds */
tokio = ">=1.36, <2.0"
```

Professional guidance:

- Use caret requirements for mature dependencies that follow SemVer well.
- Use tighter bounds for dependencies that are known to make breaking changes under minor releases.
- If you publish a library, avoid overly tight pins that force downstream dependency conflicts.

28.7.5 Release discipline: what to document for each version

A production-quality release process documents:

- **Compatibility statement**: what is compatible and what is not.
- **Migration notes**: code changes needed to upgrade, especially for breaking releases.
- **Behavioral changes**: even if signatures are unchanged, note changes affecting correctness/performance.

- **MSRV policy**: the Minimum Supported Rust Version and when it changes.

28.7.6 MSRV as part of compatibility

For systems users, the Rust compiler version is operational. Changing MSRV can break builds even if the API is unchanged. A professional approach:

- Declare MSRV in documentation and/or `Cargo.toml` metadata.
- Only raise MSRV deliberately and announce it clearly.
- Treat MSRV increases as compatibility-relevant (often aligned with a MINOR or MAJOR bump depending on your policy and audience).

28.7.7 Lockfiles and libraries

- `Cargo.lock` should generally be committed for binaries and applications to ensure reproducible builds.
- Libraries typically do not commit `Cargo.lock` (because downstream users resolve dependencies), but internal policies may differ.

28.8 Feature flags

Feature flags are Cargo's mechanism for conditional compilation and optional capabilities. They are essential for production architecture: they allow you to keep a small default surface, support multiple backends, avoid heavy dependencies by default, and control portability.

28.8.1 What a feature flag does

A feature flag can:

- enable optional dependencies,
- activate extra modules or APIs,
- switch between backends (e.g., OS-specific or performance-specific),
- control compilation of expensive functionality (serde, crypto, async runtime support).

28.8.2 Defining features in `Cargo.toml`

```
[features]
/* Keep defaults minimal for systems users; enable only what is truly baseline */
default = ["std"]

/* Typical split: std vs core/no_std support */
std = []
no_std = []

/* Optional capability flags */
serde = ["dep:serde"]
fast-hash = ["dep:ahash"]
```

Optional dependencies are declared like this:

```
[dependencies]
serde = { version = "1.0", optional = true, default-features = false }
ahash = { version = "0.8", optional = true }
```

28.8.3 Using features in code

```
#[cfg(feature = "serde")]
mod ser_support;

#[cfg(feature = "fast-hash")]
```

```
pub type HashMap<K, V> = std::collections::HashMap<K, V, ahash::RandomState>;

#[cfg(not(feature = "fast-hash"))]
pub type HashMap<K, V> = std::collections::HashMap<K, V>;
```

28.8.4 Professional rules for feature design

- **Prefer additive features**: enabling a feature should add capability, not break existing behavior.
- **Avoid “mutually exclusive” feature traps**: Cargo features are additive and unify across the dependency graph.
- **Document defaults**: what is enabled by default, what is optional, and why.
- **Keep the public API stable**: avoid changing types/signatures based on features unless clearly separated into feature-gated modules.
- **Name features for capability, not implementation**: prefer `serde`, `tls`, `simd` over `use-xyz-crate`.

28.8.5 Feature unification and its consequences

Because features unify across the entire build:

- If any dependency enables a feature of a crate, that feature is enabled for all uses of that crate.
- This can change binary size, enable optional backends unintentionally, or activate heavier code paths.

Mitigations:

- Make default features minimal.
- Prefer “capability features” that are safe to enable globally.
- Keep feature-gated APIs in clearly separated modules so the base API does not change shape.

28.8.6 Feature flags and SemVer

Feature changes can be compatibility changes:

- Removing a feature is breaking.
- Changing what a feature means (semantics) can be breaking even if it compiles.
- Turning on a feature by default can be breaking (build times, dependencies, `no_std` users, binary size budgets).

28.9 Workspaces

A Cargo workspace is a set of crates built and managed together. Workspaces enable monorepo-style development, shared dependency management, consistent tooling, and atomic refactoring across multiple packages.

28.9.1 Core workspace concepts

- A workspace has a root `Cargo.toml` that declares members.
- Each member crate is a normal crate with its own `Cargo.toml`.
- Workspace builds share a single `Cargo.lock` at the root for consistent resolution.
- You can centralize dependency versions and profiles for consistency.

28.9.2 Minimal workspace root

```
[workspace]
members = [
  "crates/telemetry_kv",
  "crates/telemetry_cli",
  "crates/telemetry_proto",
]

/* Modern dependency/feature resolution behavior (recommended) */
resolver = "2"
```

28.9.3 Centralizing dependency versions with `workspace.dependencies`

A professional monorepo avoids version drift:

```
[workspace.dependencies]
anyhow = "1.0"
clap = "4.5"
serde = { version = "1.0", default-features = false, features = ["derive"] }
```

Then in member crates:

```
[dependencies]
anyhow = { workspace = true }
serde = { workspace = true }
```

28.9.4 Profiles and build policies at the workspace level

Workspaces can set consistent build profiles:

```
[profile.release]
lto = true
```

```
codegen-units = 1
panic = "abort"

[profile.dev]
debug = 2
```

Professional considerations:

- Use `panic = "abort"` for many embedded and high-reliability binaries.
- Use LTO and fewer codegen units for maximum runtime performance (trade-off: compile time).
- Keep dev builds debuggable and fast.

28.9.5 Workspace hygiene rules

- Keep crates small and single-purpose (library core, CLI, integration adapters).
- Avoid cyclic dependencies; define clean layering (core library does not depend on CLI).
- Share types through a dedicated crate if necessary (`proto/types` crates).
- Enforce consistent linting and formatting policies across the workspace.

Lab: Small Monorepo Setup

This lab builds a small, professional monorepo with:

- **telemetry_kv**: core library (deterministic KV store),
- **telemetry_proto**: shared types and parsing contracts,
- **telemetry_cli**: a binary that uses the library.

28.9.6 Step 1: Create the workspace layout

Target layout:

```
monorepo/
  Cargo.toml
  Cargo.lock
  crates/
    telemetry_kv/
      Cargo.toml
      src/lib.rs
      src/kv.rs
      src/error.rs
    telemetry_proto/
      Cargo.toml
      src/lib.rs
    telemetry_cli/
      Cargo.toml
      src/main.rs
```

28.9.7 Step 2: Workspace root `Cargo.toml`

```
[workspace]
members = [
  "crates/telemetry_kv",
  "crates/telemetry_proto",
  "crates/telemetry_cli",
]
resolver = "2"

[workspace.package]
edition = "2021"
license = "MIT"
```

```toml
[workspace.dependencies]
anyhow = "1.0"
clap = "4.5"
serde = { version = "1.0", default-features = false, features = ["derive"] }

[profile.release]
lto = true
codegen-units = 1
panic = "abort"
```

28.9.8 Step 3: Shared types crate (`telemetry_proto`)

```toml
# crates/telemetry_proto/Cargo.toml
[package]
name = "telemetry_proto"
version = "0.1.0"
edition = { workspace = true }
license = { workspace = true }

[dependencies]
serde = { workspace = true, optional = true }

[features]
default = []
serde = ["dep:serde"]
```

```rust
/* crates/telemetry_proto/src/lib.rs */

/// A validated telemetry key (invariant: `[a-z0-9_]`, length `1..=32`).
#[derive(Debug, Clone, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct Key(String);
```

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

    pub fn parse(s: &str) -> Result<Self, ()> {
        let t = s.trim();
        if t.is_empty() || t.len() > 32 { return Err(()); }
        if !t.chars().all(|c| c.is_ascii_lowercase() || c.is_ascii_digit() || c == '_') {
            return Err(());
        }
        Ok(Key(t.to_string()))
    }
}

/// A simple record: key and signed integer value.
#[derive(Debug, Clone, PartialEq, Eq)]
pub struct Record {
    pub key: Key,
    pub value: i32,
}
```

28.9.9 Step 4: Core library crate (telemetry_kv) with SemVer discipline

```toml
# crates/telemetry_kv/Cargo.toml
[package]
name = "telemetry_kv"
version = "0.1.0"
edition = { workspace = true }
license = { workspace = true }

[dependencies]
telemetry_proto = { path = "../telemetry_proto" }

[features]
```

```
default = []
serde = [”telemetry_proto/serde”]
```

```
/* crates/telemetry_kv/src/lib.rs */

#![deny(missing_docs)]

/*!
Core deterministic KV store.

SemVer discipline:
- Public types and error kinds are stable within the policy of this crate version line.
- Feature flags are additive; default is minimal.
*/

pub mod error;
pub mod kv;

pub use crate::error::{Error, ErrorKind};
pub use crate::kv::TelemetryKv;
```

```
/* crates/telemetry_kv/src/error.rs */

/// Stable error categories for matching.
#[derive(Debug, Clone, PartialEq, Eq)]
pub enum ErrorKind {
    BadKey,
    NotFound,
}

/// Public error type.
#[derive(Debug, Clone, PartialEq, Eq)]
pub struct Error {
    kind: ErrorKind,
}

impl Error {
    pub fn kind(&self) -> ErrorKind { self.kind.clone() }
    pub(crate) fn bad_key() -> Self { Self { kind: ErrorKind::BadKey } }
    pub(crate) fn not_found() -> Self { Self { kind: ErrorKind::NotFound } }
}
```

```
/* crates/telemetry_kv/src/kv.rs */

use std::collections::BTreeMap;
use telemetry_proto::Key;
use crate::error::Error;

/// Deterministic key/value store.
pub struct TelemetryKv {
    map: BTreeMap<Key, i32>,
}

impl TelemetryKv {
    pub fn new() -> Self { Self { map: BTreeMap::new() } }

    pub fn put(&mut self, key: &str, value: i32) -> Result<(), Error> {
        let k = Key::parse(key).map_err(|_| Error::bad_key())?;
        self.map.insert(k, value);
        Ok(())
    }

    pub fn get(&self, key: &str) -> Result<Option<i32>, Error> {
        let k = Key::parse(key).map_err(|_| Error::bad_key())?;
        Ok(self.map.get(&k).copied())
    }

    pub fn must_get(&self, key: &str) -> Result<i32, Error> {
        match self.get(key)? {
            Some(v) => Ok(v),
            None => Err(Error::not_found()),
        }
    }
}
```

28.9.10 Step 5: CLI crate (`telemetry_cli`)

```
# crates/telemetry_cli/Cargo.toml
[package]
name = "telemetry_cli"
version = "0.1.0"
edition = { workspace = true }
license = { workspace = true }

[dependencies]
anyhow = { workspace = true }
clap = { workspace = true, features = ["derive"] }
telemetry_kv = { path = "../telemetry_kv" }
```

```
/* crates/telemetry_cli/src/main.rs */

use anyhow::Result;
use clap::{Parser, Subcommand};
use telemetry_kv::TelemetryKv;

#[derive(Parser)]
struct Args {
    #[command(subcommand)]
    cmd: Cmd,
}
```

```
#[derive(Subcommand)]
enum Cmd {
    Put { key: String, value: i32 },
    Get { key: String },
}

fn main() -> Result<()> {
    let args = Args::parse();
    let mut db = TelemetryKv::new();

    match args.cmd {
        Cmd::Put { key, value } => {
            db.put(&key, value)?;
            println!("OK");
        }
        Cmd::Get { key } => {
            let v = db.get(&key)?;
            match v {
                Some(x) => println!("{}", x),
                None => println!("(none)"),
            }
        }
    }
    Ok(())
}
```

28.9.11 Step 6: Versioning and feature policy checklist

Before publishing or tagging releases:

- Decide SemVer policy for `0.y.z` (what counts as breaking, and how it is communicated).
- Keep features additive; never change defaults casually.

- Keep core library minimal; place heavy/optional integration behind features or separate crates.
- Centralize dependency versions in `[workspace.dependencies]` to avoid drift.
- Use workspace profiles for consistent production builds across binaries.

This monorepo architecture scales: it supports safe refactoring, consistent dependency policy, controlled feature growth, and SemVer discipline aligned with professional production requirements.

Part IX

Capstone Projects

Project 1: Parser and Interpreter

This capstone project builds a small, production-quality parser and interpreter in Rust with a focus on correctness, clear error reporting, and maintainable architecture. The goal is not to produce the largest language, but to build a system you can trust: deterministic behavior, explicit invariants, strong test coverage, and performance-aware design.

28.10 Lexer, parser, AST

28.10.1 Language Scope and Design Targets

We implement a small expression language suitable for demonstrating the full pipeline:

- literals: integers, booleans, strings (minimal escape support),
- identifiers and `let` bindings,
- arithmetic: `+ - * /` with precedence,
- comparisons: `== != < <= > >=`,
- logical operators: `&& || !`,
- grouping with parentheses,

- statements: expression statements and `let` statements,
- end-of-input and semicolon-terminated statements (optional; we support both).

Engineering targets:

- predictable, non-panicking control flow (errors are values),
- source spans for every token and AST node,
- errors include location and a human-meaningful message,
- parser design supports incremental extension without rewriting core logic.

28.10.2 Core Data Types: Span, Token, and TokenKind

A robust lexer produces tokens annotated with spans. Spans are the backbone of error reporting and diagnostics.

```rust
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub struct Span {
    pub start: usize, /* byte offset inclusive */
    pub end: usize,   /* byte offset exclusive */
}

impl Span {
    pub fn len(self) -> usize { self.end.saturating_sub(self.start) }
    pub fn merge(self, other: Span) -> Span {
        Span { start: self.start.min(other.start), end: self.end.max(other.end) }
    }
}

#[derive(Debug, Clone, PartialEq, Eq)]
pub struct Token {
    pub kind: TokenKind,
    pub span: Span,
}

#[derive(Debug, Clone, PartialEq, Eq)]
pub enum TokenKind {
    /* single-char */
    LParen, RParen, Plus, Minus, Star, Slash, Semicolon,
    /* one-or-two char */
    Bang, BangEq, Eq, EqEq,
    Lt, LtEq, Gt, GtEq,
    AndAnd, OrOr,

    /* keywords */
    Let, True, False,

    /* literals and identifiers */
    Ident(String),
    Int(i64),
    Str(String),

    /* end marker */
    Eof,
}
```

28.10.3 Lexer: Deterministic Tokenization with Spans

A production lexer must:

- never panic on malformed input,
- always make forward progress (or report an error),
- be explicit about what is allowed (ident rules, number format),

- record spans precisely.

```
#[derive(Debug, Clone, PartialEq, Eq)]
pub enum LexErrorKind {
    UnexpectedChar(char),
    UnterminatedString,
    InvalidEscape,
    IntOverflow,
}

#[derive(Debug, Clone, PartialEq, Eq)]
pub struct LexError {
    pub kind: LexErrorKind,
    pub span: Span,
}

pub struct Lexer<'a> {
    src: &'a str,
    bytes: &'a [u8],
    i: usize,
}

impl<'a> Lexer<'a> {
    pub fn new(src: &'a str) -> Self {
        Self { src, bytes: src.as_bytes(), i: 0 }
    }

    fn at_end(&self) -> bool { self.i >= self.bytes.len() }

    fn peek(&self) -> Option<u8> {
        self.bytes.get(self.i).copied()
    }

    fn bump(&mut self) -> Option<u8> {
        let b = self.peek()?;
        self.i += 1;
        Some(b)
    }

    fn span_from(&self, start: usize) -> Span {
        Span { start, end: self.i }
    }

    fn skip_ws(&mut self) {
        while let Some(b) = self.peek() {
            match b {
                b' ' | b'\t' | b'\n' | b'\r' => { self.i += 1; }
                _ => break,
            }
        }
    }

    fn match_next(&mut self, expected: u8) -> bool {
        if self.peek() == Some(expected) {
            self.i += 1;
            true
        } else {
            false
        }
    }

    fn lex_ident_or_keyword(&mut self, start: usize) -> TokenKind {
        while let Some(b) = self.peek() {
            let c = b as char;
            if c.is_ascii_alphanumeric() || c == '_' {
                self.i += 1;
            } else {
                break;
            }
        }
        let s = &self.src[start..self.i];
        match s {
            "let" => TokenKind::Let,
            "true" => TokenKind::True,
            "false" => TokenKind::False,
            _ => TokenKind::Ident(s.to_string()),
        }
    }

    fn lex_int(&mut self, start: usize) -> Result<TokenKind, LexErrorKind> {
        while let Some(b) = self.peek() {
            let c = b as char;
            if c.is_ascii_digit() { self.i += 1; } else { break; }
        }
        let s = &self.src[start..self.i];
        match s.parse::<i64>() {
            Ok(v) => Ok(TokenKind::Int(v)),
            Err(_) => Err(LexErrorKind::IntOverflow),
        }
    }

    fn lex_string(&mut self, start_quote: usize) -> Result<TokenKind, LexErrorKind> {
        /* we are positioned right after the opening quote */
        let mut out = String::new();
        while let Some(b) = self.bump() {
            match b {
                b'"' => return Ok(TokenKind::Str(out)),
                b'\\' => {
                    let esc = self.bump().ok_or(LexErrorKind::UnterminatedString)?;
                    match esc {
```

```
                    b'n' => out.push('\n'),
                    b't' => out.push('\t'),
                    b'\\' => out.push('\\'),
                    b'"' => out.push('"'),
                    _ => return Err(LexErrorKind::InvalidEscape),
                }
            }
            _ => out.push(b as char),
        }
    }
    let _ = start_quote;
    Err(LexErrorKind::UnterminatedString)
}

pub fn next_token(&mut self) -> Result<Token, LexError> {
    self.skip_ws();
    let start = self.i;

    if self.at_end() {
        return Ok(Token { kind: TokenKind::Eof, span: Span { start, end: start } });
    }

    let b = self.bump().unwrap();
    let kind = match b {
        b'(' => TokenKind::LParen,
        b')' => TokenKind::RParen,
        b'+' => TokenKind::Plus,
        b'-' => TokenKind::Minus,
        b'*' => TokenKind::Star,
        b'/' => TokenKind::Slash,
        b';' => TokenKind::Semicolon,

        b'!' => if self.match_next(b'=') { TokenKind::BangEq } else { TokenKind::Bang },
        b'=' => if self.match_next(b'=') { TokenKind::EqEq } else { TokenKind::Eq },
        b'<' => if self.match_next(b'=') { TokenKind::LtEq } else { TokenKind::Lt },
        b'>' => if self.match_next(b'=') { TokenKind::GtEq } else { TokenKind::Gt },

        b'&' => {
            if self.match_next(b'&') { TokenKind::AndAnd }
            else {
                return Err(LexError { kind: LexErrorKind::UnexpectedChar('&'), span:
                ↪   self.span_from(start) });
            }
        }
        b'|' => {
            if self.match_next(b'|') { TokenKind::OrOr }
            else {
                return Err(LexError { kind: LexErrorKind::UnexpectedChar('|'), span:
                ↪   self.span_from(start) });
            }
        }

        b'"' => {
            match self.lex_string(start) {
                Ok(k) => k,
               Err(k) => return Err(LexError { kind: k, span: Span { start, end: self.i
                ↪   } }),
            }
        }

        _ => {
            let c = b as char;
            if c.is_ascii_alphabetic() || c == '_' {
                let k = self.lex_ident_or_keyword(start);
                k
            } else if c.is_ascii_digit() {
                match self.lex_int(start) {
                    Ok(k) => k,
                    Err(k) => return Err(LexError { kind: k, span: Span { start, end:
                    ↪  self.i } }),
                }
            } else {
                return Err(LexError { kind: LexErrorKind::UnexpectedChar(c), span:
                ↪  self.span_from(start) });
            }
        }
    };

    Ok(Token { kind, span: self.span_from(start) })
}

pub fn tokenize(mut self) -> Result<Vec<Token>, LexError> {
    let mut out = Vec::new();
    loop {
        let t = self.next_token()?;
        let is_eof = matches!(t.kind, TokenKind::Eof);
        out.push(t);
        if is_eof { break; }
    }
    Ok(out)
}
}
```

28.10.4 AST: Typed Nodes with Spans

The AST should:

- represent the language structure (not tokens),
- carry spans for diagnostics,

- be easy to extend (new operators, new statements, new expressions).

```rust
#[derive(Debug, Clone, PartialEq)]
pub enum Expr {
    Int(i64, Span),
    Bool(bool, Span),
    Str(String, Span),
    Var(String, Span),

    Unary { op: UnaryOp, rhs: Box<Expr>, span: Span },
    Binary { lhs: Box<Expr>, op: BinaryOp, rhs: Box<Expr>, span: Span },
}

impl Expr {
    pub fn span(&self) -> Span {
        match self {
            Expr::Int(_, s) => *s,
            Expr::Bool(_, s) => *s,
            Expr::Str(_, s) => *s,
            Expr::Var(_, s) => *s,
            Expr::Unary { span, .. } => *span,
            Expr::Binary { span, .. } => *span,
        }
    }
}

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum UnaryOp { Not, Neg }

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum BinaryOp {
    Add, Sub, Mul, Div,
    Eq, Ne, Lt, Le, Gt, Ge,
    And, Or,
}

#[derive(Debug, Clone, PartialEq)]
pub enum Stmt {
    Let { name: String, expr: Expr, span: Span },
    Expr { expr: Expr, span: Span },
}

#[derive(Debug, Clone, PartialEq)]
pub struct Program {
    pub stmts: Vec<Stmt>,
}
```

28.10.5 Parser: Precedence Climbing (Pratt-Style) with Clear Control Flow

We implement a precedence-aware expression parser. Goals:

- correct precedence and associativity,
- predictable error recovery points,
- spans merged from child nodes,
- minimal token lookahead.

```
#[derive(Debug, Clone, PartialEq, Eq)]
pub enum ParseErrorKind {
    UnexpectedEof,
    UnexpectedToken,
    ExpectedToken,
    ExpectedExpression,
}

#[derive(Debug, Clone, PartialEq, Eq)]
```

```
pub struct ParseError {
    pub kind: ParseErrorKind,
    pub span: Span,
    pub message: String,
}

pub struct Parser {
    tokens: Vec<Token>,
    i: usize,
}

impl Parser {
    pub fn new(tokens: Vec<Token>) -> Self { Self { tokens, i: 0 } }

    fn peek(&self) -> &Token {
        self.tokens.get(self.i).unwrap_or_else(|| self.tokens.last().unwrap())
    }

    fn bump(&mut self) -> &Token {
        let t = self.peek();
        self.i = (self.i + 1).min(self.tokens.len().saturating_sub(1));
        t
    }

    fn at_eof(&self) -> bool {
        matches!(self.peek().kind, TokenKind::Eof)
    }

    fn match_kind(&mut self, k: &TokenKind) -> Option<Token> {
        if std::mem::discriminant(&self.peek().kind) == std::mem::discriminant(k) {
            Some(self.bump().clone())
        } else {
            None
        }
    }

    fn expect<F>(&mut self, what: &str, f: F) -> Result<Token, ParseError>
    where
        F: Fn(&TokenKind) -> bool
    {
        let t = self.peek().clone();
        if f(&t.kind) {
            Ok(self.bump().clone())
        } else {
            Err(ParseError {
                kind: ParseErrorKind::ExpectedToken,
                span: t.span,
                message: format!(”expected {}, found {:?}”, what, t.kind),
            })
        }
    }

    pub fn parse_program(&mut self) -> Result<Program, ParseError> {
        let mut stmts = Vec::new();
        while !self.at_eof() {
            let s = self.parse_stmt()?;
            stmts.push(s);

            /* optional semicolon: allows both styles */
            if self.match_kind(&TokenKind::Semicolon).is_some() {
                /* consumed */
            }
        }
        Ok(Program { stmts })
    }
```

```
    fn parse_stmt(&mut self) -> Result<Stmt, ParseError> {
        if matches!(self.peek().kind, TokenKind::Let) {
            let let_tok = self.bump().clone();
            let name = match self.bump().kind.clone() {
                TokenKind::Ident(s) => s,
                other => {
                    return Err(ParseError {
                        kind: ParseErrorKind::UnexpectedToken,
                        span: self.peek().span,
                      message: format!("expected identifier after let, found {:?}", other),
                    });
                }
            };
            let eq = self.expect("'='", |k| matches!(k, TokenKind::Eq))?;
            let expr = self.parse_expr(0)?;
            let span = let_tok.span.merge(eq.span).merge(expr.span());
            Ok(Stmt::Let { name, expr, span })
        } else {
            let expr = self.parse_expr(0)?;
            let span = expr.span();
            Ok(Stmt::Expr { expr, span })
        }
    }

    fn parse_expr(&mut self, min_bp: u8) -> Result<Expr, ParseError> {
        let mut lhs = self.parse_prefix()?;

        loop {
            let (l_bp, r_bp, op) = match self.infix_binding_power() {
                Some(x) => x,
                None => break,
            };
            if l_bp < min_bp { break; }
            let op_tok = self.bump().clone();
            let rhs = self.parse_expr(r_bp)?;
            let span = lhs.span().merge(op_tok.span).merge(rhs.span());
            lhs = Expr::Binary { lhs: Box::new(lhs), op, rhs: Box::new(rhs), span };
        }

        Ok(lhs)
    }

    fn parse_prefix(&mut self) -> Result<Expr, ParseError> {
        let t = self.peek().clone();
        match t.kind.clone() {
            TokenKind::Int(v) => {
                let tok = self.bump().clone();
                Ok(Expr::Int(v, tok.span))
            }
            TokenKind::True => {
                let tok = self.bump().clone();
                Ok(Expr::Bool(true, tok.span))
            }
            TokenKind::False => {
                let tok = self.bump().clone();
                Ok(Expr::Bool(false, tok.span))
            }
            TokenKind::Str(s) => {
                let tok = self.bump().clone();
                Ok(Expr::Str(s, tok.span))
            }
            TokenKind::Ident(name) => {
                let tok = self.bump().clone();
                Ok(Expr::Var(name, tok.span))
            }
            TokenKind::Bang => {
                let op_tok = self.bump().clone();
                let rhs = self.parse_expr(80)?;
                let span = op_tok.span.merge(rhs.span());
                Ok(Expr::Unary { op: UnaryOp::Not, rhs: Box::new(rhs), span })
            }
            TokenKind::Minus => {
                let op_tok = self.bump().clone();
                let rhs = self.parse_expr(80)?;
                let span = op_tok.span.merge(rhs.span());
                Ok(Expr::Unary { op: UnaryOp::Neg, rhs: Box::new(rhs), span })
            }
            TokenKind::LParen => {
                let l = self.bump().clone();
                let e = self.parse_expr(0)?;
                let r = self.expect("')'", |k| matches!(k, TokenKind::RParen))?;
                let span = l.span.merge(e.span()).merge(r.span);
                /* keep original expression but span could be tracked differently; simplest:
                 ↪  wrap as binary? no */
                Ok(match e {
                    Expr::Int(v, _) => Expr::Int(v, span),
                    Expr::Bool(v, _) => Expr::Bool(v, span),
                    Expr::Str(v, _) => Expr::Str(v, span),
                    Expr::Var(v, _) => Expr::Var(v, span),
                    Expr::Unary { op, rhs, .. } => Expr::Unary { op, rhs, span },
                   Expr::Binary { lhs, op, rhs, .. } => Expr::Binary { lhs, op, rhs, span },
                })
            }
            TokenKind::Eof => Err(ParseError {
                kind: ParseErrorKind::UnexpectedEof,
                span: t.span,
                message: "unexpected end of input".to_string(),
            }),
            _ => Err(ParseError {
                kind: ParseErrorKind::ExpectedExpression,
                span: t.span,
                message: format!("expected expression, found {:?}", t.kind),
            }),
        }
    }

    fn infix_binding_power(&self) -> Option<(u8, u8, BinaryOp)> {
        use BinaryOp::*;
        match self.peek().kind {
            TokenKind::Star => Some((70, 71, Mul)),
            TokenKind::Slash => Some((70, 71, Div)),
            TokenKind::Plus => Some((60, 61, Add)),
            TokenKind::Minus => Some((60, 61, Sub)),

            TokenKind::EqEq => Some((50, 51, Eq)),
            TokenKind::BangEq => Some((50, 51, Ne)),
            TokenKind::Lt => Some((55, 56, Lt)),
            TokenKind::LtEq => Some((55, 56, Le)),
            TokenKind::Gt => Some((55, 56, Gt)),
            TokenKind::GtEq => Some((55, 56, Ge)),

            TokenKind::AndAnd => Some((40, 41, And)),
            TokenKind::OrOr => Some((30, 31, Or)),

            _ => None,
        }
    }
}
```

28.11 Error reporting

28.11.1 Principles of Good Diagnostics

Production-quality diagnostics must:

- include precise source location (span),
- describe what was expected and what was found,
- show a short excerpt of the source with a caret range,
- avoid cascaded nonsense errors (prefer early fail or controlled recovery),
- remain deterministic and testable (string snapshots optional but stable).

28.11.2 Rendering a Span to a Human Message

We implement a simple diagnostic printer that shows the line containing the span and marks the range.

```
pub fn format_span(src: &str, span: Span) -> (usize, usize, String, String) {
    /* returns: line_number(1-based), col_start(1-based), line_text, caret_line */
    let bytes = src.as_bytes();
    let mut line_start = 0usize;
    let mut line_no = 1usize;

    for (idx, b) in bytes.iter().enumerate() {
        if idx >= span.start { break; }
        if *b == b'\n' {
            line_no += 1;
            line_start = idx + 1;
        }
    }

    let line_end = src[line_start..].find('\n').map(|o| line_start +
    ↪  o).unwrap_or(src.len());
    let line_text = src[line_start..line_end].to_string();

    let col_start = span.start.saturating_sub(line_start) + 1;
    let col_end = span.end.saturating_sub(line_start).max(col_start - 1) + 1;

    let mut caret = String::new();
    for _ in 1..col_start { caret.push(' '); }
    let caret_len = (col_end.saturating_sub(col_start)).max(1);
    for _ in 0..caret_len { caret.push('^'); }

    (line_no, col_start, line_text, caret)
}

pub fn render_parse_error(src: &str, e: &ParseError) -> String {
    let (line_no, col, line_text, caret) = format_span(src, e.span);
    format!(
        "error: {}\n --> line {}, col {}\n  |\n  | {}\n  | {}\n”,
        e.message, line_no, col, line_text, caret
    )
}
```

28.11.3 Examples of Error Messages

Illustrate what a user sees and ensure it is stable enough for tests.

```
fn demo_error() {
    let src = "let x = (1 + 2;\n”;
    let toks = Lexer::new(src).tokenize().unwrap();
    let mut p = Parser::new(toks);
    let err = p.parse_program().unwrap_err();
    let msg = render_parse_error(src, &err);
    println!("{}", msg);
}
```

Professional rule: errors must never crash the program; they are part of the normal control flow in tools, compilers, and interpreters.

28.12 Testing strategy

28.12.1 What to Test and Why

A reliable interpreter must be tested at multiple layers:

- Lexer: tokenization correctness, spans, and malformed input handling.
- Parser: precedence, associativity, and structural correctness of the AST.
- Diagnostics: stable, correct location reporting.
- End-to-end: source text → tokens → AST → evaluation.

28.12.2 Lexer tests: correctness and edge cases

Focus on:

- operators and multi-character tokens,
- identifier rules and keyword recognition,
- numeric parsing boundaries,
- strings and escape handling,

- unexpected characters and unterminated string errors.

```
#[test]
fn lex_ops_and_keywords() {
    let src = r#"let x = 10 != 20 && true || false;"#;
    let toks = Lexer::new(src).tokenize().unwrap();
    assert!(matches!(toks[0].kind, TokenKind::Let));
    assert!(matches!(toks[1].kind, TokenKind::Ident(_)));
    assert!(matches!(toks.iter().any(|t| matches!(t.kind, TokenKind::BangEq)), true));
    assert!(matches!(toks.iter().any(|t| matches!(t.kind, TokenKind::AndAnd)), true));
    assert!(matches!(toks.iter().any(|t| matches!(t.kind, TokenKind::OrOr)), true));
}
```

28.12.3 Parser tests: precedence and associativity

Test that `1 + 2 * 3` parses as `1 + (2 * 3)`.

```
fn parse_expr_only(src: &str) -> Expr {
    let toks = Lexer::new(src).tokenize().unwrap();
    let mut p = Parser::new(toks);
    let prog = p.parse_program().unwrap();
    match &prog.stmts[0] {
        Stmt::Expr { expr, .. } => expr.clone(),
        _ => panic!("expected expression statement"),
    }
}

#[test]
fn parse_precedence_mul_over_add() {
    let e = parse_expr_only("1 + 2 * 3");
    match e {
        Expr::Binary { op: BinaryOp::Add, lhs, rhs, .. } => {
            assert!(matches!(*lhs, Expr::Int(1, _)));
            match *rhs {
                Expr::Binary { op: BinaryOp::Mul, .. } => {}
                _ => panic!("expected mul on rhs"),
            }
        }
        _ => panic!("expected add at top level"),
    }
}
```

28.12.4 Diagnostic tests: stable spans

For stable projects, test spans rather than full text rendering:

- verify the error kind,
- verify the span start/end,
- optionally snapshot the full message if you maintain a stable format.

```
#[test]
fn parse_missing_rparen_has_span() {
    let src = "let x = (1 + 2;\n";
    let toks = Lexer::new(src).tokenize().unwrap();
    let mut p = Parser::new(toks);
    let err = p.parse_program().unwrap_err();
    assert!(matches!(err.kind, ParseErrorKind::ExpectedToken));
    assert!(err.span.start <= err.span.end);
}
```

28.12.5 End-to-end tests: the user story

Even before building a full interpreter, you can test parse results and later evaluation results. Once the interpreter exists, add tests that cover:

- correct evaluation,
- variable binding behavior,
- short-circuit logic for && and ||,
- division by zero and error reporting policy,
- string operations if supported.

```
/* Placeholder: once interpreter is implemented */
fn eval_src(_src: &str) -> Result<String, String> {
    Err(”interpreter not implemented yet”.to_string())
}

#[test]
fn e2e_example_placeholder() {
    let _ = eval_src(”let x = 1 + 2 * 3; x”);
}
```

A professional testing strategy treats the language toolchain as a pipeline: each stage is tested in isolation, then the full pipeline is tested end-to-end. This prevents fragile debugging sessions where an error in the lexer appears as a “parser bug,” or an evaluation failure is caused by an earlier span mistake.

Project 2: Mini Key-Value Database

This capstone project designs and implements a small but production-minded key-value database in Rust. The objective is not to compete with industrial databases, but to build a system that demonstrates correct storage layout, safe concurrency, explicit indexing, and defensible benchmarking methodology. Every design choice prioritizes predictability, data integrity, and performance transparency.

28.13 File storage

28.13.1 Storage Model and Design Goals

The database persists key-value pairs on disk using a simple append-only log structure. The goals of the storage layer are:

- durability through explicit file writes and flush boundaries,
- crash resilience by avoiding in-place updates,
- simple recovery by replaying the log,
- predictable I/O patterns suitable for benchmarking.

We store records as length-prefixed binary entries:

- key length (u32),
- value length (u32),
- key bytes,
- value bytes.

28.13.2 Record Encoding

Binary encoding avoids parsing overhead and reduces storage size.

```rust
use std::io::{self, Write};

pub fn write_record<W: Write>(mut w: W, key: &[u8], value: &[u8]) -> io::Result<()> {
    let klen = key.len() as u32;
    let vlen = value.len() as u32;

    w.write_all(&klen.to_le_bytes())?;
    w.write_all(&vlen.to_le_bytes())?;
    w.write_all(key)?;
    w.write_all(value)?;
    Ok(())
}
```

This format allows sequential scanning and recovery without external metadata.

28.13.3 Append-Only Log and Durability

All writes append to a single data file:

- no overwrites,
- no partial record updates,

- minimal seek overhead.

```
use std::fs::{File, OpenOptions};
use std::io::{self, BufWriter};

pub fn open_data_file(path: &str) -> io::Result<BufWriter<File>> {
    let f = OpenOptions::new()
        .create(true)
        .append(true)
        .open(path)?;
    Ok(BufWriter::new(f))
}
```

Durability boundaries are explicit: the database decides when to flush buffers and call `sync_all` depending on its consistency policy.

28.14 Indexing

28.14.1 In-Memory Index Structure

To support fast reads, the database maintains an in-memory index mapping keys to file offsets. The index is rebuilt at startup by scanning the log.
Design constraints:

- index must be fast and deterministic,
- index rebuild must be linear in file size,
- index entries must point to immutable data.

```
use std::collections::HashMap;
```

```
#[derive(Debug, Clone, Copy)]
pub struct RecordPos {
    pub offset: u64,
    pub key_len: u32,
    pub val_len: u32,
}

pub type Index = HashMap<Vec<u8>, RecordPos>;
```

28.14.2 Index Construction During Recovery

At startup, the database scans the log and updates the index so that the last occurrence of a key wins.

```
use std::io::{Read, Seek, SeekFrom};

pub fn rebuild_index<R: Read + Seek>(mut r: R) -> std::io::Result<Index> {
    let mut index = Index::new();
    let mut offset = 0u64;

    loop {
        let mut hdr = [0u8; 8];
        if r.read_exact(&mut hdr).is_err() {
            break;
        }

        let klen = u32::from_le_bytes(hdr[0..4].try_into().unwrap());
        let vlen = u32::from_le_bytes(hdr[4..8].try_into().unwrap());

        let mut key = vec![0u8; klen as usize];
        r.read_exact(&mut key)?;
        r.seek(SeekFrom::Current(vlen as i64))?;
        index.insert(
            key,
            RecordPos { offset, key_len: klen, val_len: vlen },
        );

        offset += 8 + klen as u64 + vlen as u64;
    }

    Ok(index)
}
```

This approach guarantees correctness even after crashes, at the cost of startup time proportional to data size.

28.15 Concurrency

28.15.1 Concurrency Model

The database supports concurrent readers and a single writer. Design principles:

- readers must never block each other,
- writers serialize modifications to storage and index,
- readers observe a consistent view of the index.

We use a read-write lock to protect the index and a mutex for file writes.

28.15.2 Shared State

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

```
pub struct DbState {
    pub index: RwLock<Index>,
    pub writer: Mutex<std::io::BufWriter<std::fs::File>>,
}

pub type SharedDb = Arc<DbState>;
```

28.15.3 Concurrent Reads

Reads acquire a shared lock on the index and perform file I/O without modifying state.

```
use std::fs::File;
use std::io::{Read, Seek, SeekFrom};

pub fn get(db: &SharedDb, key: &[u8]) -> std::io::Result<Option<Vec<u8>>> {
    let idx = db.index.read().unwrap();
    let pos = match idx.get(key) {
        Some(p) => *p,
        None => return Ok(None),
    };
    drop(idx);

    let mut f = File::open("data.log")?;
    f.seek(SeekFrom::Start(pos.offset + 8 + pos.key_len as u64))?;

    let mut val = vec![0u8; pos.val_len as usize];
    f.read_exact(&mut val)?;
    Ok(Some(val))
}
```

28.15.4 Serialized Writes

Writes acquire exclusive access to the writer and update the index atomically.

```
pub fn put(db: &SharedDb, key: Vec<u8>, value: Vec<u8>) -> std::io::Result<()> {
    let mut writer = db.writer.lock().unwrap();
    let offset = writer.get_ref().metadata()?.len();

    write_record(&mut *writer, &key, &value)?;
    writer.flush()?;

    let mut idx = db.index.write().unwrap();
    idx.insert(
        key,
        RecordPos {
            offset,
            key_len: key.len() as u32,
            val_len: value.len() as u32,
        },
    );
    Ok(())
}
```

This design guarantees thread safety with minimal locking overhead for reads.

28.16 Benchmarking

28.16.1 Benchmarking Philosophy

Benchmarks must reflect realistic usage patterns and isolate costs. Key rules:

- separate cold-start from steady-state measurements,
- measure reads and writes independently,
- avoid measuring logging or debug output,
- control filesystem cache effects when possible.

28.16.2 Microbenchmarks for Core Operations

We benchmark:

- write throughput (records per second),
- read latency (indexed lookups),
- index rebuild time (startup cost).

```rust
use std::time::Instant;

pub fn bench_put(db: &SharedDb, n: usize) {
    let start = Instant::now();
    for i in 0..n {
        let k = format!("key{}", i).into_bytes();
        let v = vec![0u8; 128];
        put(db, k, v).unwrap();
    }
    let elapsed = start.elapsed();
    println!("put: {} ops in {:?}", n, elapsed);
}
```

28.16.3 Read Benchmark

```rust
pub fn bench_get(db: &SharedDb, n: usize) {
    let start = Instant::now();
    for i in 0..n {
        let k = format!("key{}", i).into_bytes();
        let _ = get(db, &k).unwrap();
    }
    let elapsed = start.elapsed();
    println!("get: {} ops in {:?}", n, elapsed);
}
```

28.16.4 Interpreting Results

When evaluating results:

- compare sequential vs concurrent read performance,
- observe write amplification due to flushing policy,
- evaluate index rebuild cost relative to data size,
- track how performance scales with value size.

A professional benchmark report explains *why* numbers look the way they do. Raw throughput without context is meaningless.

28.16.5 Project Outcome

By completing this project, you will have:

- implemented durable file-backed storage,
- built an index suitable for fast lookups,
- designed a safe and efficient concurrency model,
- learned to benchmark storage systems responsibly.

This mini database is intentionally simple, but its architecture mirrors the core ideas behind real-world storage engines. The same principles scale upward: explicit invariants, append-only design, careful concurrency, and honest performance measurement.

Project 3: High-Performance Network Server

This capstone project builds a high-performance network server in Rust with an explicit focus on production behavior: predictable latency, controlled memory growth, well-defined backpressure, and observable runtime behavior through metrics and logging. The server is intentionally small but engineered like a real system: it must remain stable under load, reject work safely when overloaded, and provide operational visibility.

28.17 Async I/O

28.17.1 Design Goals for Async Networking

Async I/O is not “faster by magic.” It is a concurrency model that enables:

- high connection counts without one thread per socket,
- efficient multiplexing of I/O readiness events,
- explicit control over scheduling, batching, and resource limits.

Production goals:

- avoid blocking calls on async tasks,

- bound memory per connection,
- avoid unbounded task spawning,
- isolate expensive work from the I/O reactor.

28.17.2 Protocol: Minimal Request/Response

We implement a simple line protocol:

- client sends one UTF-8 line ending with `\n`,
- server responds with one UTF-8 line ending with `\n`,
- commands: `PING`, `ECHO <text>`, `STATS`.

28.17.3 Async Server Skeleton

This skeleton accepts TCP connections and handles them concurrently. It also enforces a hard limit on concurrent connections using a semaphore.

```rust
use std::sync::Arc;
use tokio::net::{TcpListener, TcpStream};
use tokio::sync::Semaphore;

pub struct ServerConfig {
    pub bind: String,
    pub max_conns: usize,
}

pub async fn run_server(cfg: ServerConfig) -> std::io::Result<()> {
    let listener = TcpListener::bind(&cfg.bind).await?;
    let permits = Arc::new(Semaphore::new(cfg.max_conns));
    loop {
        let (sock, _) = listener.accept().await?;
        let permit = match permits.clone().try_acquire_owned() {
            Ok(p) => p,
            Err(_) => {
                /* overloaded: drop connection immediately */
                drop(sock);
                continue;
            }
        };

        tokio::spawn(async move {
            let _permit = permit; /* released when task ends */
            let _ = handle_conn(sock).await;
        });
    }
}
```

Professional rule: limiting concurrency is mandatory. Without limits, overload becomes memory exhaustion.

28.17.4 Connection Handler with Buffered I/O

We use buffered reads and explicit maximum line length to prevent unbounded memory growth.

```
use tokio::io::{AsyncBufReadExt, AsyncWriteExt, BufReader};

const MAX_LINE: usize = 8 * 1024;

async fn handle_conn(sock: TcpStream) -> std::io::Result<()> {
    let (r, mut w) = sock.into_split();
    let mut reader = BufReader::new(r);
    let mut line = String::new();
    loop {
        line.clear();
        let n = reader.read_line(&mut line).await?;
        if n == 0 {
            return Ok(());
        }

        if line.len() > MAX_LINE {
            w.write_all(b”ERR line too long\n”).await?;
            continue;
        }

        let resp = dispatch(line.trim_end_matches('\n').trim_end_matches('\r')).await;
        w.write_all(resp.as_bytes()).await?;
        w.write_all(b”\n”).await?;
    }
}

async fn dispatch(cmd: &str) -> String {
    if cmd == ”PING” {
        ”PONG”.to_string()
    } else if let Some(rest) = cmd.strip_prefix(”ECHO ”) {
        rest.to_string()
    } else if cmd == ”STATS” {
        ”OK”.to_string()
    } else {
        ”ERR unknown command”.to_string()
    }
}
```

Key production techniques shown above:

- **buffered reading** reduces syscalls and improves throughput,

- **line length limit** prevents memory abuse and accidental OOM,
- **structured dispatch** isolates protocol logic from I/O handling.

28.17.5 Avoiding Blocking Work in Async Tasks

Any CPU-heavy or blocking I/O work must not run on async tasks that share the I/O runtime threads. A safe pattern is to isolate blocking work:

```
async fn expensive_compute(input: String) -> String {
    let out = tokio::task::spawn_blocking(move || {
        /* CPU-heavy or blocking logic */
        input.chars().rev().collect::<String>()
    }).await;

    match out {
        Ok(v) => v,
        Err(_) => "ERR internal".to_string(),
    }
}
```

This preserves reactor responsiveness and avoids latency spikes caused by blocking.

28.18 Backpressure

28.18.1 What Backpressure Means

Backpressure is the ability of a system to slow down, shed load, or reject work when demand exceeds capacity. A network server must apply backpressure at multiple layers:

- **connection admission**: limit concurrent clients,
- **per-connection input**: limit buffer sizes and request sizes,

- **work queues**: bound outstanding tasks,
- **output**: avoid unbounded outbound buffering for slow clients.

28.18.2 Admission Control via Semaphore

We already applied admission control using `Semaphore`. This prevents the typical failure mode: unbounded tasks and memory pressure.

28.18.3 Bounded Work Queue Pattern

For commands that require heavy processing, route them through a bounded channel. When the queue is full, reject work immediately.

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

pub struct WorkItem {
    pub cmd: String,
    pub reply: tokio::sync::oneshot::Sender<String>,
}

pub fn make_worker_pool(capacity: usize) -> mpsc::Sender<WorkItem> {
    let (tx, mut rx) = mpsc::channel::<WorkItem>(capacity);

    tokio::spawn(async move {
        while let Some(item) = rx.recv().await {
            let out = process_command(item.cmd).await;
            let _ = item.reply.send(out);
        }
    });

    tx
}
```

```
async fn process_command(cmd: String) -> String {
    if cmd.starts_with("ECHO ") {
        cmd[5..].to_string()
    } else {
        "OK".to_string()
    }
}
```

Submitting work with backpressure:

```
use tokio::sync::oneshot;

async fn dispatch_with_queue(
    tx: tokio::sync::mpsc::Sender<WorkItem>,
    cmd: &str
) -> String {
    let (reply_tx, reply_rx) = oneshot::channel();
    let item = WorkItem { cmd: cmd.to_string(), reply: reply_tx };

    /* try_send applies immediate backpressure */
    if tx.try_send(item).is_err() {
        return "ERR overloaded".to_string();
    }

    match reply_rx.await {
        Ok(v) => v,
        Err(_) => "ERR internal".to_string(),
    }
}
```

28.18.4 Slow Client Handling: Output Backpressure

A slow client can cause output buffers to grow. Practical strategies:

- enforce per-connection write timeouts,
- limit outstanding responses per connection,
- drop connections that cannot keep up.

```rust
use tokio::time::{timeout, Duration};
use tokio::io::AsyncWriteExt;

async fn write_line_with_timeout<W: AsyncWriteExt + Unpin>(
    w: &mut W,
    line: &str
) -> std::io::Result<()> {
    let fut = async {
        w.write_all(line.as_bytes()).await?;
        w.write_all(b"\n").await?;
        w.flush().await?;
        Ok::<(), std::io::Error>(())
    };

    match timeout(Duration::from_millis(200), fut).await {
        Ok(r) => r,
        Err(_) => {
            /* treat as slow client */
            Err(std::io::Error::new(std::io::ErrorKind::TimedOut, "write timeout"))
        }
    }
}
```

Backpressure is not optional. Without it, the server does not fail gracefully; it fails catastrophically.

28.19 Metrics and logging

28.19.1 Observability Goals

A production server must explain its behavior under load:

- what is the request rate,
- what is the latency distribution,
- how many connections are active,
- how often overload triggers,
- what errors occur and where.

This requires two complementary tools:

- **logging** for discrete events and debugging,
- **metrics** for quantitative monitoring and alerting.

28.19.2 Structured Logging with Context

Prefer structured logging with stable fields:

- `conn_id`, `peer`, `cmd`, `result`, `latency_us`,
- avoid printing whole payloads unless needed,
- keep logs bounded and rate-limited for noisy paths.

```
use std::time::Instant;

async fn handle_one_command(cmd: &str) -> String {
    let start = Instant::now();
    let out = dispatch(cmd).await;
    let elapsed = start.elapsed();

    /* Conceptual: replace with your logging backend */
    println!("event=cmd cmd=\"{}\" ok={} latency_us={}",
        cmd,
        !out.starts_with("ERR"),
        elapsed.as_micros()
    );

    out
}
```

28.19.3 Metrics: Counters, Gauges, Histograms

Define a minimal metrics model:

- **counter**: total requests, errors, overload rejections,
- **gauge**: active connections, queue depth,
- **histogram**: request latency distribution.

A simple internal metrics registry can be implemented with atomics. This keeps overhead low and avoids locking hot paths.

```
use std::sync::atomic::{AtomicU64, Ordering};
use std::sync::Arc;

#[derive(Default)]
```

```
pub struct Metrics {
    pub requests: AtomicU64,
    pub errors: AtomicU64,
    pub overloads: AtomicU64,
    pub active_conns: AtomicU64,
}

pub type SharedMetrics = Arc<Metrics>;

pub fn inc(x: &AtomicU64) { x.fetch_add(1, Ordering::Relaxed); }
pub fn set(x: &AtomicU64, v: u64) { x.store(v, Ordering::Relaxed); }
```

Integrate metrics into server control flow:

```
async fn handle_conn_with_metrics(sock: TcpStream, m: SharedMetrics) -> std::io::Result<()>
↪ {
    inc(&m.active_conns);
    let r = handle_conn(sock).await;
    m.active_conns.fetch_sub(1, std::sync::atomic::Ordering::Relaxed);
    r
}
```

28.19.4 Measuring Latency with Low Overhead

Record per-request latency. In high-performance servers, prefer:

- coarse histograms (bucketed),
- sampling (record every Nth request),
- separate "debug mode" vs "production mode" overhead.

```
use std::time::Instant;

```

```
pub fn record_latency_us(sample_mask: u64, n: u64, start: Instant) -> Option<u128> {
    /* sample 1 out of (sample_mask+1) if mask is 1023 => 1/1024 sampling */
    if (n & sample_mask) == 0 {
        Some(start.elapsed().as_micros())
    } else {
        None
    }
}
```

28.19.5 A Practical STATS Command

Expose operational data via a STATS command. Keep it stable and machine-readable.

```
pub fn render_stats(m: &Metrics) -> String {
    let req = m.requests.load(std::sync::atomic::Ordering::Relaxed);
    let err = m.errors.load(std::sync::atomic::Ordering::Relaxed);
    let ov  = m.overloads.load(std::sync::atomic::Ordering::Relaxed);
    let ac  = m.active_conns.load(std::sync::atomic::Ordering::Relaxed);

    format!("requests={} errors={} overloads={} active_conns={}", req, err, ov, ac)
}
```

This is intentionally simple but operationally useful. Real systems export metrics to collectors, but the architectural principle is the same: the server must be measurable.

28.19.6 Project Completion Checklist

A professional-grade outcome should satisfy:

- stable async I/O loop without blocking on the reactor threads,
- bounded per-connection memory (line limits, bounded buffers),
- bounded global concurrency (admission control and bounded queues),

- explicit overload behavior (reject, timeout, or drop),
- logging with stable fields and minimal hot-path overhead,
- metrics that quantify load, errors, and capacity limits.

If the server remains stable and observable under synthetic overload, the project achieved its engineering goal: correctness first, performance second, and operational clarity always.

Project 4: Low-Level Library with Safe Abstractions

This capstone project builds a low-level memory utility library in Rust that exposes safe, ergonomic abstractions while using carefully-audited `unsafe` internally. The project demonstrates a core systems skill: designing an API that is hard to misuse, fast in hot paths, and explicit about invariants.

We implement a small **bump arena allocator** and a **ring buffer** as two complementary building blocks:

- the arena optimizes allocation-heavy workloads by amortizing allocations and freeing in bulk,
- the ring buffer provides bounded, predictable, cache-friendly storage for streaming pipelines.

28.20 Arena allocator or ring buffer

28.20.1 Design Goals and Non-goals

Goals:

- predictable performance (constant-time fast paths),

- bounded memory behavior (explicit capacity and reset semantics),
- clear safety invariants (no hidden aliasing or lifetime violations),
- minimal overhead compared to naive allocation strategies.

Non-goals:

- implementing a general-purpose allocator replacement,
- supporting deallocation of individual objects (arena frees in bulk),
- providing lock-free multi-producer/multi-consumer semantics (ring buffer in this project is single-producer/single-consumer by default).

28.20.2 Bump Arena: Core Idea

A bump arena owns a contiguous memory region. Allocation is performed by moving an offset forward ("bumping") with alignment. All allocations are freed at once by resetting the arena. Key properties:

- extremely fast allocation (pointer arithmetic),
- excellent cache locality,
- no per-allocation free cost,
- ideal for request-scoped or frame-scoped allocations.

28.20.3 Arena Implementation

We implement an arena backed by a `Vec<u8>` with an offset. The public API stays safe; `unsafe` is confined to internal pointer operations.

```rust
use std::alloc::Layout;
use std::ptr::NonNull;

#[derive(Debug)]
pub struct Arena {
    buf: Vec<u8>,
    pos: usize,
}

impl Arena {
    pub fn with_capacity(cap: usize) -> Self {
        Self { buf: vec![0u8; cap], pos: 0 }
    }

    pub fn capacity(&self) -> usize { self.buf.len() }
    pub fn used(&self) -> usize { self.pos }
    pub fn remaining(&self) -> usize { self.capacity().saturating_sub(self.pos) }

    pub fn reset(&mut self) {
        self.pos = 0;
    }

    fn align_up(x: usize, align: usize) -> usize {
        debug_assert!(align.is_power_of_two());
        (x + (align - 1)) & !(align - 1)
    }

    pub fn alloc_bytes(&mut self, layout: Layout) -> Option<NonNull<u8>> {
        let start = Self::align_up(self.pos, layout.align());
        let end = start.checked_add(layout.size())?;
        if end > self.buf.len() {
            return None;
        }
        self.pos = end;

        let p = unsafe { self.buf.as_mut_ptr().add(start) };
        NonNull::new(p)
    }

    pub fn alloc_slice<T: Copy>(&mut self, n: usize) -> Option<&mut [T]> {
        let layout = Layout::array::<T>(n).ok()?;
        let p = self.alloc_bytes(layout)?;
        let ptr = p.as_ptr() as *mut T;
        let slice = unsafe { std::slice::from_raw_parts_mut(ptr, n) };
        Some(slice)
    }
}
```

This arena allocates raw bytes and typed slices. It does not drop or run destructors for arena-allocated objects; therefore, we restrict the safe typed API to `Copy` values. If you extend this to non-Copy types, you must track drops explicitly (and the safety proof becomes more complex).

28.20.4 Arena Usage Example

```
fn arena_demo() {
    let mut a = Arena::with_capacity(1024);

    let xs: &mut [u32] = a.alloc_slice::<u32>(128).unwrap();
    for (i, x) in xs.iter_mut().enumerate() {
        *x = i as u32;
    }

    assert_eq!(a.used() > 0, true);
    a.reset();
    assert_eq!(a.used(), 0);
}
```

28.20.5 Ring Buffer: Core Idea

A ring buffer is a fixed-capacity queue implemented on a circular array. It is ideal for streaming and real-time pipelines because:

- it provides bounded memory usage,
- it is cache-friendly,
- push/pop are O(1),
- it naturally expresses backpressure (full vs empty).

28.20.6 Ring Buffer Implementation (SPSC)

We implement a single-producer/single-consumer ring buffer using indices. The safe API returns errors when full/empty. Internally, we use careful indexing and avoid exposing invalid references.

```
#[derive(Debug)]
pub enum RbError {
    Full,
    Empty,
}

pub struct RingBuffer<T: Copy> {
    buf: Vec<T>,
    head: usize,
    tail: usize,
    full: bool,
}
```

```rust
impl<T: Copy> RingBuffer<T> {
    pub fn with_capacity(cap: usize, init: T) -> Self {
        assert!(cap > 0);
        Self {
            buf: vec![init; cap],
            head: 0,
            tail: 0,
            full: false,
        }
    }

    pub fn capacity(&self) -> usize { self.buf.len() }

    pub fn is_empty(&self) -> bool {
        !self.full && self.head == self.tail
    }

    pub fn is_full(&self) -> bool { self.full }

    pub fn len(&self) -> usize {
        if self.full {
            self.capacity()
        } else if self.tail >= self.head {
            self.tail - self.head
        } else {
            self.capacity() - (self.head - self.tail)
        }
    }

    pub fn push(&mut self, v: T) -> Result<(), RbError> {
        if self.full {
            return Err(RbError::Full);
        }
        self.buf[self.tail] = v;
        self.tail = (self.tail + 1) % self.capacity();
        self.full = self.tail == self.head;
        Ok(())
    }

    pub fn pop(&mut self) -> Result<T, RbError> {
        if self.is_empty() {
            return Err(RbError::Empty);
        }
        let v = self.buf[self.head];
        self.head = (self.head + 1) % self.capacity();
        self.full = false;
        Ok(v)
    }
}
```

28.20.7 Ring Buffer Usage Example

```
fn ring_demo() {
    let mut rb = RingBuffer::with_capacity(4, 0u32);

    rb.push(1).unwrap();
    rb.push(2).unwrap();
    rb.push(3).unwrap();
    rb.push(4).unwrap();
    assert!(rb.push(5).is_err()); /* full */

    assert_eq!(rb.pop().unwrap(), 1);
    assert_eq!(rb.pop().unwrap(), 2);
    rb.push(5).unwrap();
    assert_eq!(rb.len(), 3);
}
```

28.21 Safety validation

28.21.1 Safety Philosophy: Constrain `unsafe` to Small, Auditable Blocks

A professional Rust low-level library follows these principles:

- expose only safe APIs unless `unsafe` is unavoidable for correctness,
- keep `unsafe` blocks minimal and local,
- document invariants that justify each `unsafe` operation,
- design APIs that make invalid states unrepresentable where possible.

28.21.2 Arena Safety Invariants

For the arena implementation above, safety depends on:

- the underlying buffer does not move while references into it exist,
- alignment is respected for typed allocations,
- returned slices do not outlive the arena,
- the arena is not reset while references into it are still used.

In the provided design, the borrow checker already enforces the last point:

- `alloc_slice` takes `&mut self` and returns `&mut [T]`,
- you cannot call `reset(&mut self)` while that slice is alive.

28.21.3 Ring Buffer Safety Invariants

For the ring buffer:

- indices are always in range `0..capacity`,
- full/empty state is represented without ambiguity,
- the API never exposes aliasing mutable references to the same element.

We keep the type constrained to `Copy` to avoid drop-order and partially-initialized element complexity. Extending to general `T` requires careful management of initialization and drop (commonly via `MaybeUninit<T>`).

28.21.4 Testing Safety Properties

Even for low-level components, many "unsafe-adjacent" bugs are detectable through tests:

- boundary conditions (exact capacity fills, wrap-around behavior),
- randomized sequences of push/pop with invariant checks,
- arena alignment checks for multiple types.

```rust
#[test]
fn ring_buffer_invariants() {
    let mut rb = RingBuffer::with_capacity(3, 0u32);
    assert!(rb.is_empty());
    assert_eq!(rb.len(), 0);

    rb.push(10).unwrap();
    rb.push(20).unwrap();
    rb.push(30).unwrap();
    assert!(rb.is_full());
    assert_eq!(rb.len(), 3);

    assert_eq!(rb.pop().unwrap(), 10);
    assert!(!rb.is_full());
    rb.push(40).unwrap();

    /* len always within 0..=capacity */
    assert!(rb.len() <= rb.capacity());
}

#[test]
fn arena_alignment_basic() {
    let mut a = Arena::with_capacity(256);

    let _x: &mut [u32] = a.alloc_slice::<u32>(1).unwrap();
    let _y: &mut [u64] = a.alloc_slice::<u64>(1).unwrap();

    assert!(a.used() <= a.capacity());
}
```

28.22 Performance comparison

28.22.1 What We Compare

We compare:

- arena allocation vs heap allocation (`Vec` growth / `Box`),
- ring buffer vs `VecDeque` in bounded queue usage,
- throughput and latency under representative workloads.

28.22.2 Benchmarking Rules

Benchmarks must:

- avoid measuring debug output or allocations unrelated to the core operation,
- warm up caches and run multiple iterations,
- measure both throughput and per-operation latency where useful,
- validate correctness during benchmarking (at least sanity checks).

28.22.3 Arena vs Heap Allocation Benchmark

This benchmark measures allocation of many small fixed-size buffers, comparing:

- heap path: `Vec::with_capacity` per allocation,
- arena path: `alloc_slice` from preallocated buffer.

```rust
use std::time::Instant;

fn bench_heap_alloc(n: usize) -> usize {
    let start = Instant::now();
    let mut checksum = 0usize;

    for i in 0..n {
        let mut v = Vec::<u32>::with_capacity(16);
        for j in 0..16 {
            v.push((i as u32) ^ (j as u32));
        }
        checksum ^= v[0] as usize;
    }
    let _elapsed = start.elapsed();
    checksum
}

fn bench_arena_alloc(n: usize) -> usize {
    let mut a = Arena::with_capacity(n * 16 * 4 + 128);
    let start = Instant::now();
    let mut checksum = 0usize;

    for i in 0..n {
        let xs = a.alloc_slice::<u32>(16).unwrap();
        for (j, x) in xs.iter_mut().enumerate() {
            *x = (i as u32) ^ (j as u32);
        }
        checksum ^= xs[0] as usize;
    }

    let _elapsed = start.elapsed();
    checksum
}
```

Interpretation guidance:

- The arena should reduce allocator overhead and improve cache locality.
- The heap path may pay repeated allocation and potential fragmentation costs.
- If the arena capacity is insufficient, allocation fails explicitly rather than silently growing.

28.22.4 Ring Buffer vs VecDeque Benchmark

We compare bounded push/pop patterns common in streaming pipelines. A ring buffer avoids capacity growth and keeps memory fixed.

```rust
use std::collections::VecDeque;
use std::time::Instant;

fn bench_ring(n: usize) -> u32 {
    let mut rb = RingBuffer::with_capacity(1024, 0u32);
    let start = Instant::now();
    let mut acc = 0u32;

    for i in 0..n as u32 {
        if rb.push(i).is_ok() {
            if let Ok(x) = rb.pop() {
                acc ^= x;
            }
        }
    }

    let _elapsed = start.elapsed();
    acc
}

fn bench_vecdeque(n: usize) -> u32 {
    let mut q = VecDeque::<u32>::with_capacity(1024);
    let start = Instant::now();
    let mut acc = 0u32;

    for i in 0..n as u32 {
        if q.len() < 1024 {
            q.push_back(i);
        }
        if let Some(x) = q.pop_front() {
            acc ^= x;
        }
    }
    let _elapsed = start.elapsed();
    acc
}
```

28.22.5 Interpreting Performance Results

A professional performance discussion includes:

- **Throughput**: operations per second (steady-state).
- **Tail latency**: worst-case spikes caused by allocation or resizing.
- **Memory behavior**: bounded vs unbounded growth.
- **Determinism**: stable performance vs occasional allocator-induced jitter.

Expected qualitative outcomes:

- The arena typically provides faster, more predictable allocation for many small objects when reset in bulk.
- The ring buffer provides bounded, constant-time queue operations with stable memory usage.
- General-purpose structures may be competitive in average throughput but can exhibit higher variance under pressure (resizing, allocator contention).

28.22.6 Project Completion Checklist

A professional-grade completion should satisfy:

- safe public APIs with minimal internal `unsafe`,

- documented invariants that justify each `unsafe` operation,
- tests that validate boundary conditions and key invariants,
- clear performance measurements with defensible methodology,
- a written comparison explaining both speed and predictability trade-offs.

This project demonstrates the essence of systems Rust: use low-level power internally, but expose safe, well-specified abstractions that make correct usage the default.

Appendices

Appendices

Appendix A: Rust vs C++ — Conceptual Mapping for Systems Engineers

This appendix maps core systems-programming concepts between Rust and Modern C++. It is written for engineers who already think in terms of CPU, memory, concurrency, ABIs, and performance envelopes, and who want a precise mental translation layer rather than beginner-level language tutorials.

How to Read This Mapping

- **C++ and Rust are both systems languages**: both can do bare-metal style work, both can hit predictable performance, both can interoperate with C ABIs, and both require discipline for correctness under concurrency and undefined behavior boundaries.
- **Rust shifts more correctness checks to compile time**: ownership/borrowing, aliasing rules, and certain thread-safety rules are enforced by the type system; C++ relies more on conventions, reviews, sanitizers, and testing.
- **Rust still allows `unsafe`**: the difference is that unsafety is localized and must be justified by explicit invariants; C++ has no syntactic boundary for "safe vs unsafe," so the whole

program is effectively “unsafe” by default.

Memory Ownership and Lifetime Model

The biggest conceptual difference is ownership and aliasing rules.

- In C++ you can express ownership conventionally (`unique_ptr`, RAII, “who frees what”), but the compiler does not enforce exclusivity of mutable aliasing.
- In Rust the borrow checker enforces a core rule: at any time, you can have either:
  - one mutable reference (`&mut T`), or
  - any number of shared references (`&T`),

  but not both in a way that violates aliasing and mutation safety.

RAII vs Drop

Both languages use deterministic resource release.

- C++: destructors run at scope exit; movable types require correct move constructors/assignments.
- Rust: `Drop` runs at scope exit; moves are the default (a move is a bitwise move plus “old binding becomes unusable”).

```
/* C++: RAII ownership */
#include <memory>
#include <vector>

struct File {
    File(const char* path);
    ~File();
    void write(const void* p, size_t n);
};

void f() {
    auto file = std::make_unique<File>(”out.bin”);
    std::vector<int> v(1024);
    file->write(v.data(), v.size() * sizeof(int));
} /* destructor runs here */
```

```
/* Rust: Drop-based ownership */
struct File { /* ... */ }
impl File {
    fn new(path: &str) -> Self { /* ... */ unimplemented!() }
    fn write(&mut self, buf: &[u8]) { /* ... */ }
}

fn f() {
    let mut file = File::new(”out.bin”);
    let v = vec![0u8; 4096];
    file.write(&v);
} /* drop runs here */
```

Borrowing vs References and Pointers

C++ has multiple reference/pointer forms; Rust distinguishes safe references from raw pointers.

- C++:
  - `T&` / `const T&` are references (aliasing not restricted).
  - `T*` raw pointers (nullable, arithmetic).
  - ownership is a convention unless using smart pointers.
- Rust:

  - &T shared borrow (read-only aliasing allowed).
  - &mut T exclusive borrow (unique mutable access).
  - *const T / *mut T raw pointers (require unsafe to dereference).

```
/* C++: aliasing is allowed by default */
void inc(int* p) { (*p)++; }

void demo() {
    int x = 0;
    int& a = x;
    int& b = x; /* two aliases */
    a++;
    b++;
}
```

```
/* Rust: exclusive mutation is enforced */
fn demo() {
    let mut x = 0;

    let a = &mut x;
    /* let b = &mut x; */ /* error: cannot borrow `x` as mutable more than once */
    *a += 1;
}
```

Move Semantics and Copy Semantics

Both languages have moving, but the defaults differ.

- C++: types can be copyable and movable; moves may still leave valid-but-unspecified states unless carefully designed.
- Rust: moves are default for most types; Copy types are implicitly copied (small scalars, simple POD-like structs).

```
/* C++: copy vs move depends on type and operations */
#include <string>
std::string g() { return "hello"; } /* move elision / NRVO possible */
```

```
/* Rust: move by default, copy only if Copy */
fn g() -> String { "hello".to_string() }
```

Error Handling Philosophy

Both ecosystems support multiple error strategies, but the defaults differ.

- C++ common patterns:
  - exceptions (with cost/model trade-offs; often avoided in low-latency systems),
  - error codes / `std::optional` / `std::expected` (C++23),
  - out-parameters + status return.
- Rust common patterns:
  - `Result<T, E>` for recoverable errors,
  - `panic!` for bugs/invariants (like assertions),
  - `Option<T>` for “present or absent.”

```
/* C++23: expected as a Result-like tool */
#include <expected>
#include <string>

std::expected<int, std::string> parse_i32(const std::string& s);

void use_it(const std::string& s) {
    auto r = parse_i32(s);
    if (!r) {
        /* handle r.error() */
        return;
    }
    int v = *r;
}
```

```
/* Rust: Result and ? operator */
fn parse_i32(s: &str) -> Result<i32, String> { /* ... */ unimplemented!() }

fn use_it(s: &str) -> Result<(), String> {
    let v = parse_i32(s)?;
    let _ = v;
    Ok(())
}
```

Generics and Templates

Both languages do static polymorphism well.

- C++ templates:
  - extremely powerful metaprogramming,
  - concepts (C++20) improve constraints and diagnostics,
  - errors can be complex but controllable with concepts.
- Rust generics:
  - trait bounds for constraints,
  - monomorphization (like templates) for performance,
  - coherence/orphan rules restrict impls to maintain global consistency.

```
/* C++20 concepts */
#include <concepts>

template <typename T>
concept Addable = requires(T a, T b) { a + b; };

template <Addable T>
T add(T a, T b) { return a + b; }
```

```
/* Rust trait bounds */
use std::ops::Add;

fn add<T: Add<Output = T>>(a: T, b: T) -> T { a + b }
```

Dynamic Polymorphism

Both support virtual dispatch; Rust makes it explicit.

- C++: `virtual` functions, base pointers/references, vtables.
- Rust: trait objects dyn `Trait`, vtables, explicit boxing/references.

```
/* C++: virtual dispatch */
struct Shape { virtual ~Shape() = default; virtual double area() const = 0; };

double total(const std::vector<Shape*>& xs) {
    double s = 0;
    for (auto* p : xs) s += p->area();
    return s;
}
```

```
/* Rust: trait objects */
trait Shape { fn area(&self) -> f64; }
```

```
fn total(xs: &[&dyn Shape]) -> f64 {
    xs.iter().map(|s| s.area()).sum()
}
```

Concurrency Model and Data Races

The core conceptual shift: Rust elevates thread-safety into type checking.

- C++:
  - data races are undefined behavior; correctness depends on discipline and tooling,
  - `std::mutex`, atomics, and memory orders must be used correctly.
- Rust:
  - types must be `Send` to move across threads and `Sync` to be shared,
  - safe shared mutation requires synchronization types (e.g., `Mutex`, `RwLock`, atomics),
  - "shared mutable state" is intentionally harder.

```
/* C++: correct, but compiler will not stop you from racing */
#include <mutex>
static int g = 0;
static std::mutex m;

void inc() {
    std::lock_guard<std::mutex> lock(m);
    g++;
}
```

```
/* Rust: shared mutation requires Sync types */
use std::sync::{Arc, Mutex};
use std::thread;
```

```
fn demo() {
    let g = Arc::new(Mutex::new(0i32));

    let mut th = Vec::new();
    for _ in 0..4 {
        let g2 = Arc::clone(&g);
        th.push(thread::spawn(move || {
            let mut x = g2.lock().unwrap();
            *x += 1;
        }));
    }
    for t in th { t.join().unwrap(); }
}
```

Atomics and Memory Ordering

Both languages expose low-level atomics with memory orderings.

- C++: `std::atomic<T>`, `memory_order` variants, fences.
- Rust: `std::sync::atomic` types, `Ordering` variants, fences.

```
/* C++: relaxed atomic increment */
#include <atomic>
std::atomic<unsigned> ctr{0};

void tick() {
    ctr.fetch_add(1, std::memory_order_relaxed);
}
```

```
/* Rust: relaxed atomic increment */
use std::sync::atomic::{AtomicU32, Ordering};
static CTR: AtomicU32 = AtomicU32::new(0);
```

```
fn tick() {
    CTR.fetch_add(1, Ordering::Relaxed);
}
```

The `unsafe` Boundary vs “Everything Is Potentially Unsafe”

Rust’s central engineering advantage for large systems is the explicit unsafe boundary.

- In C++, undefined behavior can originate anywhere: out-of-bounds, use-after-free, data races, strict-aliasing violations, etc.
- In Rust, undefined behavior is primarily possible inside `unsafe` blocks or via FFI; safe code is designed to prevent those classes of UB by construction.

Practical mapping:

- C++ “trust me” code ≈ Rust `unsafe` code.
- Rust demands you write down invariants; C++ often leaves them as documentation/comments/reviews.

FFI and ABI Interop

Both languages can interoperate with C ABIs.

- C++: expose `extern "C"` functions for stable C ABI; avoid exceptions across boundaries.
- Rust: use `extern "C"` and `#[no_mangle]` for exported symbols; FFI calls are `unsafe`.

```
/* C++: exporting a C ABI function */
extern "C" int kv_put(const char* key, const unsigned char* val, unsigned long n);
```

```
/* Rust: exporting a C ABI function */
#[no_mangle]
pub extern "C" fn kv_put(_key: *const u8, _klen: usize, _val: *const u8, _vlen: usize) ->
↪  i32 {
    /* SAFETY: caller must provide valid pointers and lengths */
    0
}
```

Compile-Time vs Tooling: Two Reliability Strategies

C++ and Rust both use tooling, but they lean differently.

- C++ reliability often depends on:
  - sanitizers (ASan/UBSan/TSan),
  - fuzzing,
  - static analysis,
  - defensive coding conventions,
  - heavy CI and test matrices.
- Rust reliability often combines:
  - strong compile-time invariants (ownership, borrowing, thread-safety),
  - `unsafe` audits and invariants documentation,
  - fuzzing and property tests (especially for parsers/decoders),
  - Miri-style interpretation for `unsafe` assumptions (where applicable),
  - the same CI discipline used in C++.

Performance Model: What Changes and What Does Not

For systems engineers, the performance question is practical:

- Both languages can be zero-cost at runtime for many abstractions.
- Both can generate vectorized code and do aggressive inlining.
- The main differences usually come from:
  - allocation strategy and patterns,
  - data layout choices,
  - aliasing assumptions (Rust often enables stronger aliasing-based optimizations in safe code),
  - error-handling style (exceptions vs `Result` pipelines),
  - concurrency architecture and synchronization design.

Mapping Cheat Sheet (High-Signal)

- **RAII**: C++ destructor ↔ Rust `Drop`.
- **Unique ownership**: `std::unique_ptr<T>` ↔ `Box<T>`.
- **Shared ownership**: `std::shared_ptr<T>` ↔ `Arc<T>` (thread-safe) / `Rc<T>` (single-thread).
- **Borrowed view**: `T& / const T&` ↔ `&mut T / &T` (with aliasing rules).
- **Optional**: `std::optional<T>` ↔ `Option<T>`.
- **Result**: `std::expected<T,E>` ↔ `Result<T,E>`.

- **Span/view**: `std::span<T>` ↔ `&[T]` / `&mut [T]`.
- **Strings**: `std::string` ↔ `String`; `std::string_view` ↔ `&str`.
- **Threads**: `std::thread` ↔ `std::thread`.
- **Mutex**: `std::mutex` ↔ `std::sync::Mutex`.
- **Atomics**: `std::atomic` ↔ `std::sync::atomic`.
- **Unsafe**: "implicit everywhere" ↔ explicit `unsafe` boundary.

Practical Guidance for Systems Engineers Switching Between Them

- Think in terms of **invariants**: Rust makes you encode them; C++ makes you enforce them socially and via tools.
- For Rust, learn to design APIs that **make invalid states unrepresentable**; for C++, learn to enforce invariants via RAII, ownership types, and careful layering.
- Treat Rust `unsafe` like a C++ "kernel": small, audited, tested, fuzzed, and surrounded by safe interfaces.
- In both languages, performance wins are still mostly about data layout, cache behavior, reducing contention, and eliminating unnecessary allocations.

Appendix B: Common Rust Design Patterns

This appendix summarizes high-signal Rust design patterns commonly used in production systems. The emphasis is on patterns that improve safety, testability, and performance by aligning with Rust's ownership model, trait system, and error handling. Each pattern is presented as a reusable mental template and supported by practical code.

Pattern 1: Newtype for Strong Typing and Invariants

Use a **newtype** (a tuple struct wrapping an underlying type) to:

- prevent accidental mixing of logically distinct values with the same representation,
- enforce invariants at construction time,
- provide domain-specific APIs while preserving zero-cost representation.

```rust
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct UserId(u64);

impl UserId {
    pub fn new(raw: u64) -> Result<Self, &'static str> {
        if raw == 0 { return Err(”id must be non-zero”); }
        Ok(UserId(raw))
    }
    pub fn get(self) -> u64 { self.0 }
}

fn demo() {
    let id = UserId::new(42).unwrap();
    let raw = id.get();
    let _ = raw;
}
```

Professional rule: prefer newtypes for identifiers, sizes, offsets, and validated strings.

Pattern 2: Builder for Complex Construction

Use a **builder** when:

- a type has many optional configuration fields,

- you want readable call sites,
- you need validation before construction.

```
#[derive(Debug, Clone)]
pub struct ServerCfg {
    pub bind: String,
    pub max_conns: usize,
    pub max_line: usize,
}

pub struct ServerCfgBuilder {
    bind: Option<String>,
    max_conns: Option<usize>,
    max_line: Option<usize>,
}

impl ServerCfgBuilder {
    pub fn new() -> Self { Self { bind: None, max_conns: None, max_line: None } }
    pub fn bind(mut self, s: impl Into<String>) -> Self { self.bind = Some(s.into()); self }
    pub fn max_conns(mut self, n: usize) -> Self { self.max_conns = Some(n); self }
    pub fn max_line(mut self, n: usize) -> Self { self.max_line = Some(n); self }

    pub fn build(self) -> Result<ServerCfg, &'static str> {
        let bind = self.bind.ok_or(”bind is required”)?;
        let max_conns = self.max_conns.unwrap_or(1024);
        let max_line = self.max_line.unwrap_or(8 * 1024);
        if max_conns == 0 { return Err(”max_conns must be > 0”); }
        if max_line == 0 { return Err(”max_line must be > 0”); }
        Ok(ServerCfg { bind, max_conns, max_line })
    }
}
```

Pattern 3: RAII Guard for Scoped Effects

Use RAII guards to ensure cleanup happens even on early returns. In Rust, this is natural via `Drop`.

```
pub struct ScopeLog<'a> {
    name: &'a str,
}

impl<'a> ScopeLog<'a> {
    pub fn new(name: &'a str) -> Self {
        println!("enter {}", name);
        Self { name }
    }
}

impl Drop for ScopeLog<'_> {
    fn drop(&mut self) {
        println!("exit {}", self.name);
    }
}

fn work() {
    let _g = ScopeLog::new("work");
    /* early returns still trigger Drop */
}
```

Pattern 4: Error Typing with Stable Categories

Use structured errors with stable kinds for matching, rather than ad-hoc strings. This enables reliable handling and prevents breaking downstream code accidentally.

```
#[derive(Debug, Clone, PartialEq, Eq)]
```

```
pub enum ErrorKind {
    InvalidInput,
    NotFound,
    Io,
}

#[derive(Debug)]
pub struct Error {
    kind: ErrorKind,
    msg: &'static str,
}

impl Error {
    pub fn kind(&self) -> ErrorKind { self.kind.clone() }
    pub fn new(kind: ErrorKind, msg: &'static str) -> Self { Self { kind, msg } }
}
```

Pattern 5: Result and ? for Linear Control Flow

Prefer `Result` with ? to keep the main logic linear and avoid deeply nested error handling.

```
fn parse_port(s: &str) -> Result<u16, &'static str> {
    let n: u32 = s.parse().map_err(|_| ”not a number”)?;
    if n > 65535 { return Err(”out of range”); }
    Ok(n as u16)
}

fn demo() -> Result<(), &'static str> {
    let p = parse_port(”8080”)?;
    let _ = p;
    Ok(())
}
```

Pattern 6: Trait-Based Abstraction Instead of Inheritance

Rust uses traits for polymorphism. Design with:

- generics for zero-cost static dispatch,
- trait objects (`dyn Trait`) when you need runtime dispatch or heterogeneous collections.

```rust
trait Clock {
    fn now_ms(&self) -> u64;
}

struct SysClock;

impl Clock for SysClock {
    fn now_ms(&self) -> u64 { 0 }
}

fn measure<C: Clock>(c: &C) -> u64 {
    let t = c.now_ms();
    t
}
```

Pattern 7: Strategy via Generic Parameter (Zero-Cost)

Pass behavior as a type parameter to let the compiler inline and optimize.

```rust
use std::marker::PhantomData;

trait HashStrategy {
    fn hash(bytes: &[u8]) -> u64;
}

struct Fnv;
```

```rust
impl HashStrategy for Fnv {
    fn hash(bytes: &[u8]) -> u64 {
        let mut h = 1469598103934665603u64;
        for &b in bytes {
            h ^= b as u64;
            h = h.wrapping_mul(1099511628211);
        }
        h
    }
}

struct Table<S: HashStrategy> {
    _s: PhantomData<S>,
}

impl<S: HashStrategy> Table<S> {
    fn key_hash(&self, k: &[u8]) -> u64 { S::hash(k) }
}
```

Pattern 8: State Machine with enum

A common Rust pattern is to model states explicitly with enum. This prevents invalid states and makes transitions explicit.

```rust
#[derive(Debug, Clone)]
enum ConnState {
    Handshake,
    Ready,
    Closing,
}

fn step(s: ConnState, input: &str) -> ConnState {
    match s {
        ConnState::Handshake => {
            if input == ”HELLO” { ConnState::Ready } else { ConnState::Closing }
        }
        ConnState::Ready => {
            if input == ”BYE” { ConnState::Closing } else { ConnState::Ready }
        }
        ConnState::Closing => ConnState::Closing,
    }
}
```

Pattern 9: Interior Mutability for Shared Access Patterns

When you need mutation behind a shared reference, Rust provides interior mutability types. Use them deliberately:

- `Cell/RefCell` for single-threaded scenarios,
- `Mutex/RwLock` for multi-threaded scenarios.

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

struct Cache {
    hits: RefCell<u64>,
}

impl Cache {
    fn new() -> Self { Self { hits: RefCell::new(0) } }
    fn hit(&self) {
        *self.hits.borrow_mut() += 1;
    }
}
```

Pattern 10: Zero-Copy Views with Slices

Prefer borrowing (`&[u8]`, `&str`) rather than allocating. This is fundamental for high-performance parsing and networking.

```rust
fn parse_header(line: &str) -> Option<(&str, &str)> {
    let (k, v) = line.split_once(':')?;
    Some((k.trim(), v.trim()))
}
```

Pattern 11: `Cow` for Borrow-or-Own APIs

Use `Cow` when you want to accept either borrowed data or owned data without forcing allocation.

```rust
use std::borrow::Cow;

fn normalize<'a>(s: &'a str) -> Cow<'a, str> {
    if s.chars().all(|c| c.is_ascii_lowercase()) {
        Cow::Borrowed(s)
    } else {
        Cow::Owned(s.to_ascii_lowercase())
    }
}
```

Pattern 12: Type-State for Compile-Time Protocol Enforcement

Model “phases” with different types so invalid calls do not compile.

```rust
use std::marker::PhantomData;

struct Disconnected;
struct Connected;

struct Client<S> {
    _s: PhantomData<S>,
}

impl Client<Disconnected> {
    fn new() -> Self { Self { _s: PhantomData } }
    fn connect(self) -> Client<Connected> { Client { _s: PhantomData } }
}

impl Client<Connected> {
    fn send(&self, _msg: &str) { /* ... */ }
}
```

Pattern 13: Small-Scope `unsafe` with Documented Invariants

When `unsafe` is required for performance or low-level integration:

- isolate it into small functions,
- document invariants at the boundary,
- expose a safe wrapper that enforces those invariants.

```
use std::ptr::NonNull;

/* Safe wrapper: ensures non-null and correct length discipline */
pub struct Buf {
    p: NonNull<u8>,
    len: usize,
}

impl Buf {
    pub fn from_slice(s: &mut [u8]) -> Self {
        Self {
            p: NonNull::new(s.as_mut_ptr()).unwrap(),
            len: s.len(),
        }
    }

    pub fn first(&self) -> u8 {
        /* SAFETY: p is non-null and len>0 checked */
        assert!(self.len > 0);
        unsafe { *self.p.as_ptr() }
    }
}
```

Pattern 14: Dependency Injection via Traits for Testability

Use traits to inject dependencies (clock, filesystem, network) and enable deterministic testing.

```
trait Fs {
    fn read(&self, path: &str) -> Result<Vec<u8>, ()>;
}

struct RealFs;
impl Fs for RealFs {
    fn read(&self, _path: &str) -> Result<Vec<u8>, ()> { Ok(Vec::new()) }
}

struct App<F: Fs> { fs: F }
impl<F: Fs> App<F> {
    fn load(&self, p: &str) -> Result<usize, ()> {
        Ok(self.fs.read(p)?.len())
    }
}
```

Pattern 15: Bounded Queues for Backpressure

Bounded queues prevent overload from turning into memory exhaustion. The pattern is:

- choose a fixed capacity,
- reject work when full,
- expose overload as a normal error path.

```rust
use std::collections::VecDeque;

pub struct BoundedQ<T> {
    q: VecDeque<T>,
    cap: usize,
}

impl<T> BoundedQ<T> {
    pub fn new(cap: usize) -> Self { Self { q: VecDeque::new(), cap } }
    pub fn push(&mut self, v: T) -> Result<(), ()> {
        if self.q.len() >= self.cap { return Err(()); }
        self.q.push_back(v);
        Ok(())
    }
    pub fn pop(&mut self) -> Option<T> { self.q.pop_front() }
}
```

Design Pattern Checklist (Production Use)

When choosing a pattern in production systems, verify:

- invariants are explicit and enforceable by the type system or API,
- error paths are stable and matchable (no fragile string contracts),

- allocations are intentional and bounded in hot paths,
- concurrency patterns provide backpressure and avoid unbounded queues,
- any `unsafe` is small, justified, and covered by tests.

These patterns are not "Rust-specific tricks." They are practical engineering forms that Rust makes easier to express safely, especially in performance- and correctness-critical systems.

Appendix C: Beginner Mistakes and How to Avoid Them

This appendix lists common Rust beginner mistakes that directly affect correctness, performance, and maintainability in real systems code. Each item explains the failure mode, why it happens, and a practical avoidance strategy with compact examples.

Mistake 1: Fighting the Borrow Checker Instead of Using It as a Design Tool

Symptoms:

- excessive cloning to "make the compiler happy,"
- trying to keep long-lived mutable borrows,
- forcing complicated lifetimes when a redesign is simpler.

Avoidance strategy:

- reduce borrow scope (create smaller blocks),
- split data into independent parts (struct fields, separate vectors),
- pass data by reference and return results instead of storing borrows.

```
fn bad(mut v: Vec<i32>) -> i32 {
    let x = &mut v;      /* long borrow */
    x.push(1);
    /* later want to read v again -> redesign instead of cloning */
    v.len() as i32
}

/* Better: keep mutable borrow in a small scope */
fn good(mut v: Vec<i32>) -> i32 {
    {
        let x = &mut v;
        x.push(1);
    } /* borrow ends here */
    v.len() as i32
}
```

Mistake 2: Overusing `clone()` as a Default Fix

Cloning is sometimes correct, but as a reflex it:

- adds hidden allocations,
- increases CPU work,
- hides design problems.

Avoidance strategy:

- prefer borrowing (`&T`, `&str`, `&[T]`),
- use `Cow` when you need “borrow-or-own,”
- move ownership instead of cloning where appropriate.

```
fn takes_str(s: &str) -> usize { s.len() }

fn avoid_clone() {
    let s = String::from("hello”);
    let n = takes_str(&s); /* borrow, no clone */
    let _ = n;
}
```

Mistake 3: Returning References to Temporaries

A classic beginner error: trying to return a reference to a value created inside the function.

```
/* This will not compile (and should not): */
fn bad() -> &str {
    let s = String::from("hi”);
    &s
}
```

Avoidance strategy:

- return an owned value (`String`),
- or accept an output buffer and write into it,
- or store the data in a struct owned by the caller.

```
fn good() -> String {
    String::from("hi”)
}
```

Mistake 4: Confusing `String` and `&str`

`String` owns heap-allocated UTF-8; `&str` is a borrowed view.

Avoidance strategy:

- accept `&str` in APIs unless ownership is required,
- only allocate (`String`) at boundaries (I/O, caching, long-term storage).

```
fn api(name: &str) -> usize { name.len() }

fn demo() {
    let s = String::from("rust");
    let a = api(&s);
    let b = api("literal");
    let _ = (a, b);
}
```

Mistake 5: Using `unwrap()` Everywhere in Production Paths

`unwrap()` is acceptable in:

- tests,
- prototypes,
- cases where panic is a deliberate invariant assertion.

But in production systems, uncontrolled panics become outages.
Avoidance strategy:

- use `Result` and propagate errors with `?`,
- convert errors to stable categories (error kinds),
- reserve panic for impossible states.

```
fn parse_port(s: &str) -> Result<u16, &'static str> {
    let n: u32 = s.parse().map_err(|_| "not a number")?;
    if n > 65535 { return Err("out of range"); }
    Ok(n as u16)
}
```

Mistake 6: Treating `panic!` Like Exceptions

Rust panics are not a general error-handling mechanism. They represent bugs or violated invariants, not normal control flow.
Avoidance strategy:

- model recoverable failures with `Result`,
- use panics for internal invariants only.

Mistake 7: Misunderstanding `Send` / `Sync` and Concurrency Guarantees

Beginners often assume that because code compiles, it is "safe and fast." Rust prevents data races in safe code, but concurrency still requires architecture:

- avoid unbounded queues,
- avoid lock contention hot spots,
- design backpressure.

Avoidance strategy:

- use `Arc<Mutex<T»` or `Arc<RwLock<T»` only with clear access patterns,
- prefer message passing for coarse-grained concurrency,
- measure contention and latency under load.

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

fn demo() {
    let x = Arc::new(Mutex::new(0i32));
    let mut th = Vec::new();

    for _ in 0..4 {
        let x2 = Arc::clone(&x);
        th.push(thread::spawn(move || {
            let mut g = x2.lock().unwrap();
            *g += 1;
        }));
    }
    for t in th { t.join().unwrap(); }
}
```

Mistake 8: Using `RefCell` as a Shortcut Without Understanding Runtime Borrow Checks

`RefCell` enables interior mutability but enforces borrowing rules at runtime. Violations panic. Avoidance strategy:

- use `RefCell` only when single-threaded and when design requires it,
- keep borrow scopes small,
- consider redesigning ownership instead.

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

fn safe_refcell() {
    let x = RefCell::new(0i32);
    {
        let mut a = x.borrow_mut();
        *a += 1;
    } /* borrow ends */
    let b = x.borrow();
    let _ = *b;
}
```

Mistake 9: Hidden Allocations in Hot Paths

Beginners often allocate inside loops unintentionally:

- repeated `format!` in tight loops,
- repeated `to_string()` conversions,
- building temporary vectors repeatedly.

Avoidance strategy:

- pre-allocate buffers (`with_capacity`),
- reuse `String` and `Vec` buffers,
- borrow rather than convert when possible.

```
fn better_loop(inputs: &[&str]) -> usize {
    let mut buf = String::with_capacity(256);
    let mut sum = 0usize;

    for s in inputs {
        buf.clear();
        buf.push_str(s);
        sum += buf.len();
    }
    sum
}
```

Mistake 10: Misusing Iterators and Collecting Too Early

Collecting creates allocations and loses laziness.
Avoidance strategy:

- keep chains lazy when possible,
- collect only at boundaries where you truly need a container.

```rust
fn sum_even(xs: &[i32]) -> i32 {
    xs.iter()
      .copied()
      .filter(|x| (x & 1) == 0)
      .sum()
}
```

Mistake 11: Confusing Slices and Ownership in APIs

A common anti-pattern is taking `Vec<T>` when you only need read access.
Avoidance strategy:

- accept `&[T]` for read-only,
- accept `&mut [T]` for in-place mutation,
- accept `Vec<T>` only when you need ownership.

```rust
fn api_read(xs: &[u8]) -> u8 {
    xs.get(0).copied().unwrap_or(0)
}
```

Mistake 12: Overexposing Internals Instead of Designing a Minimal Surface

Beginners often make everything pub early and later regret it.

Avoidance strategy:

- keep modules private by default,
- expose only the stable, intended API,
- treat public items as long-term contracts.

```
/* lib.rs */
mod internal;         /* private */
pub mod api;          /* public surface */
```

Mistake 13: Using `unsafe` Too Early

Beginners sometimes reach for `unsafe` for performance without proof.

Avoidance strategy:

- measure first,
- attempt safe optimizations (data layout, allocations, algorithmic changes),
- use `unsafe` only when required and isolate it,
- write and document invariants that justify it.

Mistake 14: Poor Feature Flag Hygiene

Features are additive. Beginners may design mutually-exclusive features that break in real dependency graphs.

Avoidance strategy:

- keep default features minimal,
- design additive capability flags,
- separate incompatible backends into different crates when necessary.

Mistake 15: Weak Testing Strategy

Beginners often test only the “happy path” and skip:

- error paths,
- boundary conditions,
- property-based tests for parsers/decoders,
- concurrency stress scenarios.

Avoidance strategy:

- test invariants, not just outputs,
- add negative tests and regression tests,
- use fuzzing/property tests for complex input spaces.

Practical Checklist (Keep This Near Your Editor)

- Prefer `&str` / `&[T]` in APIs; allocate only at boundaries.
- Replace `unwrap()` in production paths with `Result` + `?`.
- Keep borrow scopes small; redesign ownership instead of cloning.
- Make invariants explicit; treat `pub` as a long-term contract.

- Avoid hidden allocations in loops; reuse buffers.
- Add tests for errors, boundaries, and invariants; not only happy paths.
- Use `unsafe` only with proof, isolation, and documented invariants.

Appendix D: Rust Interview Questions with Answers

This appendix provides high-signal Rust interview questions with concise, technical answers aimed at systems engineers. The focus is on correctness, memory safety, performance trade-offs, concurrency, and practical API design. Many questions include minimal code examples suitable for whiteboard or take-home discussion.

Ownership, Borrowing, Lifetimes

Q1: What does "ownership" mean in Rust, and what is the core rule?

Ownership means each value has a single owning binding responsible for its lifetime. The core rule is:

- each value has exactly one owner at a time,
- when the owner goes out of scope, `Drop` runs and the value is freed,
- ownership can be moved, and the old binding becomes unusable.

```rust
fn demo() {
    let s = String::from("hi");
    let t = s;              /* move */
    /* let u = s; */         /* error: use of moved value */
    let _ = t;
}
```

Q2: What is the difference between `&T` and `&mut T`?

`&T` is a shared borrow: multiple shared borrows can coexist, and they allow reading. `&mut T` is an exclusive borrow: it is the only active reference to that value, enabling mutation. Rust enforces:

- either one `&mut T`, or any number of `&T`,
- but not both in an overlapping scope.

```rust
fn demo() {
    let mut x = 1;
    let a = &x;
    let b = &x;
    let _ = (*a + *b);

    let m = &mut x;
    *m += 1;
}
```

Q3: What problem do lifetimes solve?

Lifetimes express, at compile time, how long references are valid. They prevent dangling references by ensuring that borrowed data outlives the references to it. In practice, most lifetimes are inferred; explicit lifetimes are needed when:

- returning references tied to input references,
- storing references in structs,
- expressing relationships between multiple input borrows.

```rust
fn first<'a>(a: &'a str, _b: &'a str) -> &'a str {
    a
}
```

Q4: Why can't Rust return a reference to a local variable?

Because the local variable is dropped at the end of the function, any reference to it would dangle. Rust rejects it at compile time.

```
/* does not compile */
fn bad() -> &str {
    let s = String::from("x");
    &s
}
```

Q5: What is "interior mutability" and when should you use it?

Interior mutability allows mutation through a shared reference by shifting borrow rules to runtime or synchronization primitives:

- `Cell`/`RefCell`: single-threaded runtime borrow rules,
- `Mutex`/`RwLock`: multi-threaded synchronized mutation.

Use it when the external API should be shared (`&self`) but you need controlled internal mutation (caches, counters, memoization), and you can justify the trade-offs.

Traits, Generics, and Dispatch

Q6: What is the difference between static dispatch and dynamic dispatch in Rust?

Static dispatch uses generics and monomorphization: the compiler generates specialized code per concrete type, enabling inlining and optimization. Dynamic dispatch uses trait objects (`dyn Trait`): method calls go through a vtable, enabling heterogeneous collections at runtime with a dispatch cost and fewer inlining opportunities.

```
trait Shape { fn area(&self) -> f64; }

/* static dispatch */
fn total_static<T: Shape>(xs: &[T]) -> f64 {
    xs.iter().map(|s| s.area()).sum()
}

/* dynamic dispatch */
fn total_dyn(xs: &[&dyn Shape]) -> f64 {
    xs.iter().map(|s| s.area()).sum()
}
```

Q7: What is a "trait bound" and why does it matter for API design?

A trait bound constrains which types can be used with a generic function or type. It:

- expresses requirements explicitly,
- improves correctness and readability,
- enables better compiler diagnostics,
- narrows the API surface to meaningful operations.

```
use std::fmt::Display;

fn log_value<T: Display>(x: T) {
    println!("{}", x);
}
```

Q8: What are the orphan rules (coherence), and why do they exist?

Rust restricts trait implementations to avoid ambiguity and ensure global coherence:

- you can implement a trait for a type only if either the trait or the type is defined in your crate.

This prevents “conflicting impls” across dependencies that would otherwise make method resolution unpredictable.

Error Handling and Robustness

Q9: When should you return `Option<T>` vs `Result<T,E>`?

Use `Option<T>` when “absence” is normal and not an error (lookup miss). Use `Result<T,E>` when failure is meaningful and should carry error information (invalid input, I/O, protocol errors).

```
fn find(xs: &[i32], x: i32) -> Option<usize> {
    xs.iter().position(|&v| v == x)
}

fn parse_u16(s: &str) -> Result<u16, &'static str> {
    let n: u32 = s.parse().map_err(|_| "not a number")?;
    if n > 65535 { return Err("out of range"); }
    Ok(n as u16)
}
```

Q10: What does the ? operator do?

It propagates errors in `Result` (or `Option`) by returning early when a failure occurs, while unwrapping the success value. It keeps code linear and reduces boilerplate.

```
fn read_port(s: &str) -> Result<u16, &'static str> {
    let p = parse_u16(s)?; /* early return on error */
    Ok(p)
}
```

Q11: What is the difference between `panic!` and returning an error?

`panic!` indicates a bug or violated invariant (unrecoverable in most designs). Returning an error indicates an expected failure mode. Production systems typically reserve panics for “should never happen” conditions, and model recoverable failures with `Result`.

Ownership Patterns in APIs

Q12: Why do Rust APIs often accept `&str` instead of `String`?

Because `&str` allows callers to pass:

- string literals,
- slices of existing `String`,
- other borrowed string views,

without forcing allocation. Take `String` only when you need ownership (store long-term, transfer across threads, etc.).

```
fn greet(name: &str) -> String {
    format!(”hello {}”, name)
}
```

Q13: What is the difference between `Box<T>`, `Rc<T>`, and `Arc<T>`?

- `Box<T>`: single-owner heap allocation.
- `Rc<T>`: shared ownership via reference counting (single-threaded).
- `Arc<T>`: atomic reference counting (thread-safe shared ownership).

```
use std::rc::Rc;
use std::sync::Arc;

fn demo() {
    let a = Arc::new(5);
    let b = Arc::clone(&a);
    let _ = (a, b);

    let r = Rc::new(7);
    let s = Rc::clone(&r);
    let _ = (r, s);
}
```

Concurrency and Memory Model

Q14: How does Rust prevent data races in safe code?

Rust's type system enforces safe sharing rules:

- `Send`: a type can be transferred across threads,
- `Sync`: a type can be referenced from multiple threads safely.

Shared mutation requires synchronization primitives (mutexes, locks, atomics). Data races require unsynchronized shared mutable access, which safe Rust prevents by construction.

Q15: When do you choose `Mutex` vs `RwLock`?

- `Mutex`: good default; simpler; often faster under contention for short critical sections.
- `RwLock`: beneficial when reads dominate and write frequency is low; but can suffer writer starvation or heavier overhead depending on workload.

Q16: What does “`Ordering::Relaxed`” mean for atomics?

It provides atomicity without ordering constraints between threads. Use it for counters/statistics when you do not require synchronization with other memory accesses. For synchronization protocols, you typically need acquire/release or stronger orderings.

```rust
use std::sync::atomic::{AtomicU64, Ordering};

static REQ: AtomicU64 = AtomicU64::new(0);

fn tick() {
    REQ.fetch_add(1, Ordering::Relaxed);
}
```

Async and Performance Engineering

Q17: Why is blocking work inside async tasks a problem?

Blocking calls can stall runtime worker threads, increasing latency and reducing throughput. CPU-heavy or blocking I/O should be moved to dedicated threads (or `spawn_blocking`) and the async side should remain I/O-driven.

Q18: What is backpressure and how do you implement it?

Backpressure is controlling load when demand exceeds capacity. Practical implementations:

- bound concurrency (semaphore),
- bound queues (bounded channels),
- timeouts and rejection under overload,
- per-connection limits (max request size, max outstanding responses).

Unsafe Rust and FFI

Q19: What does `unsafe` mean in Rust?

`unsafe` means the compiler cannot guarantee certain safety properties, so the programmer must uphold specific invariants manually. Typical reasons:

- dereferencing raw pointers,
- calling FFI,
- implementing low-level performance primitives.

Professional practice isolates `unsafe` into small blocks and documents the invariants.

Q20: What are common sources of undefined behavior even in Rust?

Most UB in Rust occurs through:

- incorrect `unsafe` invariants (dangling pointers, aliasing violations),
- data races through unsafely shared mutation,
- FFI boundary contract violations (wrong types, wrong lifetimes, wrong ownership).

Q21: How do you design a safe wrapper over an unsafe core?

Principle:

- keep `unsafe` internal,
- expose a safe API that enforces invariants through types and checks,
- minimize the unsafe surface and test it heavily.

```rust
use std::ptr::NonNull;

pub struct NonEmptyBuf {
    p: NonNull<u8>,
    len: usize,
}

impl NonEmptyBuf {
    pub fn from_slice(s: &mut [u8]) -> Result<Self, &'static str> {
        if s.is_empty() { return Err("empty"); }
        Ok(Self { p: NonNull::new(s.as_mut_ptr()).unwrap(), len: s.len() })
    }

    pub fn first(&self) -> u8 {
        /* SAFETY: invariant len>0 */
        unsafe { *self.p.as_ptr() }
    }
}
```

Cargo, Features, and Project Architecture

Q22: What is feature unification and why can it surprise you?

Cargo features are additive and unify across the dependency graph: if any dependency enables a feature of a crate, that feature is enabled everywhere for that crate. This can:

- increase binary size,
- enable optional backends unexpectedly,
- change build characteristics.

Mitigation: keep default features minimal, make features additive and safe, and separate incompatible choices into different crates.

Q23: What is a workspace and why is it useful?

A workspace groups multiple crates into one build and dependency-resolution unit. It enables:

- shared `Cargo.lock` for consistent builds,
- shared dependency versions,
- monorepo refactoring across crates,
- centralized profiles and policies.

Code Quality and Testing

Q24: What is the recommended testing strategy for systems Rust?

A professional strategy layers tests:

- unit tests for logic and invariants,
- integration tests for public API behavior,
- property-based tests for parsers/encoders and edge-heavy logic,
- concurrency stress tests,
- benchmarks for hot paths and tail latency.

Q25: How do you avoid writing "tests that just duplicate the code"?

Test properties and invariants rather than internal steps:

- boundary conditions (empty/full, max sizes),
- monotonicity or idempotence,

- round-trip encode/decode,
- stable error categories and spans,
- concurrency invariants (no lost updates).

System-Level Thinking

Q26: What are common performance pitfalls in Rust systems code?

- hidden allocations (`format!`, repeated `to_string()`),
- unnecessary cloning,
- excessive locking or large critical sections,
- unbounded queues and missing backpressure,
- cache-unfriendly data layout.

Q27: How do you reason about "zero-cost abstractions" in Rust?

Rust can compile many abstractions away (monomorphized generics, inlining, iterators), but "zero-cost" is conditional:

- you must avoid hidden allocations,
- you must keep hot-path code simple enough to inline,
- you must measure (benchmarks, profiling),
- you must understand data layout and cache behavior.

Quick Mini-Exercises (Common Take-Home Style)

Exercise 1: Implement a bounded queue with backpressure

Requirement: fixed capacity, push fails when full, pop returns `Option<T>`.

```rust
use std::collections::VecDeque;

pub struct BoundedQ<T> {
    q: VecDeque<T>,
    cap: usize,
}

impl<T> BoundedQ<T> {
    pub fn new(cap: usize) -> Self { Self { q: VecDeque::new(), cap } }

    pub fn push(&mut self, v: T) -> Result<(), ()> {
        if self.q.len() >= self.cap { return Err(()); }
        self.q.push_back(v);
        Ok(())
    }

    pub fn pop(&mut self) -> Option<T> {
        self.q.pop_front()
    }
}
```

Exercise 2: Parse an integer safely without panics

```rust
fn parse_u16(s: &str) -> Result<u16, &'static str> {
    let n: u32 = s.parse().map_err(|_| "not a number")?;
    if n > 65535 { return Err("out of range"); }
    Ok(n as u16)
}
```

These questions intentionally test not only syntax, but engineering maturity: API design, correctness boundaries, concurrency reasoning, and performance awareness in Rust systems work.

Appendix E: Practical Cheatsheets (Ownership, Lifetimes, Traits, Async)

This appendix is a compact, high-signal reference for daily systems work in Rust. It summarizes rules, patterns, and "what to reach for first" across ownership, lifetimes, traits, and async. Use it as a practical checklist during implementation and code review.

Cheatsheet 1: Ownership (What Moves, What Borrows, What Copies)

Core rules:

- Values have a single owner; when it goes out of scope, `Drop` runs.
- Most non-trivial types **move** by default (the old binding becomes unusable).
- Only `Copy` types are implicitly copied (integers, floats, `bool`, raw pointers, small `Copy` structs).
- Borrowing creates references without transferring ownership.

Move vs Copy

```
fn move_vs_copy() {
    let a: u32 = 7;      /* Copy */
    let b = a;           /* copy */
    let _ = (a, b);
    let s = String::from("hi"); /* move */
    let t = s;
    /* let u = s; */     /* error: moved */
    let _ = t;
}
```

Borrowing

```
fn borrowing() {
    let s = String::from("hello");
    let r: &str = &s;       /* borrow as &String -> coerces to &str */
    let n = r.len();
    let _ = n;
}
```

Mutation Requires Exclusive Borrow

```
fn mutate() {
    let mut x = 1;
    let m = &mut x;
    *m += 1;
    /* cannot borrow x again mutably while m is alive */
}
```

API Rule of Thumb

Default choices for APIs:

- read-only: `&T`, `&[T]`, `&str`
- in-place edit: `&mut T`, `&mut [T]`
- take ownership: `T` (only if you must store/move it)

Cheatsheet 2: Lifetimes (When You Must Write Them)

You need explicit lifetimes when:

- returning a reference that is tied to an input reference,
- storing references inside a struct,
- expressing relationships between multiple input references.

Returning a Borrowed Reference

```rust
fn pick_first<'a>(a: &'a str, _b: &'a str) -> &'a str {
    a
}
```

Struct Holding References

```rust
struct View<'a> {
    bytes: &'a [u8],
}

impl<'a> View<'a> {
    fn first(&self) -> Option<u8> {
        self.bytes.get(0).copied()
    }
}
```

Lifetime Hygiene Rules

- Prefer owning data inside structs unless you truly need borrowed views.
- If lifetime annotations spread everywhere, reconsider the design:
    - store owned `String`/`Vec` instead of references,

  - split responsibilities (parse to owned AST, then process),
  - shorten borrow scopes with blocks.

Common “Cannot Return Reference to Local” Fix

```
/* wrong idea: returning reference to local value */
fn bad() -> &str {
    let s = String::from("x");
    &s
}

/* correct: return owned */
fn good() -> String {
    String::from("x")
}
```

Cheatsheet 3: Traits (Generic Bounds, dyn Trait, and Design Patterns)

Traits are Rust’s mechanism for:

- shared behavior (interfaces),
- constraints on generics,
- static dispatch and dynamic dispatch.

Static Dispatch (Generic) vs Dynamic Dispatch (Trait Object)

```
trait Compressor {
    fn compress(&self, input: &[u8]) -> Vec<u8>;
}

/* static dispatch: inlined, monomorphized */
```

```
fn run_static<C: Compressor>(c: &C, data: &[u8]) -> Vec<u8> {
    c.compress(data)
}

/* dynamic dispatch: runtime vtable call */
fn run_dyn(c: &dyn Compressor, data: &[u8]) -> Vec<u8> {
    c.compress(data)
}
```

When to Use Which

- Use generics when:
  - performance is critical (inlining),
  - you can keep types uniform at compile time,
  - code size growth is acceptable.
- Use `dyn Trait` when:
  - you need heterogeneous collections,
  - you want to reduce compile-time or code bloat,
  - runtime plugin-like behavior is desired.

Trait Bounds: Minimal Required Surface

```
use std::fmt::Display;

fn log<T: Display>(x: T) {
    println!("{}", x);
}
```

Associated Types vs Generics in Traits

Use associated types when the trait conceptually has a single "output type" per implementation (iterator pattern). Use generic parameters when the trait must accept many different types.

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

Common Trait Pattern: Extension Traits

Add methods to existing types without new wrappers.

```rust
trait BytesExt {
    fn is_ascii_word(&self) -> bool;
}

impl BytesExt for [u8] {
    fn is_ascii_word(&self) -> bool {
        self.iter().all(|&b| (b as char).is_ascii_alphanumeric() || b == b'_')
    }
}
```

Cheatsheet 4: Async (I/O, Backpressure, Cancellation)

Async in Rust is built on:

- `Future`: a computation that can be polled to completion,
- an executor: runs futures (e.g., a runtime),
- non-blocking I/O: readiness-based networking and file operations.

Golden Rules for Async Systems

- Never block inside async tasks (no blocking file I/O, no heavy CPU loops).
- Bound concurrency (semaphores) and bound queues (channels).
- Design backpressure explicitly: reject work when overloaded.
- Use timeouts for slow clients and slow dependencies.
- Treat cancellation as normal: tasks can be dropped at await points.

Bound Concurrency (Admission Control)

```rust
use std::sync::Arc;
use tokio::sync::Semaphore;

async fn serve(max: usize) {
    let sem = Arc::new(Semaphore::new(max));

    loop {
        let permit = match sem.clone().try_acquire_owned() {
            Ok(p) => p,
            Err(_) => {
                /* overloaded */
                continue;
            }
        };

        tokio::spawn(async move {
            let _p = permit;
            /* handle one connection */
        });
    }
}
```

Bound Work Queue (Backpressure)

```
use tokio::sync::{mpsc, oneshot};

struct Work {
    req: Vec<u8>,
    reply: oneshot::Sender<Vec<u8>>,
}

fn start_workers(cap: usize) -> mpsc::Sender<Work> {
    let (tx, mut rx) = mpsc::channel::<Work>(cap);
    tokio::spawn(async move {
        while let Some(w) = rx.recv().await {
            let _ = w.reply.send(w.req); /* echo */
        }
    });
    tx
}

async fn submit(tx: mpsc::Sender<Work>, req: Vec<u8>) -> Result<Vec<u8>, ()> {
    let (rtx, rrx) = oneshot::channel();
    let w = Work { req, reply: rtx };

    /* try_send rejects when full */
    tx.try_send(w).map_err(|_| ())?;
    rrx.await.map_err(|_| ())
}
```

Timeouts (Slow Client / Slow Dependency Control)

```
use tokio::time::{timeout, Duration};

async fn with_timeout() -> Result<(), ()> {
    let fut = async {
        /* some async operation */
        Ok::<(), ()>(())
    };

    timeout(Duration::from_millis(200), fut).await.map_err(|_| ())?
}
```

spawn_blocking (CPU or Blocking I/O Isolation)

```
async fn cpu_heavy(x: u64) -> u64 {
    tokio::task::spawn_blocking(move || {
        let mut a = 0u64;
        for i in 0..10_000_0 { a = a.wrapping_add(i ^ x); }
        a
    }).await.unwrap_or(0)
}
```

Cancellation Awareness

Cancellation in async often occurs when:

- a task is dropped (client disconnect),
- a timeout fires and you stop awaiting,
- you select on a shutdown signal.

Write code so partial work is safe to abandon:

- keep transactions explicit,
- write idempotent operations when possible,
- ensure resource cleanup via RAII (`Drop`) and bounded scopes.

One-Page Summary: What to Reach For First

- **Need read-only input**: `&str`, `&[T]`, `&T`.
- **Need ownership**: `String`, `Vec<T>`, `Box<T>`.
- **Shared ownership**: `Rc<T>` (single-thread), `Arc<T>` (multi-thread).
- **Shared mutation**: `Mutex<T>` / `RwLock<T>` (threads), `RefCell<T>` (single-thread).
- **Recoverable failures**: `Result<T,E>` with `?`.
- **Optional value**: `Option<T>`.
- **Polymorphism**: generics for performance; `dyn Trait` for heterogeneity.
- **Async server safety**: bound connections, bound queues, timeouts, no blocking in async.

Appendix F: A 30-Day Rust Mastery Plan

This 30-day plan is a practical, systems-oriented Rust progression. It assumes you can already program (especially in C/C++ or similar) and want to become productive in Rust for high-performance, safe systems work. Each day has:

- a focused concept target,
- a concrete deliverable (code you must produce),
- a validation step (tests, benchmarks, or review checklist).

Professional rule: do not "read only." Write code every day. Each deliverable must compile, be formatted, and have at least minimal tests.

Daily Operating Rules (Non-negotiable)

- Create one workspace repository for the month; each day is a crate or a module.
- Run unit tests every day; add at least 2 tests per deliverable.
- Track performance for hot paths: measure at least once per week.
- Keep a “mistakes log”: every compiler error you repeat twice becomes a note.
- Keep APIs minimal and safe by default; isolate `unsafe` behind safe wrappers.

Week 1 (Days 1–7): Ownership Foundations and Core Data Types

Day 1: Toolchain Discipline and Project Skeleton

Deliverable:

- a Cargo workspace,
- one binary crate app and one library crate `libcore`,
- `clippy` clean for both crates.

```
/* workspace layout concept:
workspace/
  Cargo.toml
  crates/
    app/
    libcore/
*/
```

Validation:

- `cargo test` passes,

- `cargo fmt` produces no changes,
- no warnings in `cargo clippy` for the day's code.

Day 2: Ownership, Moves, and `Copy`

Deliverable:

- implement 10 small functions demonstrating move vs copy,
- write tests that prove compile-time behavior by shaping APIs (ownership vs borrowing).

```
fn takes_owned(s: String) -> usize { s.len() }
fn takes_borrowed(s: &str) -> usize { s.len() }

#[test]
fn demo_move_vs_borrow() {
    let s = String::from("hello");
    assert_eq!(takes_borrowed(&s), 5);
    assert_eq!(takes_owned(s), 5);
}
```

Day 3: Borrowing, Mutability, and Borrow Scopes

Deliverable:

- implement a small `Vec`-based collection with APIs using `&[T]` and `&mut [T]`,
- remove all unnecessary clones.

```
fn sum(xs: &[i32]) -> i32 { xs.iter().sum() }
fn inc_all(xs: &mut [i32]) { for x in xs { *x += 1; } }
```

Day 4: Strings (`String` vs `&str`) and Zero-Copy Parsing

Deliverable:

- write a small parser for `"key:value"` lines returning borrowed slices,
- avoid allocations in hot parsing paths.

```
fn parse_kv(line: &str) -> Option<(&str, &str)> {
    let (k, v) = line.split_once(':')?;
    Some((k.trim(), v.trim()))
}
```

Day 5: Enums, Pattern Matching, and Error Modeling

Deliverable:

- implement a small command parser with an `enum Command`,
- errors as `Result<Command, ErrorKind>`.

```
#[derive(Debug, Clone, PartialEq, Eq)]
enum Cmd { Ping, Echo(String) }

fn parse_cmd(s: &str) -> Result<Cmd, &'static str> {
    if s == "PING" { Ok(Cmd::Ping) }
    else if let Some(rest) = s.strip_prefix("ECHO ") { Ok(Cmd::Echo(rest.to_string())) }
    else { Err("unknown") }
}
```

Day 6: Collections and Iterators (No Premature `collect`)

Deliverable:

- write 10 iterator pipelines and show at least 3 versions that avoid collecting,
- add tests for invariants, not just outputs.

Day 7: Review Day + Mini Project

Deliverable:

- build a small CLI that reads lines and outputs parsed commands,
- measure allocation hotspots and remove the top 2 unnecessary allocations.

Week 2 (Days 8–14): Traits, Modules, Testing, and Robust APIs

Day 8: Traits and Generic APIs

Deliverable:

- define a trait-based interface for a component (e.g., `Clock`, `Logger`, `Hasher`),
- implement 2 concrete implementations and a generic consumer.

```rust
trait Clock { fn now_ms(&self) -> u64; }

struct FakeClock { t: u64 }
impl Clock for FakeClock { fn now_ms(&self) -> u64 { self.t } }

fn stamp<C: Clock>(c: &C) -> u64 { c.now_ms() }
```

Day 9: `dyn Trait` and Trade-offs

Deliverable:

- rewrite Day 8 consumer to accept `&dyn Trait`,
- record a short note: when static dispatch is better vs dynamic dispatch.

Day 10: Modules, Visibility, and API Surface Control

Deliverable:

- reorganize code into modules,
- make internals private by default,
- expose a minimal public API.

```
/* lib.rs */
mod internal;
pub mod api;
```

Day 11: Error Handling Done Right

Deliverable:

- define `ErrorKind` + `Error` struct,
- refactor 3 modules to stop using `unwrap` in non-test paths.

Day 12: Unit vs Integration Tests

Deliverable:

- write unit tests for core logic,
- add at least one integration test that treats your crate like a black box.

Day 13: Property Thinking (Invariant Tests)

Deliverable:

- write tests that check invariants across multiple cases,

- add at least one randomized loop test (small, deterministic via a seeded RNG).

```
fn lcg(mut x: u64) -> impl Iterator<Item = u64> {
    std::iter::from_fn(move || {
        x = x.wrapping_mul(6364136223846793005).wrapping_add(1);
        Some(x)
    })
}

#[test]
fn invariant_non_negative_len() {
    let mut n = 0usize;
    for v in lcg(1).take(1000) {
        if (v & 1) == 0 { n += 1; } else { n = n.saturating_sub(1); }
        assert!(n <= 1000);
    }
}
```

Day 14: Week-2 Mini Project

Deliverable:

- build a small library with a stable API and error types,
- write a README that includes a minimal example and one doctest-style snippet (optional).

Week 3 (Days 15–21): Concurrency, Atomics, and Async Fundamentals

Day 15: Threads, Send/Sync, and Ownership Across Threads

Deliverable:

- implement a small parallel aggregator using threads,

- prove correctness with tests.

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

fn parallel_inc(k: usize) -> i32 {
    let x = Arc::new(Mutex::new(0i32));
    let mut th = Vec::new();

    for _ in 0..k {
        let x2 = Arc::clone(&x);
        th.push(thread::spawn(move || {
            let mut g = x2.lock().unwrap();
            *g += 1;
        }));
    }
    for t in th { t.join().unwrap(); }
    *x.lock().unwrap()
}
```

Day 16: Mutex vs RwLock Decision Practice

Deliverable:

- implement a read-heavy cache with `RwLock`,
- measure contention under a synthetic workload (even simple timing).

Day 17: Atomics for Counters and Metrics

Deliverable:

- implement a metrics module with atomic counters,
- demonstrate relaxed ordering usage for statistics.

```rust
use std::sync::atomic::{AtomicU64, Ordering};

static REQ: AtomicU64 = AtomicU64::new(0);

fn tick() { REQ.fetch_add(1, Ordering::Relaxed); }
fn read() -> u64 { REQ.load(Ordering::Relaxed) }
```

Day 18: Async Basics and “Do Not Block” Rule

Deliverable:

- write an async echo server skeleton,
- include a note identifying what would be considered blocking.

Day 19: Backpressure and Bounded Queues

Deliverable:

- implement a bounded queue with rejection under overload,
- integrate it into the async server skeleton conceptually.

Day 20: Timeouts and Cancellation Awareness

Deliverable:

- add timeouts around one operation,
- ensure partial work is safe to abandon (no half-written invariants).

Day 21: Week-3 Mini Project

Deliverable:

- build a small concurrent service with metrics counters and bounded queue,
- write a short postmortem note: where contention could happen and how you mitigated it.

Week 4 (Days 22–30): Systems Quality: Performance, Unsafe Discipline, Architecture

Day 22: Profiling Mindset and Allocation Control

Deliverable:

- identify one hot path and remove at least one allocation,
- add a micro-benchmark loop timing comparison.

Day 23: Data Layout and Cache-Friendly Design

Deliverable:

- refactor one structure into a more cache-friendly representation,
- add a benchmark comparing before/after on a synthetic workload.

Day 24: Unsafe Rust Discipline

Deliverable:

- write a safe wrapper over a tiny `unsafe` block,
- document invariants in comments,

- add tests that exercise boundaries.

```
use std::ptr::NonNull;

pub struct NonEmpty {
    p: NonNull<u8>,
    len: usize,
}

impl NonEmpty {
    pub fn from_slice(s: &mut [u8]) -> Result<Self, &'static str> {
        if s.is_empty() { return Err("empty"); }
        Ok(Self { p: NonNull::new(s.as_mut_ptr()).unwrap(), len: s.len() })
    }

    pub fn first(&self) -> u8 {
        /* SAFETY: invariant len>0 */
        unsafe { *self.p.as_ptr() }
    }
}
```

Day 25: FFI Thinking and ABI Boundaries

Deliverable:

- write a minimal `extern "C"` boundary (conceptual),
- document ownership and lifetime expectations for pointers.

Day 26: Library Design and SemVer Discipline

Deliverable:

- define your crate's public API,

- write an API stability checklist (what is a breaking change),
- keep public types minimal and intentional.

Day 27: Documentation as Code (rustdoc Mindset)

Deliverable:

- write module-level docs for one crate,
- include at least one example snippet per public function (short).

Day 28: Testing Strategy Upgrade

Deliverable:

- add integration tests for public behavior,
- add invariant-style tests for boundary conditions,
- write a regression test for one bug you intentionally introduced and fixed.

Day 29: Benchmarking and Performance Reporting

Deliverable:

- benchmark 2 variants of a hot path,
- write a short report: what changed, why it helped, and what the trade-offs are.

Day 30: Capstone Synthesis Day

Deliverable:

- pick one of your mini-projects and harden it:

  - bounded resources,
  - stable errors,
  - minimal public API,
  - tests and a benchmark,
  - short README.
- write a one-page “production readiness checklist” for your project.

Final Output Checklist (What “Mastery” Means After 30 Days)

By the end of day 30, you should be able to:

- design APIs that borrow by default and allocate intentionally,
- reason about ownership and lifetimes without “clone-driven development,”
- implement trait-based abstractions with clear dispatch decisions,
- build concurrent code with bounded queues and backpressure,
- use atomics appropriately for metrics and low-contention counters,
- write async code that avoids blocking and controls overload,
- isolate `unsafe` behind small, documented, tested safe wrappers,
- measure performance changes and explain them with evidence.

References

This chapter lists the most authoritative references for every topic covered in this book, organized by **source type** (manuals, books, specifications, research, and ecosystem guidance). The goal is not to overwhelm you with a long bibliography, but to give you a **small set of primary sources** you can rely on for correctness, plus a disciplined set of secondary sources for deeper study.

How to read these references (the professional method)

- **Primary sources (highest authority):** official Rust documentation (`doc.rust-lang.org`), the Rust Edition Guide, the Rust Reference, standard library docs, and official toolchain books (Cargo, rustc, rustdoc).

- **Normative vs descriptive:** the Rust Reference describes language rules; the compiler and standard library docs describe implemented behavior and stable guarantees; unsafe guidelines contain evolving discussion and should be treated as *constraints and open questions*, not a formal spec.

- **When sources conflict:** prefer the latest official docs for stable Rust; for edge cases, validate with `rustc` behavior, `rustc -explain`, and minimal test programs.

- **Freeze your toolchain for reproducibility:** most performance and correctness discussions are easiest when you fix the Rust toolchain version per project.

Language specifications and official manuals (Rust semantics)

These sources define the language model and are the foundation for ownership, borrowing, lifetimes, traits, generics, and unsafe boundaries.

- **The Rust Reference** (primary language reference; grammar, items, types, traits, macros, attributes, UB-related rules that are specified).
- **The Rust Edition Guide (Rust 2024)** (edition changes, migration guidance, and how Rust evolves while maintaining stability).
- **The Rust Programming Language ("The Book")** (official learning book; explains ownership, borrowing, lifetimes, traits, concurrency primitives, and unsafe at a practical level).
- **Rust By Example** (runnable examples illustrating core language concepts and standard library usage).

Practical verification example (what to do when you are unsure)

```
# Get an authoritative explanation of a compiler error code:
rustc --explain E0382

# Keep local offline docs for your installed toolchain:
rustup doc --std
```

Standard library and core APIs (what Rust guarantees in practice)

These sources are essential for `Arc`, `Mutex`, `RwLock`, channels, atomics, collections, allocation behavior (conceptual), and performance-relevant APIs.

- **Standard Library API Documentation** (modules: `std::sync`, `std::sync::mpsc`, `std::sync::atomic`, `std::thread`, collections, IO, networking, time).

- **Core & Alloc documentation** (core traits, allocation primitives, pointer APIs, slice/string behavior).

- **Rust API Guidelines** (official library-team guidance for designing consistent, safe, and maintainable APIs).

Example: reading the true contract of a type

When you use `Arc<T>` and locks, the contract is in the docs:

- what is `Send`/`Sync`,

- which methods block,

- poisoning behavior,

- fairness and performance notes (when documented),

- panic behavior and edge cases.

Toolchain manuals (builds, targets, diagnostics, documentation)

These are the authoritative references for `rustup`, toolchains/targets, `cargo` workflows, profiles, lock files, build performance, and documentation generation.

- **The Cargo Book** (dependency resolution, package layout, workspaces, `Cargo.toml` and `Cargo.lock`, profiles, features, publishing workflow, build performance guidance).

- **The rustc Book** (compiler options, target model, platform support tiers, and how Cargo drives `rustc`).
- **The rustdoc Book** (documentation conventions, doc tests, attributes, search behavior, and documentation as a testable API contract).
- **Compiler Error Index / `rustc –explain`** (structured training material embedded into the toolchain).

Example: documenting and testing your API like a professional

```
# Build documentation for your crate and dependencies:
cargo doc

# Run documentation examples as tests (doc tests):
cargo test --doc
```

Formatting, linting, and static correctness culture

These sources support the book's maintainability goals: consistent formatting, automated linting, and disciplined CI.

- **rustfmt documentation** (formatting rules, configuration, project-wide consistency).
- **Clippy documentation** (lint categories, common correctness and performance lints, idiom enforcement).
- **The Rust Programming Language: Useful Development Tools** (official guidance on integrating these tools into daily workflows).

Example: enforce quality gates

```
cargo fmt
cargo clippy -- -D warnings
cargo test
```

Unsafe Rust: official guidance and soundness boundaries

Unsafe Rust is where systems-level power lives, but it must be treated as a contract discipline. These sources are the best entry points for unsafe correctness.

- **The Rustonomicon** (advanced/unsafe Rust concepts: aliasing pitfalls, layout, lifetimes, variance, unsafe patterns, and how to build safe abstractions).
- **The Rust Programming Language: Unsafe Rust chapter** (official introduction and best-practice direction: isolate unsafe, wrap behind safe APIs, document invariants).
- **Unsafe Code Guidelines (UCG) project** (discussion forum and evolving understanding; useful for edge cases and open questions; treat as guidance and exploration).
- **Unsafe Code Guidelines Reference (historical)** (useful glossary and terminology; parts may be non-normative).

Example: how to document an unsafe contract (template)

```
/// # Safety
/// The caller must ensure:
/// 1) `ptr` is valid for reads of `len` bytes.
/// 2) `ptr` is properly aligned for `T` (if applicable).
/// 3) The memory region does not alias any mutable reference for the duration.
/// 4) The data outlives any returned references derived from it.
unsafe fn read_bytes(ptr: *const u8, len: usize) -> u8 {
    *ptr.add(len - 1)
}
```

Concurrency and synchronization foundations (primitives and patterns)

These sources support the chapters on channels, shared state, and practical concurrency design.

- **Standard library docs:** `std::thread`, `std::sync` (Mutex/RwLock/Condvar), `std::sync::mpsc` (channels), `std::sync::atomic` (atomics).
- **The Rust Programming Language: Fearless Concurrency** (official conceptual and practical foundation).
- **Rust API Guidelines** (design patterns for safe public APIs that expose concurrency).

Atomics and memory ordering (correctness across CPU architectures)

Atomics are where many engineers become overconfident. For these chapters, rely on the most authoritative, most stable sources.

- **Rust standard library documentation for atomics** (`std::sync::atomic` and `Ordering` definitions).
- **The Rustonomicon: advanced memory/aliasing pitfalls** (where relevant to unsafe and low-level correctness).
- **Rust + LLVM model awareness** (through rustc and official docs when discussing compiler reordering and optimization boundaries).

Example: the three-tier ordering learning ladder

```rust
use std::sync::atomic::{AtomicUsize, Ordering};

static X: AtomicUsize = AtomicUsize::new(0);

fn relaxed_counter() {
    X.fetch_add(1, Ordering::Relaxed);
}

fn clarity_first() {
    X.fetch_add(1, Ordering::SeqCst);
}
```

Performance engineering and optimization (Rust-specific and systems-wide)

Use these sources to validate claims about “zero-cost abstractions”, monomorphization, inlining, and toolchain-driven performance work.

- **The Rust Programming Language** (performance-related idioms: iterators, ownership for eliminating copies, and when cloning is appropriate).

- **Cargo Book: profiles and build performance** (debug vs release, incremental compilation concepts, build-time tradeoffs).

- **rustc Book: compiler knobs** (when you need deeper control or diagnosis).

- **Standard library docs** (costly vs cheap operations, allocation-related APIs, and iterator contracts).

Example: build profiles as a correctness & performance tool

```toml
[profile.dev]
opt-level = 0
debug = true

[profile.release]
opt-level = 3
lto = true
codegen-units = 1
panic = "abort"
```

Large-scale design, crate architecture, and maintainability

These references support the book's focus on long-lived systems: clean module boundaries, visibility discipline, error design, and API evolution.

- **The Rust Programming Language: modules, packages, and crates** (official module model and visibility rules).
- **Cargo Book: workspaces, features, publishing** (real-world structuring of multi-crate systems).
- **Rust API Guidelines** (API ergonomics, naming, error types, builders, trait object safety guidance where applicable).
- **rustdoc Book** (documentation as part of correctness and maintainability; doc tests as enforcement).

FFI, ABI, and systems integration (C/C++ interoperability)

For systems programming that overlaps with existing ecosystems, rely on official Rust docs and rigorous discipline.

- **The Rust Reference:** ABI-related and representation attributes (`repr(C)`, layout, calling conventions where documented).
- **The Rustonomicon:** unsafe boundaries, layout assumptions, pointer aliasing pitfalls, and invariants.
- **Standard library docs:** raw pointers, `std::ffi`, and related primitives.

Community process and evolution (stable truth vs changing discussions)

Rust evolves through a disciplined process. The safest way to track language evolution is to rely on official change channels.

- **Rust Release Notes** (language/library/tooling changes per release).
- **Rust Edition Guide** (edition-level changes and migrations).
- **RFC process and official team docs (when needed)** (design intent; not always the final implemented behavior).

A compact “minimum shelf” (if you want only the essentials)

If you keep only one curated set of sources open while reading this book, make it these:

- The Rust Reference
- The Rust Edition Guide (Rust 2024)
- The Rust Programming Language (The Book)
- Standard Library Documentation
- The Cargo Book
- The rustc Book
- The rustdoc Book
- The Rustonomicon
- Rust API Guidelines

Reference-driven workflow (recommended for this book)

```
# 1) Use the compiler as a structured teacher:
cargo check
rustc --explain <ERROR_CODE>

# 2) Enforce style and correctness:
cargo fmt
cargo clippy -- -D warnings

# 3) Verify behavior:
cargo test
cargo test --doc

# 4) Validate your mental model with docs:
rustup doc --std
```

This chapter intentionally avoids external links and citation commands. Every item listed above exists as an official, stable reference you can access via the Rust documentation ecosystem and toolchain.