

Rust Patterns

- [Memory & Ownership Patterns](#)
 - [Pattern 1: Zero-Copy with Clone-on-Write \(Cow\)](#)
 - [Pattern 2: Interior Mutability with Cell and RefCell](#)
 - [Pattern 3: Thread-Safe Interior Mutability \(Mutex & RwLock\)](#)
 - [Pattern 4: Custom Drop Guards](#)
 - [Pattern 5: Memory Layout Optimization](#)
 - [Pattern 6: Arena Allocation](#)
 - [Pattern 7: Custom Smart Pointers](#)
- [Struct & Enum Patterns](#)
 - [Pattern 1: Struct Design Patterns](#)
 - [Pattern 2: Newtype and Wrapper Patterns](#)
 - [Pattern 3: Struct Memory and Update Patterns](#)
 - [Pattern 4: Enum Design Patterns](#)
 - [Pattern 5: Advanced Enum Techniques](#)
 - [Pattern 6: Visitor Pattern with Enums](#)
- [Trait Design Patterns](#)
 - [Pattern 1: Trait Inheritance and Bounds](#)
 - [Pattern 2: Associated Types vs Generics](#)
 - [Pattern 3: Trait Objects and Dynamic Dispatch](#)
 - [Pattern 4: Extension Traits](#)
 - [Pattern 5: Sealed Traits](#)
- [Generics & Polymorphism](#)
 - [Pattern 1: Type-Safe Generic Functions](#)
 - [Pattern 2: Generic Structs and Enums](#)
 - [Pattern 3: Trait Bounds and Constraints](#)
 - [Pattern 4: Associated Types vs Generic Parameters](#)
 - [Pattern 5: Blanket Implementations](#)
 - [Pattern 6: Phantom Types and Type-Level State](#)
 - [Pattern 7: Higher-Ranked Trait Bounds \(HRTBs\)](#)
 - [Pattern 8: Const Generics](#)
- [Builder & API Design](#)
 - [Pattern 1: Builder Pattern Variations](#)
 - [Pattern 2: Typestate Pattern](#)
 - [Pattern 3: #\[must_use\] for Critical Return Values](#)
- [Lifetime Patterns](#)
 - [Pattern 1: Named Lifetimes and Elision](#)

- [Pattern 2: Lifetime Bounds](#)
- [Pattern 3: Higher-Ranked Trait Bounds \(for Lifetimes\)](#)
- [Pattern 4: Self-Referential Structs and Pin](#)
- [Pattern 5: Variance and Subtyping](#)
- [Pattern 6: Advanced Lifetime Patterns](#)
- [Pattern Matching & Destructuring](#)
 - [Pattern 1: Advanced Match Patterns](#)
 - [Pattern 2: Exhaustiveness and Match Ergonomics](#)
 - [Pattern 4: State Machines with Type-State Pattern](#)
- [Iterator Patterns & Combinators](#)
 - [Pattern 1: Custom Iterators and IntoIterator](#)
 - [Pattern 2: Zero-Allocation Iteration](#)
 - [Pattern 3: Advanced Iterator Composition](#)
 - [Pattern 4: Streaming Algorithms](#)
 - [Pattern 4: Parallel Iteration with Rayon](#)
- [Error Handling Architecture](#)
 - [Pattern 1: Custom Error Enums for Libraries](#)
 - [Pattern 2: anyhow for Application-Level Errors](#)
 - [Pattern 2: Error Propagation Strategies](#)
 - [Pattern 3: Custom Error Types with Context](#)
 - [Pattern 4: Recoverable vs Unrecoverable Errors](#)
 - [Pattern 5: Error Handling in Async Contexts](#)
 - [Pattern 6: Error Handling Anti-Patterns](#)
- [Vec & Slice Manipulation](#)
 - [Pattern 1: Capacity Management and Amortization](#)
 - [Pattern 2: Slice Algorithms](#)
 - [Pattern 3: Chunking and Windowing](#)
 - [Pattern 4: Zero-Copy Slicing](#)
 - [Pattern 5: SIMD Operations](#)
 - [Pattern 6: Advanced Slice Patterns](#)
- [String Processing](#)
 - [Pattern 1: String Type Selection](#)
 - [Pattern 2: String Builder Pattern](#)
 - [Pattern 3: Zero-Copy String Operations](#)
 - [Pattern 4: Cow for Conditional Allocation](#)
 - [Pattern 5: UTF-8 Validation and Repair](#)
 - [Pattern 6: Character and Grapheme Iteration](#)
 - [Pattern 7: String Parsing State Machines](#)
 - [Pattern 8: URL Parser State Machine](#)
 - [Pattern 9: Gap Buffer Implementation](#)
 - [Pattern 10: Knuth-Morris-Pratt \(KMP\) String Search](#)
 - [Pattern 12: Boyer-Moore String Search](#)
 - [Pattern 12: String Interning](#)
- [HashMap & HashSet Patterns](#)
 - [Pattern 1: The Entry API](#)
 - [Pattern 2: Custom Hashing and Equality](#)
 - [Pattern 3: Capacity and Memory Management](#)

- [Pattern 4: Alternative Map Implementations](#)
- [Pattern 5: Concurrent HashMaps](#)
- [Key Takeaways**:](#)
- [Advanced Collections](#)
 - [Pattern 1: VecDeque and Ring Buffers](#)
 - [Pattern 2: BinaryHeap and Priority Queues](#)
 - [Pattern 3: Graph Representations](#)
 - [Pattern 4: Trie and Radix Tree Structures](#)
 - [Pattern 5: Lock-Free Data Structures](#)
- [Threading Patterns](#)
 - [Pattern 1: Thread Spawn and Join Patterns](#)
 - [Pattern 2: Thread Pools and Work Stealing](#)
 - [Pattern 3: Message Passing with Channels](#)
 - [Pattern 4: Shared State with Locks \(Mutex & RwLock\)](#)
 - [Pattern 5: Synchronization Primitives \(Barrier & Condvar\)](#)
- [Async Runtime Patterns](#)
 - [Pattern 1: Future Composition](#)
 - [Pattern 2: Stream Processing](#)
 - [Pattern 3: Async/Await Patterns](#)
 - [Pattern 4: Select and Timeout Patterns](#)
 - [Pattern 5: Runtime Comparison](#)
- [Atomic Operations & Lock-Free Programming](#)
 - [Pattern 1: Memory Ordering Semantics](#)
 - [Pattern 2: Compare-and-Swap Patterns](#)
 - [Pattern 3: Lock-Free Queues and Stacks](#)
 - [Pattern 4: Hazard Pointers](#)
 - [Pattern 5: Seqlock Pattern](#)
- [Parallel Algorithms](#)
 - [Pattern 1: Rayon Patterns](#)
 - [Pattern 2: Work Partitioning Strategies](#)
 - [Pattern 3: Parallel Reduce and Fold](#)
 - [Pattern 4: Pipeline Parallelism](#)
 - [Pattern 5: SIMD Parallelism](#)
- [Smart Pointer Patterns](#)
 - [Pattern 1: Box, Rc, Arc Usage Patterns](#)
 - [Pattern 2: Weak References and Cycles](#)
 - [Pattern 3: Custom Smart Pointers](#)
 - [Pattern 4: Intrusive Data Structures](#)
 - [Pattern 5: Reference Counting Optimization](#)
- [Unsafe Rust Patterns](#)
 - [Pattern 1: Raw Pointer Manipulation](#)
 - [Pattern 2: FFI and C Interop](#)
 - [Pattern 3: Uninitialized Memory Handling](#)
 - [Pattern 4: Transmute and Type Punning](#)
 - [Pattern 5: Safe Abstractions Over Unsafe](#)
 - [Pattern 5: Best Practices for Unsafe Code](#)
- [Synchronous I/O](#)
 - [Pattern 1: Basic File Operations](#)

- [Pattern 2: Buffered Reading and Writing](#)
- [Pattern 3: Standard Streams](#)
- [Pattern 4: Memory-Mapped I/O](#)
- [Pattern 5: Directory Traversal](#)
- [Pattern 6: Process Spawning and Piping](#)
- [Async I/O Patterns](#)
 - [Pattern 1: Tokio File and Network I/O](#)
 - [Pattern 2: Buffered Async Streams](#)
 - [Pattern 3: Backpressure Handling](#)
 - [Pattern 4: Connection Pooling](#)
 - [Pattern 5: Timeout and Cancellation](#)
- [Serialization Patterns](#)
 - [Pattern 1: Serde Patterns](#)
 - [Pattern 2: Zero-Copy Deserialization](#)
 - [Pattern 3: Schema Evolution](#)
 - [Pattern 4: Binary vs Text Formats](#)
 - [Pattern 5: Streaming Serialization](#)
- [Declarative Macros](#)
 - [Pattern 1: Macro Patterns and Repetition](#)
 - [Pattern 2: Hygiene and Scoping](#)
 - [Pattern 3: DSL Construction](#)
 - [Pattern 4: Code Generation Patterns](#)
 - [Pattern 5: Macro Debugging](#)
- [Procedural Macros](#)
 - [Pattern 1: Derive Macros](#)
 - [Pattern 2: Attribute Macros](#)
 - [Pattern 3: Function-like Macros](#)
 - [Pattern 4: Token Stream Manipulation](#)
 - [Pattern 5: Macro Helper Crates \(syn, quote\)](#)

Memory & Ownership Patterns

Memory & Ownership Patterns

Rust's ownership system is best understood not as a single feature, but as a foundation that enables a wide range of design patterns. This chapter focuses on **practical ownership-driven patterns** that arise once you move beyond the basics and start building real systems.

Rather than re-explaining ownership rules, we explore how Rust programmers *use* ownership, borrowing, and lifetimes to solve concrete problems such as:

- Conditional allocation and zero-copy APIs
- Safe mutation through shared references
- Coordinating shared state across threads
- Deterministic resource cleanup
- Cache-friendly memory layouts

- High-performance allocation strategies
- Custom pointer abstractions

Each pattern in this chapter answers a recurring question: > “*How do I express this design safely and efficiently within Rust’s ownership model?*”

This chapter assumes you already understand basic ownership, borrowing, and lifetimes. The goal here is to help you recognize ownership patterns in the wild—and to design your own—while keeping Rust’s core safety guarantees intact.

Pattern 1: Zero-Copy with Clone-on-Write (Cow)

- **Problem:** Functions that sometimes need to modify their input face a dilemma: always clone the input (which is wasteful if no modification is needed), or require a mutable reference (which makes the API less ergonomic).
- **Solution:** Use `Cow<T>` (Clone-on-Write). This is a smart pointer that can enclose either borrowed data (`Cow::Borrowed`) or owned data (`Cow::Owned`).
- **Why It Matters:** This pattern enables a “fast path” for zero-allocation operations. In high-throughput systems like web servers or parsers, avoiding millions of unnecessary string allocations per second can lead to significant performance gains.

Examples

Example: Conditional Modification

A common use for `Cow` is in functions that may or may not need to modify their string-like input. This `normalize_whitespace` function provides a zero-allocation “fast path”. It only allocates a new `String` and returns `Cow::Owned` if the input text actually contains characters that need to be replaced. Otherwise, it returns a borrowed slice `Cow::Borrowed` without any heap allocation.

```
use std::borrow::Cow;

// Returns borrowed data when possible, owned only when necessary
fn normalize_whitespace(text: &str) -> Cow<str> {
    if text.contains(" ") || text.contains('\t') {
        // Only allocate if we need to modify
        let mut result = text.replace(" ", " ");
        result = result.replace('\t', " ");
        Cow::Owned(result)
    } else {
        // Zero-copy return
        Cow::Borrowed(text)
    }
}
```

Example: Lazy Mutation Chains

`Cow` can be used to build a chain of potential modifications. An allocation is performed only on the first step that requires a change. This example demonstrates how a path might be processed, first by

expanding the tilde `~` and then by normalizing path separators. The `Cow` will only become `Owned` if one of these conditions is met.

```
use std::borrow::Cow;

fn process_path(path: &str) -> Cow<'str> {
    let mut result = Cow::Borrowed(path);

    // Expand tilde
    if path.starts_with("~/") {
        result = Cow::Owned(path.replace("~", "/home/user", 1));
    }

    // Normalize separators (Windows)
    if result.contains('\\') {
        result = Cow::Owned(result.replace('\\', "/"));
    }

    // Only allocates if modifications were needed
    result
}
```

Example: In-Place Modification with `to_mut()`

The `to_mut()` method is a powerful tool for getting a mutable reference to the underlying data. If the `Cow` is `Borrowed`, `to_mut()` will clone the data to make it `Owned` and then return a mutable reference. If it's already `Owned`, it returns a mutable reference without any allocation. This is perfect for efficient in-place modifications.

```
use std::borrow::Cow;

fn capitalize_first<'a>(s: &'a str) -> Cow<'a, str> {
    if let Some(first_char) = s.chars().next() {
        if first_char.is_lowercase() {
            let mut owned = s.to_string();
            owned[0..first_char.len_utf8()].make_ascii_uppercase();
            Cow::Owned(owned)
        } else {
            Cow::Borrowed(s)
        }
    } else {
        Cow::Borrowed(s)
    }
}
```

Use Case: Configuration with Defaults

`Cow` is excellent for handling configuration that involves default values. A `Config` struct can hold borrowed string slices for default values, avoiding allocations. If a user provides an override (an owned `String`), the `Cow` can seamlessly switch to holding the owned data.

```

use std::borrow::Cow;

struct Config<'a> {
    host: Cow<'a, str>,
    port: u16,
    database: Cow<'a, str>,
}

impl<'a> Config<'a> {
    fn new(host: &'a str, port: u16) -> Self {
        Config {
            host: Cow::Borrowed(host),
            port,
            // 'default_db' is a static str, so it can be borrowed safely.
            database: Cow::Borrowed("default_db"),
        }
    }

    fn with_database(mut self, db: String) -> Self {
        self.database = Cow::Owned(db);
        self
    }
}

```

When to use Cow: - Library APIs that accept string input and may need to modify it - Processing pipelines where some inputs need transformation, others don't - Configuration systems with optional overrides - Parsing where most tokens are substrings of input

Performance characteristics: - Zero allocation when borrowing - Single allocation when owned - Same size as a pointer + discriminant (24 bytes on 64-bit)

Pattern 2: Interior Mutability with Cell and RefCell

- **Problem:** Rust's borrowing rules require `&mut self` for mutation, but some designs need mutation through shared references (`&self`). Examples: caching computed values, counters in shared structures, graph nodes that need to update neighbors, observer patterns.
- **Solution:** Use interior mutability types—`Cell<T>` for `Copy` types (get/set without borrowing), `RefCell<T>` for non-`Copy` types (runtime-checked borrows). These move borrow checking from compile-time to runtime.
- **Why It Matters:** Some data structures are impossible without interior mutability. Doubly-linked lists, graphs with cycles, and the observer pattern all require mutation through shared references.

The Problem: Experiencing the Borrow Checker

Let's start by trying to implement a simple counter. We want to pass this counter to multiple functions that can increment it, but we only have a shared reference (`&Counter`). This code will not compile, because `increment` requires a mutable reference `&mut self`, but `process_item` only has an immutable one.

```

// This is our first attempt - it seems reasonable!
struct Counter {
    count: usize,
}

impl Counter {
    fn new() -> Self { Counter { count: 0 } }
    fn increment(&mut self) { self.count += 1; }
    fn get(&self) -> usize { self.count }
}

fn process_item(counter: &Counter) {
    // Inside here, we only have &Counter, not &mut Counter
    // But we need to increment!
    // counter.increment(); // ✗ ERROR: cannot call `&mut self` method with `&self`
}

```

The Solution for Copy Types: Cell<T>

For types that are `Copy` (like `usize`), `Cell<T>` solves the problem. It allows you to `get()` a copy of the value or `set()` a new value, even through a shared reference. Notice the `increment` method now takes `&self`, and it works perfectly.

```

use std::cell::Cell;

struct Counter {
    count: Cell<usize>, // Wrapped in Cell!
}

impl Counter {
    fn new() -> Self {
        Counter { count: Cell::new(0) }
    }

    fn increment(&self) { // ✅ Note: takes &self, not &mut self!
        self.count.set(self.count.get() + 1);
    }

    fn get(&self) -> usize {
        self.count.get()
    }
}

// Now this works!
fn process_item(counter: &Counter) {
    counter.increment(); // ✅ Works even with &self!
}

```

`Cell` is safe because it never gives out references to the inner data; it only moves `Copy` values in and out.

The Solution for Non-Copy Types: RefCell<T>

But what if the data isn't `Copy`, like a `Vec` or `HashMap`? You can't use `Cell`. The solution is `RefCell<T>`, which moves Rust's borrow checking rules from compile-time to *run-time*. You can ask to `borrow()` (immutable) or `borrow_mut()` (mutable). If you violate the rules (e.g., ask for a mutable borrow while an immutable one exists), your program will panic.

This example shows a cache that can be modified internally via `&self`.

```
use std::cell::RefCell;
use std::collections::HashMap;

struct Cache {
    data: RefCell<HashMap<String, String>>,
}

impl Cache {
    fn new() -> Self {
        Cache { data: RefCell::new(HashMap::new()) }
    }

    fn get_or_compute(&self, key: &str, compute: impl FnOnce() -> String) -> String {
        // Try to get from cache (immutable borrow)
        if let Some(value) = self.data.borrow().get(key) {
            return value.clone();
        }

        // Not found, compute and insert (mutable borrow)
        let value = compute();
        self.data.borrow_mut().insert(key.to_string(), value.clone());
        value
    }
}
```

RefCell Patterns and Pitfalls

Pattern: Careful Borrow Scoping

The most important pattern with `RefCell` is to keep borrow lifetimes as short as possible to avoid panics. A common way to do this is to introduce a new scope `{}`.

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

fn process_cache(cache: &RefCell<Vec<String>>) {
    // Read operation in its own scope
    {
        let borrowed = cache.borrow();
        println!("Cache size: {}", borrowed.len());
    } // `borrowed` guard is dropped here, releasing the borrow
```

```
// Write operation is now safe
cache.borrow_mut().push("new_item".to_string());
}
```

Pattern: Non-Panicking Borrows with `try_borrow`

If you're not sure if a borrow will succeed, use `try_borrow()` or `try_borrow_mut()`. These return a `Result` instead of panicking, allowing you to handle the "already borrowed" case gracefully.

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

fn safe_access(data: &RefCell<Vec<i32>>) -> Result<(), &'static str> {
    if let Ok(mut borrowed) = data.try_borrow_mut() {
        borrowed.push(42);
        Ok(())
    } else {
        Err("Could not acquire lock: data is already borrowed.")
    }
}
```

Use Case: Graph Structures

Interior mutability is essential for graph data structures or any time you have objects that point to each other and need to be modified, like a doubly-linked list. `Rc<RefCell<T>>` is a very common pattern for creating graph-like structures where nodes have shared ownership and can be mutated.

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

struct Node {
    value: i32,
    edges: RefCell<Vec<Rc<Node>>,
}

impl Node {
    fn add_edge(&self, target: Rc<Node>) {
        self.edges.borrow_mut().push(target);
    }
}
```

Summary: `Cell` vs. `RefCell`

| Feature | <code>Cell<T></code> | <code>RefCell<T></code> |
|------------|---|--|
| Works with | Copy types only | Any <code>Sized</code> type |
| API | <code>get()</code> , <code>set()</code> | <code>borrow()</code> , <code>borrow_mut()</code> |
| Checking | | Runtime (panics on violation) |

| Feature | <code>Cell<T></code> | <code>RefCell<T></code> |
|--------------|---|---|
| Overhead | Compile-time (enforced by <code>Copy</code> trait) | Small (a runtime borrow flag) |
| Panics? | No | Yes , if rules are violated |
| Thread-safe? | No | No |
| Use For | Simple <code>Copy</code> data like <code>u32, bool</code> . | Complex data like <code>Vec, HashMap</code> . |

Critical safety note: - `RefCell` is for **single-threaded** scenarios only. For multiple threads, you need `Mutex` or `RwLock`. - Always keep borrow scopes as short as possible. Never hold a borrow guard across a call to an unknown function.

Pattern 3: Thread-Safe Interior Mutability (`Mutex` & `RwLock`)

- **Problem:** `RefCell<T>` provides interior mutability but panics if used incorrectly across threads. Multi-threaded code needs safe shared mutable state—incrementing counters, updating caches, modifying shared collections—with data races.
- **Solution:** Use `Mutex<T>` for exclusive access (like `RefCell` but thread-safe) or `RwLock<T>` for reader-writer patterns (multiple readers OR one writer). Combine with `Arc<T>` to share across threads.
- **Why It Matters:** Multi-threaded programming without data races is notoriously difficult in C/C++. Rust's type system makes it impossible to compile racy code—you must use `Mutex` or `RwLock` for shared mutation.
- **Use Cases:** Shared counters in multi-threaded servers, concurrent caches, thread pools with shared work queues, parallel data processing with result aggregation, connection pools.

Examples

Example: Shared Counter Across Threads

To share mutable state across threads, you wrap it in `Arc<Mutex<T>>`. `Arc` is the “Atomically Reference Counted” pointer that lets multiple threads “own” the data. `Mutex` ensures that only one thread can access the data at a time. When `.lock()` is called, it blocks until the lock is available. The returned guard object automatically releases the lock when it goes out of scope.

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

fn parallel_counter() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];
    for _ in 0..10 {
        let handle = thread::spawn(move || {
            let mut value = counter.lock().unwrap();
            *value += 1;
        });
        handles.push(handle);
    }
    for handle in handles {
        handle.join().unwrap();
    }
    let value = counter.lock().unwrap();
    assert_eq!(*value, 10);
}
```

```

for _ in 0..10 {
    let counter_clone = Arc::clone(&counter);
    let handle = thread::spawn(move || {
        for _ in 0..100 {
            let mut num = counter_clone.lock().unwrap();
            *num += 1;
        } // lock automatically released when guard `num` is dropped
    });
    handles.push(handle);
}

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

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

```

Example: Reader-Writer Lock for Read-Heavy Workloads

A **Mutex** is exclusive. If you have a situation where many threads need to read data and only a few need to write, a **Mutex** is inefficient. **RwLock** is the solution. It allows any number of readers to access the data simultaneously, but write access is exclusive (it waits for all readers to finish).

```

use std::sync::RwLock;
use std::collections::HashMap;

struct SharedCache {
    data: RwLock<HashMap<String, String>>,
}

impl SharedCache {
    fn get(&self, key: &str) -> Option<String> {
        // Multiple readers can hold read locks simultaneously.
        self.data.read().unwrap().get(key).cloned()
    }

    fn insert(&self, key: String, value: String) {
        // Write lock is exclusive. It will wait for all readers to unlock.
        self.data.write().unwrap().insert(key, value);
    }
}

```

Example: Minimize Lock Duration

Locks can become performance bottlenecks. A critical pattern is to hold the lock for the shortest time possible. Perform expensive computations *outside* the lock, and only acquire the lock when you are ready to quickly read or write the shared data.

```

use std::sync::Mutex;

fn optimized_update(shared: &Mutex<Vec<i32>>, new_value: i32) {
    // Good: compute outside the lock
    let computed = expensive_computation(new_value);

    // Acquire lock only for the quick push operation
    shared.lock().unwrap().push(computed);
}

// Bad: holding the lock during a slow operation
fn unoptimized_update(shared: &Mutex<Vec<i32>>, new_value: i32) {
    let mut data = shared.lock().unwrap();
    let computed = expensive_computation(new_value); // Don't do this!
    data.push(computed);
}

fn expensive_computation(x: i32) -> i32 {
    std::thread::sleep(std::time::Duration::from_millis(50)); // Imagine this is slow
    x * 2
}

```

Example: Deadlock Prevention with Lock Ordering

A classic problem in concurrent programming is deadlock. If Thread 1 locks A and waits for B, while Thread 2 locks B and waits for A, they will wait forever. The solution is to ensure all threads acquire locks in a globally consistent order. A simple way to achieve this is to order locks by their memory address.

```

use std::sync::Mutex;

struct Account {
    id: u32,
    balance: Mutex<i64>,
}

fn transfer(from: &Account, to: &Account, amount: i64) {
    // To prevent deadlock, we always acquire locks in a consistent order.
    // Here, we use the account ID.
    let (lock1, lock2) = if from.id < to.id {
        (from.balance.lock().unwrap(), to.balance.lock().unwrap())
    } else {
        (to.balance.lock().unwrap(), from.balance.lock().unwrap())
    };

    // Now that locks are acquired, we can perform the logic.
    // Note: this logic is simplified and assumes the `if` branch matches the original intent.
    // A real implementation would need to handle the amounts correctly regardless of lock
    // order.
}

```

Example: Non-Blocking Access with `try_lock`

Sometimes, you don't want to wait for a lock. You'd rather do something else if the data is currently locked. `try_lock` returns immediately with a `Result`. If it acquires the lock, it returns `Ok(Guard)`; if not, it returns `Err`.

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

fn try_update(data: &Mutex<Vec<i32>>) -> Result<(), &'static str> {
    if let Ok(mut guard) = data.try_lock() {
        guard.push(42);
        Ok(())
    } else {
        Err("Lock held by another thread, skipping update.")
    }
}
```

Mutex vs RwLock trade-offs: - **Mutex**: Simpler, lower overhead, exclusive access - **RwLock**: Multiple readers, write-heavy can starve readers - RwLock ~3x slower for writes, but allows concurrent reads - Use Mutex unless >70% reads and contention is proven issue

Lock granularity strategies: - Fine-grained: More parallelism, higher overhead, deadlock risk - Coarse-grained: Less parallelism, simpler reasoning - Profile first, optimize second

Pattern 4: Custom Drop Guards

- **Problem:** Manual resource cleanup is error-prone. Forgetting to close files, release locks, or rollback transactions causes resource leaks, deadlocks, and data corruption.
- **Solution:** Implement the `Drop` trait to tie resource cleanup to scope. Create guard types that acquire resources in their constructor and release them in `Drop`.
- **Why It Matters:** RAII eliminates entire categories of bugs. You cannot forget to unlock a `Mutex`—`MutexGuard`'s `Drop` releases it automatically.

Examples

Example: Temporary File Guard

This `TempFile` struct creates a file upon construction. The `Drop` implementation ensures that no matter how the function exits—success, error, or panic—the file is guaranteed to be deleted.

```
use std::fs::File;
use std::io::{self, Write};
use std::path::{Path, PathBuf};

struct TempFile {
    path: PathBuf,
    file: File,
}
```

```

impl TempFile {
    fn new(path: impl AsRef<Path>) -> io::Result<Self> {
        let path = path.as_ref().to_path_buf();
        let file = File::create(&path)?;
        Ok(TempFile { path, file })
    }
}

impl Drop for TempFile {
    fn drop(&mut self) {
        // Cleanup happens automatically when TempFile goes out of scope.
        println!("Dropping TempFile, deleting {}", self.path.display());
        let _ = std::fs::remove_file(&self.path);
    }
}

```

Example: Custom Lock Guard

You can create your own guards that behave like `MutexGuard`. This `LockGuard` uses a `Cell<bool>` to track the lock state. When the guard is created, it sets the flag to `true`. When it's dropped, it sets it back to `false`. The `Deref` and `DerefMut` traits provide ergonomic access to the inner data.

```

use std::ops::{Deref, DerefMut};
use std::cell::Cell;

struct MyLock<T> {
    locked: Cell<bool>,
    data: T,
}

struct LockGuard<'a, T> {
    lock: &'a MyLock<T>,
}

impl<'a, T> LockGuard<'a, T> {
    fn new(lock: &'a MyLock<T>) -> Option<Self> {
        if lock.locked.get() {
            None // Already locked
        } else {
            lock.locked.set(true);
            Some(LockGuard { lock })
        }
    }
}

impl<T> Drop for LockGuard<'_, T> {
    fn drop(&mut self) {
        self.lock.locked.set(false);
    }
}

```

```

impl<T> Deref for LockGuard<'_, T> {
    type Target = T;
    fn deref(&self) -> &T {
        &self.lock.data
    }
}

```

Example: Panic-Safe State Restoration

A guard can be used to ensure state is restored, even in the case of a panic. This `StateGuard` sets a boolean flag to a new value on creation and restores the old value when it's dropped. This is useful for things like a “processing” flag.

```

struct StateGuard<'a> {
    state: &'a mut bool,
    old_value: bool,
}

impl<'a> StateGuard<'a> {
    fn new(state: &'a mut bool, new_value: bool) -> Self {
        let old_value = *state;
        *state = new_value;
        StateGuard { state, old_value }
    }
}

impl Drop for StateGuard<'_> {
    fn drop(&mut self) {
        // Restore the original state, no matter what.
        *self.state = self.old_value;
    }
}

// Usage: State is restored even if a panic occurs
fn complex_operation(processing: &mut bool) {
    let _guard = StateGuard::new(processing, true);
    // If this panics, `_guard` is dropped and `processing` is reset to its old value.
    // risky_operation();
}

```

Example: Generic Scope Guard

For arbitrary cleanup logic, a generic `ScopeGuard` can be used. It takes a closure and executes it on `drop`. This is useful for things like database transaction rollbacks. If the operation completes successfully, the guard can be `disarmed` to prevent the cleanup from running.

```

struct ScopeGuard<F: FnOnce()> {
    cleanup: Option<F>,
}

```

```

impl<F: FnOnce()> ScopeGuard<F> {
    fn new(cleanup: F) -> Self {
        ScopeGuard { cleanup: Some(cleanup) }
    }

    fn disarm(mut self) {
        self.cleanup = None;
    }
}

impl<F: FnOnce()> Drop for ScopeGuard<F> {
    fn drop(&mut self) {
        if let Some(cleanup) = self.cleanup.take() {
            cleanup();
        }
    }
}

// Usage: Generic cleanup on scope exit
fn transactional_update() {
    println!("Starting transaction...");
    let guard = ScopeGuard::new(|| {
        println!("Rolling back transaction due to error or panic.");
    });

    // perform_operations();

    // If we get here, the operation was successful.
    println!("Committing transaction.");
    guard.disarm(); // Don't run the rollback closure.
}

```

RAII benefits: - Impossible to forget cleanup - Exception-safe (panic-safe in Rust) - Scope-based reasoning about resources - Composable (guards can be nested)

Common guard patterns: - File handles (automatic close) - Locks (automatic release) - Transactions (automatic rollback) - Metrics/timers (automatic reporting) - State flags (automatic reset)

Pattern 5: Memory Layout Optimization

Problem: Naive struct definitions waste memory through padding and hurt performance via poor cache utilization. False sharing in multi-threaded code can cause 10-100x slowdowns.

Solution: Use `#[repr(C)]` for predictable layout (FFI), `#[repr(align(N))]` for cache alignment, `#[repr(packed)]` to eliminate padding (with care). Order struct fields from largest to smallest alignment.

Why It Matters: Modern CPUs are dominated by memory hierarchy—cache misses cost 100-200 cycles while arithmetic costs 1-4 cycles. A cache miss is 50-100x slower than a cache hit.

Use Cases: High-frequency trading systems, game engines, scientific computing, embedded systems, FFI with C libraries, SIMD optimization, lock-free data structures.

What is Alignment? CPUs do not read memory one byte at a time. They fetch it in chunks, typically the size of a machine word (e.g., 8 bytes on a 64-bit system). Access is fastest when a data type of size N is located at a memory address that is a multiple of N . For example, a `u64` (8 bytes) should ideally start at an address like 0, 8, 16, etc. This is its **alignment requirement**. Accessing a `u64` at an unaligned address (e.g., address 1) would be slow, as the CPU might need to perform two memory reads instead of one.

What is Padding? To satisfy these alignment requirements, the Rust compiler may insert invisible, unused bytes into a struct. This is called **padding**. The goal is to ensure every field is properly aligned.

There are two rules for a struct's layout: 1. Each field must be placed at an offset that is a multiple of its alignment. 2. The total size of the struct must be a multiple of the struct's overall alignment, which is the largest alignment of any of its fields.

Examples

Example: Field Ordering to Minimize Padding

By default, Rust reorders struct fields to minimize padding, but with `##[repr(C)]` the order is fixed. Understanding the rules helps in all cases. By ordering fields from largest to smallest, you can minimize wasted space.

```
// In this example, we use `#[repr(C)]` to disable the automatic field
// reordering that Rust would normally perform. This lets us see the
// effects of padding manually.

// Bad: 24 bytes due to padding
#[repr(C)]
struct Unoptimized {
    a: u8,
    b: u64,
    c: u8,
}
// How the compiler lays this out:
// - `a: u8` (size 1, align 1): offset 0.
// - 7 bytes of padding are added to align `b`.
// - `b: u64` (size 8, align 8): offset 8.
// - `c: u8` (size 1, align 1): offset 16.
// - 7 bytes of padding are added at the end to make the total size a multiple of 8.
// - Total size = 24 bytes.

// Good: 16 bytes by reordering fields
#[repr(C)]
struct Optimized {
    b: u64, // Largest alignment first
    a: u8,
    c: u8,
}
// How this improves things:
// - `b: u64`: offset 0.
// - `a: u8`: offset 8.
```

```

// - `c: u8` : offset 9.
// - 6 bytes of padding at the end makes the total size 16.
// - Total size = 16 bytes.

// Verify sizes
const _: () = assert!(std::mem::size_of::<Unoptimized>() == 24);
const _: () = assert!(std::mem::size_of::<Optimized>() == 16);

```

Example: Layout Attributes `#[repr(...)]`

Rust provides attributes to control memory layout.

- `#[repr(C)]`: Guarantees the same layout as a C struct. Essential for FFI.
- `#[repr(packed)]`: Removes all padding. This can lead to unaligned-access performance penalties or even crashes on some architectures. Use with extreme care.
- `#[repr(align(N))]`: Forces the struct's alignment to be at least `N` bytes.
- `#[repr(u8)]`: Specifies the memory representation for an enum's discriminant.

```

// For FFI compatibility
#[repr(C)]
struct Point {
    x: f64,
    y: f64,
}

// To eliminate padding (use carefully!)
#[repr(packed)]
struct Packed {
    a: u8,
    b: u32, // `b` may be at an unaligned address
}

// To align to a cache line (e.g., 64 bytes)
#[repr(align(64))]
struct CacheAligned {
    data: [u8; 64],
}

// To define an enum's size
#[repr(u8)]
enum Status {
    Idle = 0,
    Running = 1,
    Failed = 2,
}

```

Example: Preventing False Sharing

False sharing is a silent performance killer in multi-threaded code. It happens when two threads write to different variables that happen to live on the same CPU cache line. The CPU's cache coherency protocol forces the cores to fight over the cache line, serializing execution. The fix is to pad data to ensure contended variables are on different cache lines.

```

use std::sync::atomic::AtomicUsize;

const CACHE_LINE_SIZE: usize = 64;

#[repr(align(CACHE_LINE_SIZE))]
struct Padded<T> {
    value: T,
}

// With this structure, counter1 and counter2 are guaranteed to be on
// different cache lines, preventing false sharing when updated by different threads.
struct SharedCounters {
    counter1: Padded<AtomicUsize>,
    counter2: Padded<AtomicUsize>,
}

```

Example: Optimizing Enum Size

An enum's size is determined by its largest variant. If one variant is huge, the whole enum becomes huge. To fix this, you can **Box** the large variant. This makes the variant a pointer, and the enum's size becomes the size of the pointer plus a tag, which is much smaller.

```

// Bad: Size is over 1024 bytes
enum LargeEnum {
    Small(u8),
    Big([u8; 1024]),
}

// Good: Size is the size of a Box (a pointer) + a tag.
enum OptimizedEnum {
    Small(u8),
    Big(Box<[u8; 1024]>),
}

```

Example: Data-Oriented Design (SoA vs. AoS)

For performance-critical loops, memory access patterns are key. "Array of Structs" (AoS) is common but can be bad for cache performance if you only need one field per iteration. "Struct of Arrays" (SoA) organizes the data by field, ensuring that when you iterate over one field, all the data for that field is contiguous in memory.

```

// Bad: Array of Structs (AoS) – poor cache locality for single-field access
struct ParticleAoS {
    position: [f32; 3],
    velocity: [f32; 3],
    mass: f32,
}

fn update_aos(particles: &mut [ParticleAoS]) {
    for p in particles {

```

```

    // When accessing p.position, the CPU loads the entire struct (position,
    // velocity, mass) into the cache, even though we don't need the other fields.
    p.position[0] += p.velocity[0];
}

// Good: Struct of Arrays (SoA) - excellent cache locality
struct ParticlesSoA {
    positions_x: Vec<f32>,
    velocities_x: Vec<f32>,
    // ... and so on for other fields
}

impl ParticlesSoA {
    fn update_positions(&mut self) {
        // All the x positions are contiguous in memory. The CPU can prefetch
        // them efficiently, leading to far fewer cache misses.
        for i in 0..self.positions_x.len() {
            self.positions_x[i] += self.velocities_x[i];
        }
    }
}

```

Memory layout principles: - Order struct fields from largest to smallest alignment - Use `#[repr(C)]` when layout matters (FFI, serialization) - Pad to cache lines (64 bytes) to prevent false sharing - Box large enum variants to keep enum size small - Consider SoA over AoS for performance-critical loops

Performance characteristics: - False sharing can degrade performance by 10-100x - Proper alignment enables SIMD operations - Cache line is typically 64 bytes - L1 cache miss: ~4 cycles, L3 miss: ~40 cycles, RAM: ~200 cycles

Pattern 6: Arena Allocation

- **Problem:** Allocating many small objects with `Box::new()` or `Vec::push()` is slow. Each call invokes the system's general-purpose allocator (`malloc`), which involves locking and metadata overhead.
- **Solution:** Use an arena allocator (also called a bump allocator). Pre-allocate a large, contiguous chunk of memory.
- **Why It Matters:** Arena allocation is 10-100x faster than general-purpose allocators for scenarios involving many small objects. For applications like compilers (which create millions of AST nodes) or web servers (which create objects per-request), this can dramatically improve performance by reducing allocation bottlenecks.

Examples

```

//=====
// Pattern: Simple arena allocator
//=====

struct Arena {
    chunks: Vec<Vec<u8>>,

```

```

        current: Vec<u8>,
        position: usize,
    }

impl Arena {
    fn new() -> Self {
        Arena {
            chunks: Vec::new(),
            current: vec![0; 4096],
            position: 0,
        }
    }

    fn alloc<T>(&mut self, value: T) -> &mut T {
        let size = std::mem::size_of::<T>();
        let align = std::mem::align_of::<T>();

        // Align position
        let padding = (align - (self.position % align)) % align;
        self.position += padding;

        // Check if we need a new chunk
        if self.position + size > self.current.len() {
            let old = std::mem::replace(&mut self.current, vec![0; 4096]);
            self.chunks.push(old);
            self.position = 0;
        }

        // Allocate
        let ptr = &mut self.current[self.position] as *mut u8 as *mut T;
        self.position += size;

        unsafe {
            std::ptr::write(ptr, value);
            &mut *ptr
        }
    }
}

//=====
// Use case: AST nodes during parsing
//=====

struct AstArena {
    arena: Arena,
}

enum Expr<'a> {
    Number(i64),
    Add(&'a Expr<'a>, &'a Expr<'a>),
    Multiply(&'a Expr<'a>, &'a Expr<'a>),
}

impl AstArena {

```

```

fn new() -> Self {
    AstArena { arena: Arena::new() }
}

fn number(&mut self, n: i64) -> &Expr {
    self.arena.alloc(Expr::Number(n))
}

fn add<'a>(&'a mut self, left: &'a Expr, right: &'a Expr) -> &'a Expr<'a> {
    self.arena.alloc(Expr::Add(left, right))
}
}

//=====
// Pattern: Typed arena with better ergonomics
//=====

use typed_arena::Arena as TypedArena;

struct Parser<'ast> {
    arena: &'ast TypedArena<Expr<'ast>>,
}

impl<'ast> Parser<'ast> {
    fn parse_number(&self, n: i64) -> &'ast Expr<'ast> {
        self.arena.alloc(Expr::Number(n))
    }

    fn parse_binary(&self, left: &'ast Expr<'ast>, right: &'ast Expr<'ast>)
        -> &'ast Expr<'ast>
    {
        self.arena.alloc(Expr::Add(left, right))
    }
}

//=====
// Pattern: Arena for temporary string allocations
//=====

struct StringArena {
    arena: TypedArena<String>,
}

impl StringArena {
    fn new() -> Self {
        StringArena { arena: TypedArena::new() }
    }

    fn alloc(&self, s: &str) -> &str {
        let owned = self.arena.alloc(s.to_string());
        owned.as_str()
    }
}

//=====

```

```
// Use case: Request-scoped allocations in web server
//=====
struct RequestContext<'arena> {
    arena: &'arena TypedArena<Vec<u8>>,
}

impl<'arena> RequestContext<'arena> {
    fn allocate_buffer(&self, size: usize) -> &'arena mut Vec<u8> {
        self.arena.alloc(vec![0; size])
    }
}
```

When to use arenas: - Compiler frontends (AST, IR nodes) - Request handlers in servers - Graph algorithms with temporary nodes - Game engine frame allocations - Any scenario with bulk deallocation

Performance characteristics: - Allocation: O(1), just increment pointer - Deallocation: O(1), drop entire arena - 10-100x faster than malloc/free for small objects - Better cache locality (allocated objects are contiguous) - Cannot free individual objects (trade-off)

Pattern 7: Custom Smart Pointers

- **Problem:** The standard smart pointers (`Box`, `Rc`, `Arc`) are excellent general-purpose tools, but they have limitations. `Rc/Arc` require a separate heap allocation for their reference counts, and simple vector indices can be invalidated by insertions or removals.
- **Solution:** Build custom smart pointers using `unsafe` Rust primitives like `NonNull<T>`, `PhantomData`, and the `Deref`, `DerefMut`, and `Drop` traits. This allows for patterns like intrusive reference counting (where the count is stored in the object itself) or generational indices (which prevent use-after-free errors with vector-like containers).
- **Why It Matters:** Custom smart pointers unlock performance and memory layout patterns that are impossible with standard types. An intrusive `Rc` can save one allocation per object, which is critical when creating millions of them.

Examples

Example: Intrusive Reference Counting

Standard `Rc` and `Arc` perform two allocations: one for the object, and one for the reference-count block. An *intrusive* counter stores the count inside the object itself, saving an allocation. This is critical when you have millions of small, reference-counted objects. This example shows a simplified intrusive `Rc`.

```
use std::ptr::NonNull;
use std::marker::PhantomData;
use std::cell::Cell;
use std::ops::Deref;

// The data and its refcount live in the same heap allocation.
struct IntrusiveNode<T> {
    refcount: Cell<usize>,
    data: T,
```

```

}

struct IntrusiveRc<T> {
    ptr: NonNull<IntrusiveNode<T>>,
    _marker: PhantomData<T>,
}

impl<T> IntrusiveRc<T> {
    fn new(data: T) -> Self {
        let node = Box::new(IntrusiveNode {
            refcount: Cell::new(1),
            data,
        });
        IntrusiveRc {
            ptr: unsafe { NonNull::new_unchecked(Box::into_raw(node)) },
            _marker: PhantomData,
        }
    }
}

impl<T> Clone for IntrusiveRc<T> {
    fn clone(&self) -> Self {
        let node = unsafe { self.ptr.as_ref() };
        let count = node.refcount.get();
        node.refcount.set(count + 1);
        IntrusiveRc { ptr: self.ptr, _marker: PhantomData }
    }
}

impl<T> Drop for IntrusiveRc<T> {
    fn drop(&mut self) {
        unsafe {
            let node = self.ptr.as_ref();
            let count = node.refcount.get();
            if count == 1 {
                // Last reference, so deallocate the whole Box.
                drop(Box::from_raw(self.ptr.as_ptr()));
            } else {
                // Decrement the refcount.
                node.refcount.set(count - 1);
            }
        }
    }
}

impl<T> Deref for IntrusiveRc<T> {
    type Target = T;
    fn deref(&self) -> &T {
        unsafe { &self.ptr.as_ref().data }
    }
}

```

Example: Generational Arena for Stable Handles

When you store objects in a `Vec`, their indices are not stable. If you remove an element from the middle, all subsequent indices change. A **generational arena** solves this. It gives you a stable `Handle` (or ID) for an object. The handle contains both an index and a “generation” number. When an object is removed, its slot is marked free, and its generation is incremented. If old code tries to use a stale handle, the generation numbers won’t match, preventing use-after-free bugs. This is a cornerstone of modern Entity-Component-System (ECS) game engines.

```
#[derive(Clone, Copy, PartialEq, Eq)]
struct Handle {
    index: usize,
    generation: u64,
}

struct Slot<T> {
    value: Option<T>,
    generation: u64,
}

struct GenerationalArena<T> {
    slots: Vec<Slot<T>>,
    free_list: Vec<usize>,
}

impl<T> GenerationalArena<T> {
    fn new() -> Self {
        GenerationalArena { slots: Vec::new(), free_list: Vec::new() }
    }

    fn insert(&mut self, value: T) -> Handle {
        if let Some(index) = self.free_list.pop() {
            let slot = &mut self.slots[index];
            slot.generation += 1;
            slot.value = Some(value);
            Handle { index, generation: slot.generation }
        } else {
            let index = self.slots.len();
            self.slots.push(Slot { value: Some(value), generation: 0 });
            Handle { index, generation: 0 }
        }
    }

    fn get(&self, handle: Handle) -> Option<&T> {
        self.slots.get(handle.index)
            .filter(|slot| slot.generation == handle.generation)
            .and_then(|slot| slot.value.as_ref())
    }

    fn remove(&mut self, handle: Handle) -> Option<T> {
        if let Some(slot) = self.slots.get_mut(handle.index) {
            if slot.generation == handle.generation {
```

```

        self.free_list.push(handle.index);
        slot.generation += 1; // Invalidate existing handles
        return slot.value.take();
    }
}
None
}
}

```

Example: Copy-on-Write Smart Pointer

This custom `Immutable<T>` pointer makes a type immutable by default, but allows for cheap clones. Clones share the same underlying data. Only when `modify` is called does the data get copied, ensuring that modifications don't affect other copies. This is a simplified, custom version of the standard library's `Cow`.

```

use std::rc::Rc;
use std::ops::Deref;

struct Immutable<T: Clone> {
    data: Rc<T>,
}

impl<T: Clone> Immutable<T> {
    fn new(data: T) -> Self {
        Immutable { data: Rc::new(data) }
    }

    fn modify<F>(&mut self, f: F)
    where
        F: FnOnce(&mut T),
    {
        // If the data is shared (more than one reference exists)...
        if Rc::strong_count(&self.data) > 1 {
            // ...clone it to create a new, unique copy.
            self.data = Rc::new(*self.data).clone();
        }
        // Now we have the only reference, so we can safely get a mutable one.
        let data_mut = Rc::get_mut(&mut self.data).unwrap();
        f(data_mut);
    }
}

impl<T: Clone> Deref for Immutable<T> {
    type Target = T;
    fn deref(&self) -> &T {
        &self.data
    }
}

impl<T: Clone> Clone for Immutable<T> {

```

```

fn clone(&self) -> Self {
    // Cloning is cheap: it just clones the Rc, incrementing the ref count.
    Immutable {
        data: Rc::clone(&self.data),
    }
}

```

When to build custom smart pointers: - Specialized allocation patterns (pools, arenas) - Intrusive data structures for cache efficiency - Game engines (generational indices) - Systems with unique ownership semantics - Performance-critical code where std overhead matters

Performance Summary

| Pattern | Allocation Cost | Access Cost | Best Use Case |
|------------|-----------------|-----------------|--------------------------------------|
| Box<T> | O(1) heap | O(1) | Heap allocation, trait objects |
| Rc<T> | O(1) heap | O(1) + refcount | Shared ownership, single-threaded |
| Arc<T> | O(1) heap | O(1) + atomic | Shared ownership, multi-threaded |
| Cow<T> | O(0) or O(n) | O(1) | Conditional cloning |
| RefCell<T> | O(0) | O(1) + check | Interior mutability, single-threaded |
| Mutex<T> | O(0) | O(lock) | Interior mutability, multi-threaded |
| Arena | O(1) bump | O(1) | Bulk allocation/deallocation |

Common Anti-Patterns

```

// ❌ Holding RefCell borrow across function call
let borrowed = data.borrow();
might_borrow_again(&data); // Runtime panic!

// ✓ Scope borrows tightly
{
    let borrowed = data.borrow();
    use_data(&borrowed);
} // Dropped here
might_borrow_again(&data); // Safe

// ❌ Arc<Mutex<T>> when single-threaded
let shared = Arc::new(Mutex::new(data)); // Unnecessary overhead

// ✓ Use Rc<RefCell<T>> for single-threaded
let shared = Rc::new(RefCell::new(data));

// ❌ Cloning Cow unnecessarily
fn process(s: Cow<str>) -> String {

```

```

    s.into_owned() // Always allocates
}

// ✓ Return Cow to defer cloning
fn process(s: &str) -> Cow<str> {
    if needs_modification(s) {
        Cow::Owned(modify(s))
    } else {
        Cow::Borrowed(s)
    }
}

fn needs_modification(_s: &str) -> bool { true }
fn modify(s: &str) -> String { s.to_uppercase() }

```

Rust's ownership system is its defining feature, enabling memory safety without garbage collection. This chapter explores advanced patterns that leverage ownership, borrowing, and lifetimes to write efficient, safe code. For experienced programmers, understanding these patterns is crucial for designing high-performance systems where memory allocation, cache locality, and zero-copy operations matter.

The ownership model enforces three fundamental rules at compile time:

1. Each value has exactly one owner
2. Values are dropped when their owner goes out of scope
3. References must never outlive their referents

These rules enable sophisticated zero-cost abstractions while preventing entire classes of bugs: use-after-free, double-free, dangling pointers, and data races.

Pattern 1: Zero-Copy with Clone-on-Write (Cow)

- **Problem:** Functions that sometimes need to modify their input face a dilemma: always clone the input (which is wasteful if no modification is needed), or require a mutable reference (which makes the API less ergonomic).
- **Solution:** Use `Cow<T>` (Clone-on-Write). This is a smart pointer that can enclose either borrowed data (`Cow::Borrowed`) or owned data (`Cow::Owned`).
- **Why It Matters:** This pattern enables a “fast path” for zero-allocation operations. In high-throughput systems like web servers or parsers, avoiding millions of unnecessary string allocations per second can lead to significant performance gains.

Examples

Example: Conditional Modification

A common use for `Cow` is in functions that may or may not need to modify their string-like input. This `normalize_whitespace` function provides a zero-allocation “fast path”. It only allocates a new `String` and returns `Cow::Owned` if the input text actually contains characters that need to be replaced. Otherwise, it returns a borrowed slice `Cow::Borrowed` without any heap allocation.

```

use std::borrow::Cow;

// Returns borrowed data when possible, owned only when necessary
fn normalize_whitespace(text: &str) -> Cow<str> {
    if text.contains(" ") || text.contains('\t') {
        // Only allocate if we need to modify
        let mut result = text.replace(" ", " ");
        result = result.replace('\t', " ");
        Cow::Owned(result)
    } else {
        // Zero-copy return
        Cow::Borrowed(text)
    }
}

```

Example: Lazy Mutation Chains

`Cow` can be used to build a chain of potential modifications. An allocation is performed only on the first step that requires a change. This example demonstrates how a path might be processed, first by expanding the tilde `~` and then by normalizing path separators. The `Cow` will only become `Owned` if one of these conditions is met.

```

use std::borrow::Cow;

fn process_path(path: &str) -> Cow<str> {
    let mut result = Cow::Borrowed(path);

    // Expand tilde
    if path.starts_with("~/") {
        result = Cow::Owned(path.replace("~/", "/home/user", 1));
    }

    // Normalize separators (Windows)
    if result.contains('\\') {
        result = Cow::Owned(result.replace('\\', "/"));
    }

    // Only allocates if modifications were needed
    result
}

```

Example: In-Place Modification with `to_mut()`

The `to_mut()` method is a powerful tool for getting a mutable reference to the underlying data. If the `Cow` is `Borrowed`, `to_mut()` will clone the data to make it `Owned` and then return a mutable reference. If it's already `Owned`, it returns a mutable reference without any allocation. This is perfect for efficient in-place modifications.

```

use std::borrow::Cow;

fn capitalize_first<'a>(s: &'a str) -> Cow<'a, str> {
    if let Some(first_char) = s.chars().next() {
        if first_char.is_lowercase() {
            let mut owned = s.to_string();
            owned[0..first_char.len_utf8()].make_ascii_uppercase();
            Cow::Owned(owned)
        } else {
            Cow::Borrowed(s)
        }
    } else {
        Cow::Borrowed(s)
    }
}

```

Use Case: Configuration with Defaults

`Cow` is excellent for handling configuration that involves default values. A `Config` struct can hold borrowed string slices for default values, avoiding allocations. If a user provides an override (an owned `String`), the `Cow` can seamlessly switch to holding the owned data.

```

use std::borrow::Cow;

struct Config<'a> {
    host: Cow<'a, str>,
    port: u16,
    database: Cow<'a, str>,
}

impl<'a> Config<'a> {
    fn new(host: &'a str, port: u16) -> Self {
        Config {
            host: Cow::Borrowed(host),
            port,
            // 'default_db' is a &'static str, so it can be borrowed safely.
            database: Cow::Borrowed("default_db"),
        }
    }

    fn with_database(mut self, db: String) -> Self {
        self.database = Cow::Owned(db);
        self
    }
}

```

When to use `Cow`: - Library APIs that accept string input and may need to modify it - Processing pipelines where some inputs need transformation, others don't - Configuration systems with optional overrides - Parsing where most tokens are substrings of input

Performance characteristics: - Zero allocation when borrowing - Single allocation when owned - Same size as a pointer + discriminant (24 bytes on 64-bit)

Pattern 2: Interior Mutability with Cell and RefCell

- **Problem:** Rust's borrowing rules require `&mut self` for mutation, but some designs need mutation through shared references (`&self`). Examples: caching computed values, counters in shared structures, graph nodes that need to update neighbors, observer patterns.
- **Solution:** Use interior mutability types—`Cell<T>` for `Copy` types (get/set without borrowing), `RefCell<T>` for non-`Copy` types (runtime-checked borrows). These move borrow checking from compile-time to runtime.
- **Why It Matters:** Some data structures are impossible without interior mutability. Doubly-linked lists, graphs with cycles, and the observer pattern all require mutation through shared references.

The Problem: Experiencing the Borrow Checker

Let's start by trying to implement a simple counter. We want to pass this counter to multiple functions that can increment it, but we only have a shared reference (`&Counter`). This code will not compile, because `increment` requires a mutable reference `&mut self`, but `process_item` only has an immutable one.

```
// This is our first attempt – it seems reasonable!
struct Counter {
    count: usize,
}

impl Counter {
    fn new() -> Self { Counter { count: 0 } }
    fn increment(&mut self) { self.count += 1; }
    fn get(&self) -> usize { self.count }
}

fn process_item(counter: &Counter) {
    // Inside here, we only have &Counter, not &mut Counter
    // But we need to increment!
    // counter.increment(); // ✘ ERROR: cannot call `&mut self` method with `&self`
}
```

The Solution for `Copy` Types: `Cell<T>`

For types that are `Copy` (like `usize`), `Cell<T>` solves the problem. It allows you to `get()` a copy of the value or `set()` a new value, even through a shared reference. Notice the `increment` method now takes `&self`, and it works perfectly.

```
use std::cell::Cell;

struct Counter {
    count: Cell<usize>, // Wrapped in Cell!
}
```

```

impl Counter {
    fn new() -> Self {
        Counter { count: Cell::new(0) }
    }

    fn increment(&self) { // ✅ Note: takes &self, not &mut self!
        self.count.set(self.count.get() + 1);
    }

    fn get(&self) -> usize {
        self.count.get()
    }
}

// Now this works!
fn process_item(counter: &Counter) {
    counter.increment(); // ✅ Works even with &self!
}

```

`Cell` is safe because it never gives out references to the inner data; it only moves `Copy` values in and out.

The Solution for Non-`Copy` Types: `RefCell<T>`

But what if the data isn't `Copy`, like a `Vec` or `HashMap`? You can't use `Cell`. The solution is `RefCell<T>`, which moves Rust's borrow checking rules from compile-time to *run-time*. You can ask to `borrow()` (immutable) or `borrow_mut()` (mutable). If you violate the rules (e.g., ask for a mutable borrow while an immutable one exists), your program will panic.

This example shows a cache that can be modified internally via `&self`.

```

use std::cell::RefCell;
use std::collections::HashMap;

struct Cache {
    data: RefCell<HashMap<String, String>>,
}

impl Cache {
    fn new() -> Self {
        Cache { data: RefCell::new(HashMap::new()) }
    }

    fn get_or_compute(&self, key: &str, compute: impl FnOnce() -> String) -> String {
        // Try to get from cache (immutable borrow)
        if let Some(value) = self.data.borrow().get(key) {
            return value.clone();
        }

        // Not found, compute and insert (mutable borrow)
    }
}

```

```

        let value = compute();
        self.data.borrow_mut().insert(key.to_string(), value.clone());
        value
    }
}

```

RefCell Patterns and Pitfalls

Pattern: Careful Borrow Scoping

The most important pattern with `RefCell` is to keep borrow lifetimes as short as possible to avoid panics. A common way to do this is to introduce a new scope `{}`.

```

use std::cell::RefCell;

fn process_cache(cache: &RefCell<Vec<String>>) {
    // Read operation in its own scope
    {
        let borrowed = cache.borrow();
        println!("Cache size: {}", borrowed.len());
    } // `borrowed` guard is dropped here, releasing the borrow

    // Write operation is now safe
    cache.borrow_mut().push("new_item".to_string());
}

```

Pattern: Non-Panicking Borrows with `try_borrow`

If you're not sure if a borrow will succeed, use `try_borrow()` or `try_borrow_mut()`. These return a `Result` instead of panicking, allowing you to handle the "already borrowed" case gracefully.

```

use std::cell::RefCell;

fn safe_access(data: &RefCell<Vec<i32>>) -> Result<(), &'static str> {
    if let Ok(mut borrowed) = data.try_borrow_mut() {
        borrowed.push(42);
        Ok(())
    } else {
        Err("Could not acquire lock: data is already borrowed.")
    }
}

```

Use Case: Graph Structures

Interior mutability is essential for graph data structures or any time you have objects that point to each other and need to be modified, like a doubly-linked list. `Rc<RefCell<T>>` is a very common pattern for creating graph-like structures where nodes have shared ownership and can be mutated.

```

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

struct Node {
    value: i32,
    edges: RefCell<Vec<Rc<Node>>,
}

impl Node {
    fn add_edge(&self, target: Rc<Node>) {
        self.edges.borrow_mut().push(target);
    }
}

```

Summary: Cell vs. RefCell

| Feature | Cell<T> | RefCell<T> |
|--------------|--|---------------------------------|
| Works with | Copy types only | Any Sized type |
| API | get(), set() | borrow(), borrow_mut() |
| Checking | Compile-time (enforced by Copy trait) | Runtime (panics on violation) |
| Overhead | Zero | Small (a runtime borrow flag) |
| Panics? | No | Yes, if rules are violated |
| Thread-safe? | No | No |
| Use For | Simple Copy data like u32, bool. | Complex data like Vec, HashMap. |

Critical safety note: - `RefCell` is for **single-threaded** scenarios only. For multiple threads, you need `Mutex` or `RwLock`. - Always keep borrow scopes as short as possible. Never hold a borrow guard across a call to an unknown function.

Pattern 3: Thread-Safe Interior Mutability (Mutex & RwLock)

- **Problem:** `RefCell<T>` provides interior mutability but panics if used incorrectly across threads. Multi-threaded code needs safe shared mutable state—incrementing counters, updating caches, modifying shared collections—without data races.
- **Solution:** Use `Mutex<T>` for exclusive access (like `RefCell` but thread-safe) or `RwLock<T>` for reader-writer patterns (multiple readers OR one writer). Combine with `Arc<T>` to share across threads.

- **Why It Matters:** Multi-threaded programming without data races is notoriously difficult in C/C++. Rust's type system makes it impossible to compile racy code—you must use **Mutex** or **RwLock** for shared mutation.
- **Use Cases:** Shared counters in multi-threaded servers, concurrent caches, thread pools with shared work queues, parallel data processing with result aggregation, connection pools.

Examples

Example: Shared Counter Across Threads

To share mutable state across threads, you wrap it in `Arc<Mutex<T>>`. **Arc** is the “Atomically Reference Counted” pointer that lets multiple threads “own” the data. **Mutex** ensures that only one thread can access the data at a time. When `.lock()` is called, it blocks until the lock is available. The returned guard object automatically releases the lock when it goes out of scope.

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

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

    for _ in 0..10 {
        let counter_clone = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            for _ in 0..100 {
                let mut num = counter_clone.lock().unwrap();
                *num += 1;
            } // lock automatically released when guard `num` is dropped
        });
        handles.push(handle);
    }

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

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

Example: Reader-Writer Lock for Read-Heavy Workloads

A **Mutex** is exclusive. If you have a situation where many threads need to read data and only a few need to write, a **Mutex** is inefficient. **RwLock** is the solution. It allows any number of readers to access the data simultaneously, but write access is exclusive (it waits for all readers to finish).

```
use std::sync::RwLock;
use std::collections::HashMap;
```

```

struct SharedCache {
    data: RwLock<HashMap<String, String>>,
}

impl SharedCache {
    fn get(&self, key: &str) -> Option<String> {
        // Multiple readers can hold read locks simultaneously.
        self.data.read().unwrap().get(key).cloned()
    }

    fn insert(&self, key: String, value: String) {
        // Write lock is exclusive. It will wait for all readers to unlock.
        self.data.write().unwrap().insert(key, value);
    }
}

```

Example: Minimize Lock Duration

Locks can become performance bottlenecks. A critical pattern is to hold the lock for the shortest time possible. Perform expensive computations *outside* the lock, and only acquire the lock when you are ready to quickly read or write the shared data.

```

use std::sync::Mutex;

fn optimized_update(shared: &Mutex<Vec<i32>>, new_value: i32) {
    // Good: compute outside the lock
    let computed = expensive_computation(new_value);

    // Acquire lock only for the quick push operation
    shared.lock().unwrap().push(computed);
}

// Bad: holding the lock during a slow operation
fn unoptimized_update(shared: &Mutex<Vec<i32>>, new_value: i32) {
    let mut data = shared.lock().unwrap();
    let computed = expensive_computation(new_value); // Don't do this!
    data.push(computed);
}

fn expensive_computation(x: i32) -> i32 {
    std::thread::sleep(std::time::Duration::from_millis(50)); // Imagine this is slow
    x * 2
}

```

Example: Deadlock Prevention with Lock Ordering

A classic problem in concurrent programming is deadlock. If Thread 1 locks A and waits for B, while Thread 2 locks B and waits for A, they will wait forever. The solution is to ensure all threads acquire locks in a globally consistent order. A simple way to achieve this is to order locks by their memory address.

```

use std::sync::Mutex;

struct Account {
    id: u32,
    balance: Mutex<i64>,
}

fn transfer(from: &Account, to: &Account, amount: i64) {
    // To prevent deadlock, we always acquire locks in a consistent order.
    // Here, we use the account ID.
    let (lock1, lock2) = if from.id < to.id {
        (from.balance.lock().unwrap(), to.balance.lock().unwrap())
    } else {
        (to.balance.lock().unwrap(), from.balance.lock().unwrap())
    };

    // Now that locks are acquired, we can perform the logic.
    // Note: this logic is simplified and assumes the `if` branch matches the original intent.
    // A real implementation would need to handle the amounts correctly regardless of lock
    // order.
}

```

Example: Non-Blocking Access with `try_lock`

Sometimes, you don't want to wait for a lock. You'd rather do something else if the data is currently locked. `try_lock` returns immediately with a `Result`. If it acquires the lock, it returns `Ok(Guard)`; if not, it returns `Err`.

```

use std::sync::Mutex;

fn try_update(data: &Mutex<Vec<i32>>) -> Result<(), &'static str> {
    if let Ok(mut guard) = data.try_lock() {
        guard.push(42);
        Ok(())
    } else {
        Err("Lock held by another thread, skipping update.")
    }
}

```

Mutex vs RwLock trade-offs: - **Mutex:** Simpler, lower overhead, exclusive access - **RwLock:** Multiple readers, write-heavy can starve readers - RwLock ~3x slower for writes, but allows concurrent reads - Use Mutex unless >70% reads and contention is proven issue

Lock granularity strategies: - Fine-grained: More parallelism, higher overhead, deadlock risk - Coarse-grained: Less parallelism, simpler reasoning - Profile first, optimize second

Pattern 4: Custom Drop Guards

- **Problem:** Manual resource cleanup is error-prone. Forgetting to close files, release locks, or rollback transactions causes resource leaks, deadlocks, and data corruption.

- **Solution:** Implement the `Drop` trait to tie resource cleanup to scope. Create guard types that acquire resources in their constructor and release them in `Drop`.
- **Why It Matters:** RAII eliminates entire categories of bugs. You cannot forget to unlock a `Mutex`—`MutexGuard`'s `Drop` releases it automatically.

Examples

Example: Temporary File Guard

This `TempFile` struct creates a file upon construction. The `Drop` implementation ensures that no matter how the function exits—success, error, or panic—the file is guaranteed to be deleted.

```
use std::fs::File;
use std::io::{self, Write};
use std::path::{Path, PathBuf};

struct TempFile {
    path: PathBuf,
    file: File,
}

impl TempFile {
    fn new(path: impl AsRef) -> io::Result<Self> {
        let path = path.as_ref().to_path_buf();
        let file = File::create(&path)?;
        Ok(TempFile { path, file })
    }
}

impl Drop for TempFile {
    fn drop(&mut self) {
        // Cleanup happens automatically when TempFile goes out of scope.
        println!("Dropping TempFile, deleting {}", self.path.display());
        let _ = std::fs::remove_file(&self.path);
    }
}
```

Example: Custom Lock Guard

You can create your own guards that behave like `MutexGuard`. This `LockGuard` uses a `Cell<bool>` to track the lock state. When the guard is created, it sets the flag to `true`. When it's dropped, it sets it back to `false`. The `Deref` and `DerefMut` traits provide ergonomic access to the inner data.

```
use std::ops::{Deref, DerefMut};
use std::cell::Cell;

struct MyLock<T> {
    locked: Cell<bool>,
    data: T,
}
```

```

struct LockGuard<'a, T> {
    lock: &'a MyLock<T>,
}

impl<'a, T> LockGuard<'a, T> {
    fn new(lock: &'a MyLock<T>) -> Option<Self> {
        if lock.locked.get() {
            None // Already locked
        } else {
            lock.locked.set(true);
            Some(LockGuard { lock })
        }
    }
}

impl<T> Drop for LockGuard<'_, T> {
    fn drop(&mut self) {
        self.lock.locked.set(false);
    }
}

impl<T> Deref for LockGuard<'_, T> {
    type Target = T;
    fn deref(&self) -> &T {
        &self.lock.data
    }
}

```

Example: Panic-Safe State Restoration

A guard can be used to ensure state is restored, even in the case of a panic. This **StateGuard** sets a boolean flag to a new value on creation and restores the old value when it's dropped. This is useful for things like a "processing" flag.

```

struct StateGuard<'a> {
    state: &'a mut bool,
    old_value: bool,
}

impl<'a> StateGuard<'a> {
    fn new(state: &'a mut bool, new_value: bool) -> Self {
        let old_value = *state;
        *state = new_value;
        StateGuard { state, old_value }
    }
}

impl Drop for StateGuard<'_> {
    fn drop(&mut self) {
        // Restore the original state, no matter what.
    }
}

```

```

        *self.state = self.old_value;
    }
}

// Usage: State is restored even if a panic occurs
fn complex_operation(processing: &mut bool) {
    let _guard = StateGuard::new(processing, true);
    // If this panics, `_guard` is dropped and `processing` is reset to its old value.
    // risky_operation();
}

```

Example: Generic Scope Guard

For arbitrary cleanup logic, a generic `ScopeGuard` can be used. It takes a closure and executes it on `drop`. This is useful for things like database transaction rollbacks. If the operation completes successfully, the guard can be `disarmed` to prevent the cleanup from running.

```

struct ScopeGuard<F: FnOnce()> {
    cleanup: Option<F>,
}

impl<F: FnOnce()> ScopeGuard<F> {
    fn new(cleanup: F) -> Self {
        ScopeGuard { cleanup: Some(cleanup) }
    }

    fn disarm(&mut self) {
        self.cleanup = None;
    }
}

impl<F: FnOnce()> Drop for ScopeGuard<F> {
    fn drop(&mut self) {
        if let Some(cleanup) = self.cleanup.take() {
            cleanup();
        }
    }
}

// Usage: Generic cleanup on scope exit
fn transactional_update() {
    println!("Starting transaction...");
    let guard = ScopeGuard::new(|| {
        println!("Rolling back transaction due to error or panic.");
    });

    // perform_operations();

    // If we get here, the operation was successful.
    println!("Committing transaction.");
}

```

```
    guard.disarm(); // Don't run the rollback closure.  
}
```

RAII benefits: - Impossible to forget cleanup - Exception-safe (panic-safe in Rust) - Scope-based reasoning about resources - Composable (guards can be nested)

Common guard patterns: - File handles (automatic close) - Locks (automatic release) - Transactions (automatic rollback) - Metrics/timers (automatic reporting) - State flags (automatic reset)

Pattern 5: Memory Layout Optimization

Problem: Naive struct definitions waste memory through padding and hurt performance via poor cache utilization. False sharing in multi-threaded code can cause 10-100x slowdowns.

Solution: Use `#[repr(C)]` for predictable layout (FFI), `#[repr(align(N))]` for cache alignment, `#[repr(packed)]` to eliminate padding (with care). Order struct fields from largest to smallest alignment.

Why It Matters: Modern CPUs are dominated by memory hierarchy—cache misses cost 100-200 cycles while arithmetic costs 1-4 cycles. A cache miss is 50-100x slower than a cache hit.

Use Cases: High-frequency trading systems, game engines, scientific computing, embedded systems, FFI with C libraries, SIMD optimization, lock-free data structures.

What is Alignment? CPUs do not read memory one byte at a time. They fetch it in chunks, typically the size of a machine word (e.g., 8 bytes on a 64-bit system). Access is fastest when a data type of size N is located at a memory address that is a multiple of N. For example, a `u64` (8 bytes) should ideally start at an address like 0, 8, 16, etc. This is its **alignment requirement**. Accessing a `u64` at an unaligned address (e.g., address 1) would be slow, as the CPU might need to perform two memory reads instead of one.

What is Padding? To satisfy these alignment requirements, the Rust compiler may insert invisible, unused bytes into a struct. This is called **padding**. The goal is to ensure every field is properly aligned.

There are two rules for a struct's layout: 1. Each field must be placed at an offset that is a multiple of its alignment. 2. The total size of the struct must be a multiple of the struct's overall alignment, which is the largest alignment of any of its fields.

Examples

Example: Field Ordering to Minimize Padding

By default, Rust reorders struct fields to minimize padding, but with `#[repr(C)]` the order is fixed. Understanding the rules helps in all cases. By ordering fields from largest to smallest, you can minimize wasted space.

```
// In this example, we use `#[repr(C)]` to disable the automatic field  
// reordering that Rust would normally perform. This lets us see the  
// effects of padding manually.
```

```

// Bad: 24 bytes due to padding
#[repr(C)]
struct Unoptimized {
    a: u8,
    b: u64,
    c: u8,
}
// How the compiler lays this out:
// - `a: u8` (size 1, align 1): offset 0.
// - 7 bytes of padding are added to align `b`.
// - `b: u64` (size 8, align 8): offset 8.
// - `c: u8` (size 1, align 1): offset 16.
// - 7 bytes of padding are added at the end to make the total size a multiple of 8.
// - Total size = 24 bytes.

// Good: 16 bytes by reordering fields
#[repr(C)]
struct Optimized {
    b: u64, // Largest alignment first
    a: u8,
    c: u8,
}
// How this improves things:
// - `b: u64`: offset 0.
// - `a: u8`: offset 8.
// - `c: u8`: offset 9.
// - 6 bytes of padding at the end makes the total size 16.
// - Total size = 16 bytes.

// Verify sizes
const _: () = assert!(std::mem::size_of::<Unoptimized>() == 24);
const _: () = assert!(std::mem::size_of::<Optimized>() == 16);

```

Example: Layout Attributes #[repr(...)]

Rust provides attributes to control memory layout.

- `#[repr(C)]`: Guarantees the same layout as a C struct. Essential for FFI.
- `#[repr(packed)]`: Removes all padding. This can lead to unaligned-access performance penalties or even crashes on some architectures. Use with extreme care.
- `#[repr(align(N))]`: Forces the struct's alignment to be at least `N` bytes.
- `#[repr(u8)]`: Specifies the memory representation for an enum's discriminant.

```

// For FFI compatibility
#[repr(C)]
struct Point {
    x: f64,
    y: f64,
}

// To eliminate padding (use carefully!)
#[repr(packed)]
struct Packed {

```

```

    a: u8,
    b: u32, // `b` may be at an unaligned address
}

// To align to a cache line (e.g., 64 bytes)
#[repr(align(64))]
struct CacheAligned {
    data: [u8; 64],
}

// To define an enum's size
#[repr(u8)]
enum Status {
    Idle = 0,
    Running = 1,
    Failed = 2,
}

```

Example: Preventing False Sharing

False sharing is a silent performance killer in multi-threaded code. It happens when two threads write to different variables that happen to live on the same CPU cache line. The CPU's cache coherency protocol forces the cores to fight over the cache line, serializing execution. The fix is to pad data to ensure contended variables are on different cache lines.

```

use std::sync::atomic::AtomicUsize;

const CACHE_LINE_SIZE: usize = 64;

#[repr(align(CACHE_LINE_SIZE))]
struct Padded<T> {
    value: T,
}

// With this structure, counter1 and counter2 are guaranteed to be on
// different cache lines, preventing false sharing when updated by different threads.
struct SharedCounters {
    counter1: Padded<AtomicUsize>,
    counter2: Padded<AtomicUsize>,
}

```

Example: Optimizing Enum Size

An enum's size is determined by its largest variant. If one variant is huge, the whole enum becomes huge. To fix this, you can **Box** the large variant. This makes the variant a pointer, and the enum's size becomes the size of the pointer plus a tag, which is much smaller.

```

// Bad: Size is over 1024 bytes
enum LargeEnum {
    Small(u8),

```

```

        Big([u8; 1024]),
    }

// Good: Size is the size of a Box (a pointer) + a tag.
enum OptimizedEnum {
    Small(u8),
    Big(Box<[u8; 1024]>),
}

```

Example: Data-Oriented Design (SoA vs. AoS)

For performance-critical loops, memory access patterns are key. "Array of Structs" (AoS) is common but can be bad for cache performance if you only need one field per iteration. "Struct of Arrays" (SoA) organizes the data by field, ensuring that when you iterate over one field, all the data for that field is contiguous in memory.

```

// Bad: Array of Structs (AoS) – poor cache locality for single-field access
struct ParticleAoS {
    position: [f32; 3],
    velocity: [f32; 3],
    mass: f32,
}

fn update_aos(particles: &mut [ParticleAoS]) {
    for p in particles {
        // When accessing p.position, the CPU loads the entire struct (position,
        // velocity, mass) into the cache, even though we don't need the other fields.
        p.position[0] += p.velocity[0];
    }
}

// Good: Struct of Arrays (SoA) – excellent cache locality
struct ParticlesSoA {
    positions_x: Vec<f32>,
    velocities_x: Vec<f32>,
    // ... and so on for other fields
}

impl ParticlesSoA {
    fn update_positions(&mut self) {
        // All the x positions are contiguous in memory. The CPU can prefetch
        // them efficiently, leading to far fewer cache misses.
        for i in 0..self.positions_x.len() {
            self.positions_x[i] += self.velocities_x[i];
        }
    }
}

```

Memory layout principles: - Order struct fields from largest to smallest alignment - Use `#[repr(C)]` when layout matters (FFI, serialization) - Pad to cache lines (64 bytes) to prevent false sharing - Box large enum variants to keep enum size small - Consider SoA over AoS for performance-critical loops

Performance characteristics: - False sharing can degrade performance by 10-100x - Proper alignment enables SIMD operations - Cache line is typically 64 bytes - L1 cache miss: ~4 cycles, L3 miss: ~40 cycles, RAM: ~200 cycles

Pattern 6: Arena Allocation

- **Problem:** Allocating many small objects with `Box::new()` or `Vec::push()` is slow. Each call invokes the system's general-purpose allocator (`malloc`), which involves locking and metadata overhead.
- **Solution:** Use an arena allocator (also called a bump allocator). Pre-allocate a large, contiguous chunk of memory.
- **Why It Matters:** Arena allocation is 10-100x faster than general-purpose allocators for scenarios involving many small objects. For applications like compilers (which create millions of AST nodes) or web servers (which create objects per-request), this can dramatically improve performance by reducing allocation bottlenecks.

Examples

```
//=====
// Pattern: Simple arena allocator
//=====

struct Arena {
    chunks: Vec<Vec<u8>>,
    current: Vec<u8>,
    position: usize,
}

impl Arena {
    fn new() -> Self {
        Arena {
            chunks: Vec::new(),
            current: vec![0; 4096],
            position: 0,
        }
    }

    fn alloc<T>(&mut self, value: T) -> &mut T {
        let size = std::mem::size_of::<T>();
        let align = std::mem::align_of::<T>();

        // Align position
        let padding = (align - (self.position % align)) % align;
        self.position += padding;

        // Check if we need a new chunk
        if self.position + size > self.current.len() {
            let old = std::mem::replace(&mut self.current, vec![0; 4096]);
            self.chunks.push(old);
            self.position = 0;
        }
    }
}
```

```

// Allocate
let ptr = &mut self.current[self.position] as *mut u8 as *mut T;
self.position += size;

unsafe {
    std::ptr::write(ptr, value);
    &mut *ptr
}
}

//=====
// Use case: AST nodes during parsing
//=====

struct AstArena {
    arena: Arena,
}

enum Expr<'a> {
    Number(i64),
    Add(&'a Expr<'a>, &'a Expr<'a>),
    Multiply(&'a Expr<'a>, &'a Expr<'a>),
}

impl AstArena {
    fn new() -> Self {
        AstArena { arena: Arena::new() }
    }

    fn number(&mut self, n: i64) -> &Expr {
        self.arena.alloc(Expr::Number(n))
    }

    fn add<'a>(&'a mut self, left: &'a Expr, right: &'a Expr) -> &'a Expr<'a> {
        self.arena.alloc(Expr::Add(left, right))
    }
}

//=====
// Pattern: Typed arena with better ergonomics
//=====

use typed_arena::Arena as TypedArena;

struct Parser<'ast> {
    arena: &'ast TypedArena<Expr<'ast>>,
}

impl<'ast> Parser<'ast> {
    fn parse_number(&self, n: i64) -> &'ast Expr<'ast> {
        self.arena.alloc(Expr::Number(n))
    }
}

```

```

fn parse_binary(&self, left: &'ast Expr<'ast>, right: &'ast Expr<'ast>)
    -> &'ast Expr<'ast>
{
    self.arena.alloc(Expr::Add(left, right))
}
}

//=====
// Pattern: Arena for temporary string allocations
//=====

struct StringArena {
    arena: TypedArena<String>,
}

impl StringArena {
    fn new() -> Self {
        StringArena { arena: TypedArena::new() }
    }

    fn alloc(&self, s: &str) -> &str {
        let owned = self.arena.alloc(s.to_string());
        owned.as_str()
    }
}

//=====
// Use case: Request-scoped allocations in web server
//=====

struct RequestContext<'arena> {
    arena: &'arena TypedArena<Vec<u8>>,
}

impl<'arena> RequestContext<'arena> {
    fn allocate_buffer(&self, size: usize) -> &'arena mut Vec<u8> {
        self.arena.alloc(vec![0; size])
    }
}

```

When to use arenas: - Compiler frontends (AST, IR nodes) - Request handlers in servers - Graph algorithms with temporary nodes - Game engine frame allocations - Any scenario with bulk deallocation

Performance characteristics: - Allocation: O(1), just increment pointer - Deallocation: O(1), drop entire arena - 10-100x faster than malloc/free for small objects - Better cache locality (allocated objects are contiguous) - Cannot free individual objects (trade-off)

Pattern 7: Custom Smart Pointers

- **Problem:** The standard smart pointers (`Box`, `Rc`, `Arc`) are excellent general-purpose tools, but they have limitations. `Rc/Arc` require a separate heap allocation for their reference counts, and simple vector indices can be invalidated by insertions or removals.

- **Solution:** Build custom smart pointers using `unsafe` Rust primitives like `NonNull<T>`, `PhantomData`, and the `Deref`, `DerefMut`, and `Drop` traits. This allows for patterns like intrusive reference counting (where the count is stored in the object itself) or generational indices (which prevent use-after-free errors with vector-like containers).
- **Why It Matters:** Custom smart pointers unlock performance and memory layout patterns that are impossible with standard types. An intrusive `Rc` can save one allocation per object, which is critical when creating millions of them.

Examples

Example: Intrusive Reference Counting

Standard `Rc` and `Arc` perform two allocations: one for the object, and one for the reference-count block. An *intrusive* counter stores the count inside the object itself, saving an allocation. This is critical when you have millions of small, reference-counted objects. This example shows a simplified intrusive `Rc`.

```
use std::ptr::NonNull;
use std::marker::PhantomData;
use std::cell::Cell;
use std::ops::Deref;

// The data and its refcount live in the same heap allocation.
struct IntrusiveNode<T> {
    refcount: Cell<usize>,
    data: T,
}

struct IntrusiveRc<T> {
    ptr: NonNull<IntrusiveNode<T>>,
    _marker: PhantomData<T>,
}

impl<T> IntrusiveRc<T> {
    fn new(data: T) -> Self {
        let node = Box::new(IntrusiveNode {
            refcount: Cell::new(1),
            data,
        });
        IntrusiveRc {
            ptr: unsafe { NonNull::new_unchecked(Box::into_raw(node)) },
            _marker: PhantomData,
        }
    }
}

impl<T> Clone for IntrusiveRc<T> {
    fn clone(&self) -> Self {
        let node = unsafe { self.ptr.as_ref() };
        let count = node.refcount.get();
        node.refcount.set(count + 1);
        IntrusiveRc { ptr: self.ptr, _marker: PhantomData }
    }
}
```

```

    }

}

impl<T> Drop for IntrusiveRc<T> {
    fn drop(&mut self) {
        unsafe {
            let node = self.ptr.as_ref();
            let count = node.refcount.get();
            if count == 1 {
                // Last reference, so deallocate the whole Box.
                drop(Box::from_raw(self.ptr.as_ptr()));
            } else {
                // Decrement the refcount.
                node.refcount.set(count - 1);
            }
        }
    }
}

impl<T> Deref for IntrusiveRc<T> {
    type Target = T;
    fn deref(&self) -> &T {
        unsafe { &self.ptr.as_ref().data }
    }
}

```

Example: Generational Arena for Stable Handles

When you store objects in a `Vec`, their indices are not stable. If you remove an element from the middle, all subsequent indices change. A **generational arena** solves this. It gives you a stable `Handle` (or ID) for an object. The handle contains both an index and a “generation” number. When an object is removed, its slot is marked free, and its generation is incremented. If old code tries to use a stale handle, the generation numbers won’t match, preventing use-after-free bugs. This is a cornerstone of modern Entity-Component-System (ECS) game engines.

```

#[derive(Clone, Copy, PartialEq, Eq)]
struct Handle {
    index: usize,
    generation: u64,
}

struct Slot<T> {
    value: Option<T>,
    generation: u64,
}

struct GenerationalArena<T> {
    slots: Vec<Slot<T>>,
    free_list: Vec<usize>,
}

```

```

impl<T> GenerationalArena<T> {
    fn new() -> Self {
        GenerationalArena { slots: Vec::new(), free_list: Vec::new() }
    }

    fn insert(&mut self, value: T) -> Handle {
        if let Some(index) = self.free_list.pop() {
            let slot = &mut self.slots[index];
            slot.generation += 1;
            slot.value = Some(value);
            Handle { index, generation: slot.generation }
        } else {
            let index = self.slots.len();
            self.slots.push(Slot { value: Some(value), generation: 0 });
            Handle { index, generation: 0 }
        }
    }

    fn get(&self, handle: Handle) -> Option<&T> {
        self.slots.get(handle.index)
            .filter(|slot| slot.generation == handle.generation)
            .and_then(|slot| slot.value.as_ref())
    }

    fn remove(&mut self, handle: Handle) -> Option<T> {
        if let Some(slot) = self.slots.get_mut(handle.index) {
            if slot.generation == handle.generation {
                self.free_list.push(handle.index);
                slot.generation += 1; // Invalidate existing handles
                return slot.value.take();
            }
        }
        None
    }
}

```

Example: Copy-on-Write Smart Pointer

This custom `Immutable<T>` pointer makes a type immutable by default, but allows for cheap clones. Clones share the same underlying data. Only when `modify` is called does the data get copied, ensuring that modifications don't affect other copies. This is a simplified, custom version of the standard library's `Cow`.

```

use std::rc::Rc;
use std::ops::Deref;

struct Immutable<T: Clone> {
    data: Rc<T>,
}

impl<T: Clone> Immutable<T> {

```

```

fn new(data: T) -> Self {
    Immutable { data: Rc::new(data) }
}

fn modify<F>(&mut self, f: F)
where
    F: FnOnce(&mut T),
{
    // If the data is shared (more than one reference exists)...
    if Rc::strong_count(&self.data) > 1 {
        // ...clone it to create a new, unique copy.
        self.data = Rc::new(*self.data).clone();
    }
    // Now we have the only reference, so we can safely get a mutable one.
    let data_mut = Rc::get_mut(&mut self.data).unwrap();
    f(data_mut);
}

impl<T: Clone> Deref for Immutable<T> {
    type Target = T;
    fn deref(&self) -> &T {
        &self.data
    }
}

impl<T: Clone> Clone for Immutable<T> {
    fn clone(&self) -> Self {
        // Cloning is cheap: it just clones the Rc, incrementing the ref count.
        Immutable {
            data: Rc::clone(&self.data),
        }
    }
}

```

When to build custom smart pointers: - Specialized allocation patterns (pools, arenas) - Intrusive data structures for cache efficiency - Game engines (generational indices) - Systems with unique ownership semantics - Performance-critical code where std overhead matters

Performance Summary

| Pattern | Allocation Cost | Access Cost | Best Use Case |
|------------|-----------------|-----------------|--------------------------------------|
| Box<T> | O(1) heap | O(1) | Heap allocation, trait objects |
| Rc<T> | O(1) heap | O(1) + refcount | Shared ownership, single-threaded |
| Arc<T> | O(1) heap | O(1) + atomic | Shared ownership, multi-threaded |
| Cow<T> | O(0) or O(n) | O(1) | Conditional cloning |
| RefCell<T> | O(0) | O(1) + check | Interior mutability, single-threaded |

| Pattern | Allocation Cost | Access Cost | Best Use Case |
|----------|-----------------|-------------|-------------------------------------|
| Mutex<T> | O(0) | O(lock) | Interior mutability, multi-threaded |
| Arena | O(1) bump | O(1) | Bulk allocation/deallocation |

Common Anti-Patterns

```
// ✗ Holding RefCell borrow across function call
let borrowed = data.borrow();
might_borrow_again(&data); // Runtime panic!

// ✓ Scope borrows tightly
{
    let borrowed = data.borrow();
    use_data(&borrowed);
} // Dropped here
might_borrow_again(&data); // Safe

// ✗ Arc<Mutex<T>> when single-threaded
let shared = Arc::new(Mutex::new(data)); // Unnecessary overhead

// ✓ Use Rc<RefCell<T>> for single-threaded
let shared = Rc::new(RefCell::new(data));

// ✗ Cloning Cow unnecessarily
fn process(s: Cow<str>) -> String {
    s.into_owned() // Always allocates
}

// ✓ Return Cow to defer cloning
fn process(s: &str) -> Cow<str> {
    if needs_modification(s) {
        Cow::Owned(modify(s))
    } else {
        Cow::Borrowed(s)
    }
}

fn needs_modification(_s: &str) -> bool { true }
fn modify(s: &str) -> String { s.to_uppercase() }
```

Struct & Enum Patterns

Rust doesn't just give you `struct` and `enum` as containers for data. This chapter explores struct and enum patterns for type-safe **data modeling**: choosing struct types, newtype wrappers for domain types, zero-sized types for compile-time guarantees, enums for variants, and advanced techniques for memory efficiency and recursion.

Pattern 1: Struct Design Patterns

- **Problem:** It's often unclear when to use a named-field struct, a tuple struct, or a unit struct. Named fields can be verbose for simple types (`Point { x: f64, y: f64 }`), while tuple structs can be ambiguous (`Point(1.0, 2.0)`).
- **Solution:** Use named-field structs for complex data models where clarity is key. Use tuple structs for simple wrappers and the newtype pattern to create distinct types from primitives.
- **Why It Matters:** This choice enhances type safety and code clarity. Named fields are self-documenting.

Example: Named Field Structs

```
#[derive(Debug, Clone)]
struct User {
    id: u64,
    username: String,
    email: String,
    active: bool,
}

impl User {
    fn new(id: u64, username: String, email: String) -> Self {
        Self {
            id,
            username,
            email,
            active: true,
        }
    }

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

// Usage
let user = User::new(1, "alice".to_string(), "alice@example.com".to_string());
println!("User {} is active: {}", user.username, user.active);
```

Why this matters: Named fields provide self-documenting code. When you see `user.email`, the intent is clear. They also allow field reordering without breaking code.

Example: Tuple Structs

Tuple structs are useful when field names would be redundant or when you want to create distinct types:

```
// Coordinates where position matters more than names
struct Point3D(f64, f64, f64);
```

```

// Type-safe wrappers (newtype pattern)
struct Kilometers(f64);
struct Miles(f64);

impl Point3D {
    fn origin() -> Self {
        Point3D(0.0, 0.0, 0.0)
    }

    fn distance_from_origin(&self) -> f64 {
        (self.0.powi(2) + self.1.powi(2) + self.2.powi(2)).sqrt()
    }
}

// Usage
let point = Point3D(3.0, 4.0, 0.0);
println!("Distance: {}", point.distance_from_origin());

// Type safety prevents mixing units
let distance_km = Kilometers(100.0);
let distance_mi = Miles(62.0);
// let total = distance_km.0 + distance_mi.0; // Compiles but semantically wrong!

```

The pattern: Use tuple structs when the structure itself conveys meaning more than field names would. They're particularly powerful for the newtype pattern.

Example: Unit Structs

Unit structs carry no data but can implement traits and provide type-level information:

```

// Marker types for type-level programming
struct Authenticated;
struct Unauthenticated;

// Zero-sized types for phantom data
struct Database<State> {
    connection_string: String,
    _state: std::marker::PhantomData<State>,
}

impl Database<Unauthenticated> {
    fn new(connection_string: String) -> Self {
        Database {
            connection_string,
            _state: std::marker::PhantomData,
        }
    }

    fn authenticate(self, password: &str) -> Result<Database<Authenticated>, String> {
        if password == "secret" {
            Ok(Database {
                connection_string: self.connection_string,

```

```

        _state: std::marker::PhantomData,
    })
} else {
    Err("Invalid password".to_string())
}
}

impl Database<Authenticated> {
    fn query(&self, sql: &str) -> Vec<String> {
        println!("Executing: {}", sql);
        vec!["result1".to_string(), "result2".to_string()]
    }
}

// Usage
let db = Database::new("postgres://localhost".to_string());
// db.query("SELECT *"); // Error! Can't query unauthenticated database
let db = db.authenticate("secret").unwrap();
let results = db.query("SELECT * FROM users"); // Now this works

```

The insight: Unit structs enable compile-time state tracking without runtime overhead. This is the typestate pattern in action.

Pattern 2: Newtype and Wrapper Patterns

- **Problem:** Using raw primitive types like `u64` for different kinds of IDs (`UserId`, `OrderId`) can lead to bugs where they are accidentally mixed up. Primitives can't enforce invariants (e.g., a `String` that must be non-empty) and lack domain-specific meaning.
- **Solution:** Wrap primitive types in a tuple struct (e.g., `struct UserId(u64)`). This creates a new, distinct type that cannot be mixed with other types, even if they wrap the same primitive.
- **Why It Matters:** This pattern provides compile-time type safety at zero runtime cost. It prevents logical errors like passing an `OrderId` to a function expecting a `UserId`.

Example: Newtype

```

use std::fmt;

// Newtype for semantic clarity
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord)]
struct UserId(u64);

#[derive(Debug, Clone, Copy)]
struct OrderId(u64);

// Prevent accidentally mixing IDs
fn get_user(id: UserId) -> User {
    println!("Fetching user {}", id.0);
    // ... fetch user
    unimplemented!()
}

```

```

}

// This won't compile:
// let order_id = OrderId(123);
// get_user(order_id); // Type error!

// Wrapper for adding functionality
struct PositiveInteger(i32);

impl PositiveInteger {
    fn new(value: i32) -> Result<Self, String> {
        if value > 0 {
            Ok(PositiveInteger(value))
        } else {
            Err(format!("{} is not positive", value))
        }
    }

    fn get(&self) -> i32 {
        self.0
    }
}

impl fmt::Display for PositiveInteger {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "{}", self.0)
    }
}

// Usage prevents invalid states
let num = PositiveInteger::new(42).unwrap();
// let invalid = PositiveInteger::new(-5); // Returns Err

```

Why wrappers matter: They encode invariants in the type system. Once you have a `PositiveInteger`, you know it's valid. This eliminates defensive checks throughout your codebase.

Example: Transparent Wrappers with Deref

For ergonomic access to the wrapped type:

```

use std::ops::Deref;

struct Validated<T> {
    value: T,
    validated_at: std::time::Instant,
}

impl<T> Validated<T> {
    fn new(value: T) -> Self {
        Self {
            value,
            validated_at: std::time::Instant::now(),
        }
    }
}

```

```

    }

    fn age(&self) -> std::time::Duration {
        self.validated_at.elapsed()
    }
}

impl<T> Deref for Validated<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &self.value
    }
}

// Usage
let validated_string = Validated::new("hello".to_string());
println!("Length: {}", validated_string.len()); // Deref to String
println!("Age: {:?}", validated_string.age()); // Validated method

```

Pattern 3: Struct Memory and Update Patterns

- **Problem:** Understanding struct update syntax `..other` can lead to confusion about ownership and partial moves. Creating variations of a struct immutably can feel clumsy, and the interaction between `Copy` and non-`Copy` fields during updates is not always intuitive.
- **Solution:** Use the struct update syntax `..other` to create a new struct instance from an old one. Be aware that this will *move* any non-`Copy` fields, making the original struct partially unusable.
- **Why It Matters:** This syntax enables ergonomic, immutable updates. A clear understanding of the move semantics involved prevents surprising compile-time ownership errors.

Note: For compile-time state checking with phantom types and typestate patterns, see **Chapter 4: Pattern 6 (Phantom Types)** and **Chapter 5: Pattern 2 (Typestate Pattern)**.

Example: Struct Update Syntax

The struct update syntax `..` is a convenient way to create a new instance of a struct using the values from another instance. Fields that implement the `Copy` trait are copied, while non-`Copy` fields are moved. Because a move occurs, the original instance can no longer be used. To preserve the original, you must `clone()` it.

```

#[derive(Debug, Clone)]
struct Config {
    host: String,
    port: u16,
    timeout_ms: u64,
}

// Usage with move (original is consumed)

```

```

let config1 = Config {
    host: "localhost".to_string(),
    port: 8080,
    timeout_ms: 5000,
};

let config2 = Config {
    port: 9090,
    ..config1 // `config1.host` is moved, `timeout_ms` is copied.
};
// println!("{}:?}", config1); // ERROR: `host` field was moved.

// Usage with clone (original is preserved)
let config3 = Config {
    host: "localhost".to_string(),
    port: 8080,
    timeout_ms: 5000,
};
let config4 = Config {
    port: 9090,
    ..config3.clone() // Clones the `host` string.
};
println!("Original: {:?}", config3); // OK
println!("New: {:?}", config4);

```

Example: Understanding Partial Moves

You can move specific fields out of a struct. If a field does not implement `Copy` (like `String`), moving it means the original struct can no longer be fully accessed, as it is now “partially moved”. You can still access the remaining `Copy` fields, but you cannot move the struct as a whole.

```

struct Data {
    copyable: i32,      // Implements Copy
    moveable: String,   // Does not implement Copy
}

let data = Data {
    copyable: 42,
    moveable: "hello".to_string(),
};

// Move the non-Copy field out of the struct.
let s = data.moveable;
println!("Moved string: {}", s);

// You can still access the Copy field.
println!("Copyable field: {}", data.copyable);

// But you cannot use the whole struct anymore, as it's partially moved.
// let moved_data = data; // ERROR: use of partially moved value: `data`

```

The pattern: When building fluent APIs or config builders, be mindful of moves. Consider consuming `self` and returning `Self`, or use `&mut self` for in-place updates. For full builder pattern coverage, see [Chapter 5: Builder & API Design](#).

Pattern 4: Enum Design Patterns

- **Problem:** Representing a value that can be one of several related kinds is difficult with structs alone. Using `Option` for optional fields can create invalid states (e.g., a “shipped” order with no shipping address).
- **Solution:** Use an `enum` to define a type that can be one of several variants. Each variant can have its own associated data.
- **Why It Matters:** Enums make impossible states unrepresentable. The compiler’s exhaustive checking for `match` statements prevents bugs from forgotten cases.

Example: Basic Enum with Pattern Matching

```
// Model HTTP responses precisely
enum HttpResponse {
    Ok { body: String, headers: Vec<(String, String)> },
    Created { id: u64, location: String },
    NoContent,
    BadRequest { error: String },
    Unauthorized,
    NotFound,
    ServerError { message: String, details: Option<String> },
}

impl HttpResponse {
    fn status_code(&self) -> u16 {
        match self {
            HttpResponse::Ok { .. } => 200,
            HttpResponse::Created { .. } => 201,
            HttpResponse::NoContent => 204,
            HttpResponse::BadRequest { .. } => 400,
            HttpResponse::Unauthorized => 401,
            HttpResponse::NotFound => 404,
            HttpResponse::ServerError { .. } => 500,
        }
    }

    fn is_success(&self) -> bool {
        matches!(self, HttpResponse::Ok { .. } | HttpResponse::Created { .. } |
            HttpResponse::NoContent)
    }
}

// Usage
fn handle_request(path: &str) -> HttpResponse {
    match path {
        "/users" => HttpResponse::Ok {
            body: "[{\\"id\\": 1}]" .to_string(),
        }
    }
}
```

```

        headers: vec![("Content-Type".to_string(), "application/json".to_string())],
    },
    "/users/create" => HttpResponse::Created {
        id: 123,
        location: "/users/123".to_string(),
    },
    _ => HttpResponse::NotFound,
}
}

```

The power: Each variant carries exactly the data it needs. No null or undefined—if a variant needs an ID, it has one.

Example: Enum State Machines

Enums model state machines with exhaustive matching:

```

enum OrderStatus {
    Pending { items: Vec<String>, customer_id: u64 },
    Processing { order_id: u64, started_at: std::time::Instant },
    Shipped { order_id: u64, tracking_number: String },
    Delivered { order_id: u64, signature: Option<String> },
    Cancelled { order_id: u64, reason: String },
}

impl OrderStatus {
    fn process(self) -> Result<OrderStatus, String> {
        match self {
            OrderStatus::Pending { items, .. } => {
                if items.is_empty() {
                    return Err("Cannot process empty order".to_string());
                }
                Ok(OrderStatus::Processing {
                    order_id: 12345,
                    started_at: std::time::Instant::now(),
                })
            }
            _ => Err("Order is not in pending state".to_string()),
        }
    }

    fn can_cancel(&self) -> bool {
        matches!(self, OrderStatus::Pending { .. } | OrderStatus::Processing { .. })
    }
}

```

Note: For compile-time enforced state machines using types (typestate pattern), see [Chapter 5: Pattern 2 \(Typestate Pattern\)](#).

Pattern 5: Advanced Enum Techniques

- **Problem:** Enums can have issues with memory usage if one variant is much larger than the others. Recursive enums (like a tree where a node contains other nodes) are impossible to define directly.
- **Solution:** Use `Box<T>` to heap-allocate the data for large or recursive variants. This makes the size of the variant a pointer size, not the size of the data itself.
- **Why It Matters:** Boxing variants is crucial for two reasons: it makes recursive enum definitions possible, and it makes enums with large variants memory-efficient, improving cache performance. Implementing methods and conversion traits on enums leads to cleaner, more idiomatic, and more reusable code.

Example: Recursive Enums with Box

```
// Binary tree - recursive enum needs Box to break infinite size
enum Tree<T> {
    Leaf(T),
    Node {
        value: T,
        left: Box<Tree<T>>,
        right: Box<Tree<T>>,
    },
}

impl<T: std::fmt::Debug> Tree<T> {
    fn depth(&self) -> usize {
        match self {
            Tree::Leaf(_) => 1,
            Tree::Node { left, right, .. } => {
                1 + left.depth().max(right.depth())
            }
        }
    }
}

// AST nodes often use Box for recursion
enum Expr {
    Number(i32),
    Add(Box<Expr>, Box<Expr>),
    Mul(Box<Expr>, Box<Expr>),
}

impl Expr {
    fn eval(&self) -> i32 {
        match self {
            Expr::Number(n) => *n,
            Expr::Add(l, r) => l.eval() + r.eval(),
            Expr::Mul(l, r) => l.eval() * r.eval(),
        }
    }
}
```

```
}
```

Example: Memory-Efficient Large Variants

```
// Without Box: enum size = size of largest variant (LargeData)
enum Inefficient {
    Small(u8),
    Large([u8; 1024]), // 1KB - every variant takes this space
}

// With Box: enum size = size of pointer (8 bytes on 64-bit)
enum Efficient {
    Small(u8),
    Large(Box<[u8; 1024]>), // Only allocates when this variant is used
}

fn check_sizes() {
    println!("Inefficient: {} bytes", std::mem::size_of::<Inefficient>());
    println!("Efficient: {} bytes", std::mem::size_of::<Efficient>());
}
```

Pattern 6: Visitor Pattern with Enums

- **Problem:** You have a complex, tree-like data structure, such as an Abstract Syntax Tree (AST). You want to perform various operations on this structure (e.g., pretty-printing, evaluation, type-checking) without cluttering the data structure's definition with all of this logic.
- **Solution:** Define a `Visitor` trait with a `visit` method for each variant of your enum-based data structure. Each operation is then implemented as a separate struct that implements the `Visitor` trait.
- **Why It Matters:** This pattern decouples the logic of an operation from the data structure it operates on. This makes it easy to add new operations (just add a new visitor struct) without modifying the (potentially complex) data structure code.

The visitor pattern in Rust leverages enums for traversing complex structures. It involves three parts: the data structure, the visitor trait, and one or more visitor implementations.

1. The Data Structure (AST)

First, define the enum that represents the tree-like structure. For a simple expression language, this is the Abstract Syntax Tree (AST). Note the use of `Box<Expr>` to handle recursion.

```
// AST for a simple expression language
enum Expr {
    Number(f64),
    Variable(String),
    BinaryOp {
        op: BinOp,
        left: Box<Expr>,
```

```

        right: Box<Expr>,
    },
    UnaryOp {
        op: UnOp,
        expr: Box<Expr>,
    },
}

enum BinOp {
    Add,
    Subtract,
    Multiply,
    Divide,
}

enum UnOp {
    Negate,
    Abs,
}

```

2. The Visitor Trait

Next, define the `ExprVisitor` trait. It has a `visit` method for each variant of the `Expr` enum. The `visit` method on the trait itself handles dispatching to the correct specific method.

```

// AST for a simple expression language
enum Expr {
    Number(f64),
    Variable(String),
    BinaryOp {
        op: BinOp,
        left: Box<Expr>,
        right: Box<Expr>,
    },
    UnaryOp {
        op: UnOp,
        expr: Box<Expr>,
    },
}

enum BinOp {
    Add,
    Subtract,
    Multiply,
    Divide,
}

enum UnOp {
    Negate,
    Abs,
}
// Visitor trait

```

```

trait ExprVisitor {
    type Output;

    fn visit(&mut self, expr: &Expr) -> Self::Output {
        match expr {
            Expr::Number(n) => self.visit_number(*n),
            Expr::Variable(name) => self.visit_variable(name),
            Expr::BinaryOp { op, left, right } => {
                self.visit_binary_op(op, left, right)
            }
            Expr::UnaryOp { op, expr } => {
                self.visit_unary_op(op, expr)
            }
        }
    }

    fn visit_number(&mut self, n: f64) -> Self::Output;
    fn visit_variable(&mut self, name: &str) -> Self::Output;
    fn visit_binary_op(&mut self, op: &BinOp, left: &Expr, right: &Expr) -> Self::Output;
    fn visit_unary_op(&mut self, op: &UnOp, expr: &Expr) -> Self::Output;
}

```

3. Visitor Implementations

Finally, implement the visitors. Each visitor is a separate struct that implements the `ExprVisitor` trait, providing concrete logic for each `visit_*` method. This separates the concerns of pretty-printing and evaluation from the data structure itself.

```

# // AST for a simple expression language
# enum Expr {
#     Number(f64),
#     Variable(String),
#     BinaryOp {
#         op: BinOp,
#         left: Box<Expr>,
#         right: Box<Expr>,
#     },
#     UnaryOp {
#         op: UnOp,
#         expr: Box<Expr>,
#     },
# }
#
# enum BinOp {
#     Add,
#     Subtract,
#     Multiply,
#     Divide,
# }
#
# enum UnOp {

```

```

#     Negate,
#     Abs,
# }
#
# // Visitor trait
# trait ExprVisitor {
#     type Output;
#
#     fn visit(&mut self, expr: &Expr) -> Self::Output {
#         match expr {
#             Expr::Number(n) => self.visit_number(*n),
#             Expr::Variable(name) => self.visit_variable(name),
#             Expr::BinaryOp { op, left, right } => {
#                 self.visit_binary_op(op, left, right)
#             }
#             Expr::UnaryOp { op, expr } => {
#                 self.visit_unary_op(op, expr)
#             }
#         }
#     }
#
#     fn visit_number(&mut self, n: f64) -> Self::Output;
#     fn visit_variable(&mut self, name: &str) -> Self::Output;
#     fn visit_binary_op(&mut self, op: &BinOp, left: &Expr, right: &Expr) -> Self::Output;
#     fn visit_unary_op(&mut self, op: &UnOp, expr: &Expr) -> Self::Output;
# }

// Pretty printer visitor
struct PrettyPrinter;

impl ExprVisitor for PrettyPrinter {
    type Output = String;

    fn visit_number(&mut self, n: f64) -> String { n.to_string() }
    fn visit_variable(&mut self, name: &str) -> String { name.to_string() }

    fn visit_binary_op(&mut self, op: &BinOp, left: &Expr, right: &Expr) -> String {
        let op_str = match op {
            BinOp::Add => "+", BinOp::Subtract => "-",
            BinOp::Multiply => "*", BinOp::Divide => "/",
        };
        format!("({} {} {})", self.visit(left), op_str, self.visit(right))
    }

    fn visit_unary_op(&mut self, op: &UnOp, expr: &Expr) -> String {
        let op_str = match op { UnOp::Negate => "-", UnOp::Abs => "abs" };
        format!("{}({})", op_str, self.visit(expr))
    }
}

// Evaluator visitor
struct Evaluator {
    variables: std::collections::HashMap<String, f64>,
}

```

```

}

impl ExprVisitor for Evaluator {
    type Output = Result<f64, String>;

    fn visit_number(&mut self, n: f64) -> Self::Output { Ok(n) }

    fn visit_variable(&mut self, name: &str) -> Self::Output {
        self.variables.get(name).copied().ok_or_else(|| format!("Undefined variable: {}", name))
    }

    fn visit_binary_op(&mut self, op: &BinOp, left: &Expr, right: &Expr) -> Self::Output {
        let left_val = self.visit(left)?;
        let right_val = self.visit(right)?;
        match op {
            BinOp::Add => Ok(left_val + right_val),
            BinOp::Subtract => Ok(left_val - right_val),
            BinOp::Multiply => Ok(left_val * right_val),
            BinOp::Divide => Ok(left_val / right_val),
        }
    }

    fn visit_unary_op(&mut self, op: &UnOp, expr: &Expr) -> Self::Output {
        let val = self.visit(expr)?;
        match op {
            UnOp::Negate => Ok(-val),
            UnOp::Abs => Ok(val.abs()),
        }
    }
}

```

The pattern: Visitors separate traversal logic from data structure. You can add new operations without modifying the enum definition.

Summary

This chapter covered struct and enum patterns for type-safe data modeling:

1. **Struct Design Patterns:** Named fields for clarity, tuple for newtypes/position, unit for markers
2. **Newtype and Wrapper Patterns:** Domain IDs, validated types, invariant enforcement, orphan rule workaround
3. **Struct Memory and Update Patterns:** Struct update syntax, partial moves, builder-style transformations
4. **Enum Design Patterns:** Variants for related types, exhaustive matching, state machines, error types
5. **Advanced Enum Techniques:** Box for large/recursive variants, methods on enums, memory optimization
6. **Visitor Pattern:** Separating traversal logic from data structure with enums

Key Takeaways: - Struct choice is semantic: named for data models, tuple for wrappers, unit for markers - Newtype pattern: UserId(u64) vs OrderId(u64) prevents mixing at zero cost - Enums enforce

exhaustiveness: adding variant causes compile errors in incomplete matches - Box breaks infinite size for recursive enums and reduces memory for large variants

Design Principles: - Use named fields when clarity matters, tuple when type itself is meaningful - Wrap primitives in domain types (UserId not u64) for type safety - Encode invariants in types (PositiveInteger guaranteed positive) - Enums for “one of” types, structs for “all of” types - Box large/recursive enum variants for memory efficiency

Performance Characteristics: - Newtype: zero runtime cost, same representation as wrapped type - Enum size: largest variant + discriminant (usually 1 byte) - Boxing: reduces enum to pointer size, adds indirection

Memory Layout: - Named struct: fields in declaration order (subject to alignment) - Tuple struct: same as tuple with same types - Unit struct: 0 bytes - Enum: size_of(largest variant) + discriminant - Box: size_of pointer (8 bytes on 64-bit)

Pattern Decision Matrix: - **Multiple types, all fields present:** Named struct - **Simple wrapper, distinct type:** Tuple struct (newtype) - **No data, marker only:** Unit struct - **One of several types:** Enum - **Recursive structure:** Enum with Box - **Validated type:** Newtype with smart constructor - **Domain-specific ID:** Newtype (struct UserId(u64))

Anti-Patterns to Avoid: - Using u64 for IDs instead of newtypes (loses type safety) - Multiple Option fields instead of enum (unclear which combinations valid) - Large enum variants without Box (wastes memory) - Missing exhaustive match (non-exhaustive pattern use `_`) - Type aliases for distinct types (`type UserId = u64` doesn't prevent mixing)

Trait Design Patterns

This chapter explores advanced trait patterns: inheritance and bounds for capabilities, associated types vs generics for API design, trait objects for dynamic dispatch, extension traits for extending external types, and sealed traits for controlled implementation.

Pattern 1: Trait Inheritance and Bounds

- **Problem:** Expressing complex capability requirements is unclear—a trait needs `Display` but can't require it directly. Combining multiple capabilities is verbose (`T: Clone + Debug + Display`).
- **Solution:** Use supertrait relationships (`trait Loggable: Debug`) to express requirements. Use trait bounds in generics (`fn process<T: Clone>`), and `where` clauses for readability.
- **Why It Matters:** Supertraits create clear capability requirements. Trait bounds allow for powerful composition of abstractions from simple components.

Example: Super Traits

```
//=====
// Supertrait relationship: Printable requires Debug
//=====

trait Printable: std::fmt::Debug {
    fn print(&self) {
```

```

        println!("{:?}", self);
    }

//=====
// Any type implementing Printable must also implement Debug
//=====

#[derive(Debug)]
struct Document {
    title: String,
    content: String,
}

impl Printable for Document {}

fn example() {
    let doc = Document {
        title: "Rust Guide".to_string(),
        content: "...".to_string(),
    };
    doc.print(); // Uses Debug implementation
}

```

The supertrait relationship expresses a requirement: “To be Printable, you must first be Debug.” This is similar to inheritance in object-oriented languages, but more flexible.

Example: Multiple Supertraits

Traits can require multiple supertraits, combining different capabilities:

```

use std::fmt::{Debug, Display};

//=====
// Requires both Debug and Display
//=====

trait Loggable: Debug + Display {
    fn log(&self) {
        println!("[DEBUG] {:?}", self);
        println!("[INFO] {}", self);
    }
}

#[derive(Debug)]
struct User {
    name: String,
    id: u32,
}

impl Display for User {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "User {} (ID: {})", self.name, self.id)
    }
}

```

```
}
```

```
impl Loggable for User {}
```

```
fn use_loggable<T: Loggable>(item: &T) {
```

```
    item.log();
```

```
}
```

This pattern is useful when your abstraction needs multiple orthogonal capabilities. The `Loggable` trait doesn't need to know *how* to debug or display items—it just requires that the capability exists.

Example: Trait Bounds in Generic Functions

Trait bounds specify what capabilities a generic type must have:

```
//=====
// Simple bound
//=====
fn print_item<T: std::fmt::Display>(item: T) {
    println!("{}", item);
}

//=====
// Multiple bounds
//=====
fn process<T: Clone + std::fmt::Debug>(item: T) {
    let copy = item.clone();
    println!("Processing: {:?}", copy);
}

//=====
// Where clause for readability
//=====
fn complex_function<T, U>(t: T, u: U) -> String
where
    T: std::fmt::Debug + Clone,
    U: std::fmt::Display + Default,
{
    format!(":{} and {}", t, u)
}
```

The `where` clause improves readability when you have many bounds or complex constraints. It's especially useful in traits and `impl` blocks:

```
trait DataProcessor {
    fn process<T>(&self, data: T) -> String
    where
        T: serde::Serialize + std::fmt::Debug;
}
```

Example: Conditional Implementation with Trait Bounds

You can implement traits conditionally based on what traits the type parameters implement:

```
struct Wrapper<T>(T);

//=====
// Only implement Clone if T is Clone
//=====

impl<T: Clone> Clone for Wrapper<T> {
    fn clone(&self) -> Self {
        Wrapper(self.0.clone())
    }
}

//=====
// Only implement Debug if T is Debug
//=====

impl<T: std::fmt::Debug> std::fmt::Debug for Wrapper<T> {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Wrapper({:?}", self.0)
    }
}
```

This pattern allows `Wrapper<String>` to be `Clone` and `Debug`, while `Wrapper<Rc<RefCell<i32>>>` is only `Clone` (because `RefCell` isn't `Debug` in a useful way). The compiler automatically determines which implementations apply.

Example: Trait Bound Patterns

Several common patterns emerge when working with trait bounds:

```
//=====
// Builder pattern with trait bounds
//=====

struct Query<T> {
    data: T,
}

impl<T> Query<T> {
    fn new(data: T) -> Self {
        Query { data }
    }
}

impl<T: Clone> Query<T> {
    // Only available if T is Clone
    fn duplicate(&self) -> Self {
        Query {
            data: self.data.clone(),
        }
    }
}
```

```

    }
}

impl<T: serde::Serialize> Query<T> {
    // Only available if T is Serialize
    fn to_json(&self) -> Result<String, serde_json::Error> {
        serde_json::to_string(&self.data)
    }
}

//=====
// Higher-rank trait bounds (for all lifetimes)
//=====

fn process_with_lifetime<F>(f: F)
where
    F: for<'a> Fn(&'a str) -> &'a str,
{
    let result = f("hello");
    println!("{}", result);
}

```

The builder pattern becomes particularly powerful with conditional trait implementations, as methods only appear when the type parameter supports them.

Pattern 2: Associated Types vs Generics

- **Problem:** A generic trait like `Parser<Output>` allows a single type to have multiple implementations (e.g., for different `Output` types), which can be confusing. Call sites become verbose (`parser.parse::<serde_json::Value>()`), and it's unclear if a type parameter is an "input" or an "output".
- **Solution:** Use **associated types** when an implementing type determines a single, specific "output" type (`trait Parser { type Output; }`). Use **generics** when the caller chooses an "input" type and multiple implementations are desirable (`trait From<T>`).
- **Why It Matters:** Associated types lead to more ergonomic APIs, as the compiler can infer the output type (`parser.parse()` is clean). This prevents ambiguity and simplifies trait bounds.

Example: Generics

```

//=====
// With generics: Multiple implementations possible
//=====

trait Parser<Output> {
    fn parse(&self, input: &str) -> Result<Output, String>;
}

struct JsonParser;

impl Parser<serde_json::Value> for JsonParser {
    fn parse(&self, input: &str) -> Result<serde_json::Value, String> {
        serde_json::from_str(input).map_err(|e| e.to_string())
    }
}

```

```
}

//=====
// Could also implement Parser<MyCustomType> for JsonParser
//=====
```

With generics, a single type can implement the trait multiple times with different type parameters. Sometimes this is exactly what you want, but often it's confusing.

Example: Associated Types: One Implementation

Associated types express "there is one specific type for this implementation":

```
//=====
// With associated types: Only one implementation possible
//=====

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

struct JsonParser;

impl Parser for JsonParser {
    type Output = serde_json::Value;

    fn parse(&self, input: &str) -> Result<Self::Output, String> {
        serde_json::from_str(input).map_err(|e| e.to_string())
    }
}

//=====
// Cannot implement Parser again for JsonParser with different Output
//=====
```

Now `JsonParser` has exactly one `Output` type. Users don't need to specify it—the compiler infers it.

Example: Ergonomics: Associated Types Win for Consumers

Associated types lead to cleaner call sites:

```
//=====
// With generic parameter
//=====

fn use_generic_parser<T, P: Parser<T>>(parser: P, input: &str) -> T {
    parser.parse(input).unwrap()
}

//=====
// Caller must specify T
```

```

//=====
let value: serde_json::Value = use_generic_parser::<serde_json::Value, _>(JsonParser, "{}");

//=====
// With associated type
//=====

fn use_associated_parser<P: Parser>(parser: P, input: &str) -> P::Output {
    parser.parse(input).unwrap()
}

//=====
// Compiler infers Output
//=====

let value = use_associated_parser(JsonParser, "{}");

```

The associated type version is cleaner because the output type is determined by the parser, not by the caller.

Example: When to Use Each

Use generics when: - A type might implement the trait multiple times with different type parameters - The type parameter is an input to the behavior - You want flexibility at the call site

```

//=====
// Generic: Different conversions possible
//=====

trait From<T> {
    fn from(value: T) -> Self;
}

//=====
// String can be created from &str, String, Vec<u8>, etc.
//=====

impl From<&str> for String { /* ... */ }
impl From<Vec<u8>> for String { /* ... */ }

```

Use associated types when: - Only one implementation makes sense for a given type - The associated type is an output of the behavior - You want simpler API for consumers

```

//=====
// Associated type: One iterator type per collection
//=====

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

//=====
// Vec<i32>'s iterator produces i32, not anything else
//=====

```

Example: Combining Both

Sometimes you want both generics and associated types:

```
trait Converter<Input> {
    type Output;
    type Error;

    fn convert(&self, input: Input) -> Result<Self::Output, Self::Error>;
}

struct TemperatureConverter;

impl Converter<f64> for TemperatureConverter {
    type Output = f64;
    type Error = String;

    fn convert(&self, celsius: f64) -> Result<f64, String> {
        Ok(celsius * 9.0 / 5.0 + 32.0)
    }
}

//=====
// Could also implement Converter<i32> with different Output/Error types
//=====
```

This pattern gives you flexibility where you need it (the input type can vary) while maintaining clarity where you don't (each `Converter<Input>` implementation has one output type).

Example: Associated Types with Bounds

Associated types can have trait bounds:

```
trait Graph {
    type Node: std::fmt::Display;
    type Edge: Clone;

    fn nodes(&self) -> Vec<Self::Node>;
    fn edges(&self) -> Vec<Self::Edge>;
}

//=====
// Implementation must satisfy the bounds
//=====

struct SimpleGraph;

impl Graph for SimpleGraph {
    type Node = String; // String implements Display ✓
    type Edge = (usize, usize); // Tuple implements Clone ✓

    fn nodes(&self) -> Vec<String> {
```

```

        vec!["A".to_string(), "B".to_string()]
    }

    fn edges(&self) -> Vec<usize, usize> {
        vec![(0, 1)]
    }
}

```

This pattern ensures that associated types have the capabilities you need to use them correctly.

Example: The Iterator Pattern Deep Dive

`Iterator` is the canonical example of associated types done right:

```

pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;

    // Many provided methods using Self::Item
    fn count(self) -> usize where Self: Sized { /* ... */ }
    fn map<B, F>(self, f: F) -> Map<Self, F>
    where
        Self: Sized,
        F: FnMut(Self::Item) -> B,
    { /* ... */ }
}

```

Why associated type instead of generic? 1. Each iterator produces one type of item 2. The item type is determined by the collection, not chosen by the caller 3. Simpler APIs: `Vec<i32>::iter()` returns iterator of `&i32`, not iterator of some generic `T`

Pattern 3: Trait Objects and Dynamic Dispatch

- **Problem:** Static dispatch via generics (`fn foo<T: Trait>`) creates a copy of the function for each concrete type, leading to code bloat. It's also impossible to create a collection of different types that share a behavior, like `Vec<[Circle, Rectangle]>`.
- **Solution:** Use trait objects (`&dyn Trait`) for dynamic dispatch. This creates a single version of the function that accepts any type implementing the trait, looking up the correct method at runtime via a vtable.
- **Why It Matters:** Dynamic dispatch results in smaller binary sizes and allows for runtime polymorphism (e.g., plugin systems). This flexibility comes at the small cost of a vtable pointer lookup for each method call, which prevents inlining.

Example: static dispatch

```

fn process<T: Display>(item: T) {
    println!("{}: {}", item);
}

```

```
// Compiler generates:  
// fn process_i32(item: i32) { println!("{}", item); }  
// fn process_String(item: String) { println!("{}", item); }
```

Each call site gets optimized code for that specific type. Fast, but increases binary size (code bloat).

Example: Dynamic dispatch (trait objects):

```
fn process(item: &dyn Display) {  
    println!("{}", item);  
}  
  
// Single function generated  
// Uses vtable lookup to find the right Display implementation at runtime
```

One function handles all types. Smaller binary, but slight runtime cost for the vtable lookup.

Example: Creating Trait Objects

Trait objects must be behind a pointer (reference, `Box`, `Rc`, `Arc`):

```
trait Drawable {  
    fn draw(&self);  
}  
  
struct Circle {  
    radius: f64,  
}  
  
impl Drawable for Circle {  
    fn draw(&self) {  
        println!("Drawing circle with radius {}", self.radius);  
    }  
}  
  
struct Rectangle {  
    width: f64,  
    height: f64,  
}  
  
impl Drawable for Rectangle {  
    fn draw(&self) {  
        println!("Drawing rectangle {}x{}", self.width, self.height);  
    }  
}  
  
fn draw_all(shapes: &[Box<dyn Drawable>]) {  
    for shape in shapes {  
        shape.draw();  
    }  
}
```

```

}

fn example() {
    let shapes: Vec<Box<dyn Drawable>> = vec![
        Box::new(Circle { radius: 5.0 }),
        Box::new(Rectangle { width: 10.0, height: 20.0 }),
    ];

    draw_all(&shapes);
}

```

This pattern is powerful: `shapes` can contain different types, all treated uniformly through the `Drawable` interface.

Example: Object Safety Requirements

Not all traits can be used as trait objects. A trait is “object safe” if:

1. **No generic methods:** Methods cannot have type parameters

```

trait NotObjectSafe {
    fn generic_method<T>(&self, value: T); // ✗ Generic method
}

//=====
// Cannot create &dyn NotObjectSafe
//=====
```

1. **No Self: Sized bound:** The trait can't require `Self` to be sized

```

trait NotObjectSafe {
    fn returns_self(self) -> Self; // ✗ Takes self by value, requires Sized
}
```

1. **No associated functions:** Methods must have a `self` receiver

```

trait NotObjectSafe {
    fn new() -> Self; // ✗ No self parameter
}
```

The reasoning: when calling a method on a trait object, the compiler doesn't know the concrete type. Generic methods and associated functions need to know the type at compile time.

Example: Making Traits Object-Safe

You can make traits object-safe with careful design:

```

//=====
// Not object-safe
//=====
```

```

trait Repository {
    fn create<T: Serialize>(&self, item: T) -> Result<(), Error>;
}

//=====
// Object-safe version
//=====

trait Repository {
    fn create(&self, item: &dyn Serialize) -> Result<(), Error>;
}

//=====
// Or split into two traits
//=====

trait Repository {
    fn create(&self, item: Box<dyn Item>) -> Result<(), Error>;
}

trait Item: Serialize {
    // Specific item methods
}

```

This pattern—accepting trait objects instead of generics—makes the trait object-safe while maintaining flexibility.

Example: Downcasting Trait Objects

Sometimes you need to convert a trait object back to a concrete type:

```

use std::any::Any;

trait Shape: Any {
    fn area(&self) -> f64;

    // Provided method for downcasting
    fn as_any(&self) -> &dyn Any {
        self
    }
}

struct Circle {
    radius: f64,
}

impl Shape for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * self.radius * self.radius
    }
}

fn try_as_circle(shape: &dyn Shape) -> Option<&Circle> {
    shape.as_any().downcast_ref::<Circle>()
}

```

```

}

fn example() {
    let circle = Circle { radius: 5.0 };
    let shape: &dyn Shape = &circle;

    if let Some(circle) = try_as_circle(shape) {
        println!("It's a circle with radius {}", circle.radius);
    }
}

```

This pattern is useful but breaks abstraction—use it sparingly, only when you truly need concrete type information.

Example: Trait Objects with Lifetime Bounds

Trait objects can have lifetime bounds:

```

trait Processor {
    fn process(&self, data: &str) -> String;
}

//=====
// Trait object with lifetime
//=====

fn process_data<'a>(processor: &'a dyn Processor, data: &'a str) -> String {
    processor.process(data)
}

//=====
// Boxed trait object with lifetime
//=====

struct Handler<'a> {
    processor: Box<dyn Processor + 'a>,
}

```

The `+ 'a` syntax means “the trait object must live at least as long as `'a`”. This ensures references in the trait implementation remain valid.

Example: Costs of Dynamic Dispatch

Dynamic dispatch has small but real costs:

```

//=====
// Static dispatch
//=====

fn static_dispatch<T: Display>(items: &[T]) {
    for item in items {
        println!("{}", item); // Direct function call, inlinable
    }
}

```

```

//=====
// Dynamic dispatch
//=====

fn dynamic_dispatch(items: &[&dyn Display]) {
    for item in items {
        println!("{}", item); // Indirect call through vtable
    }
}

```

Benchmarking typical code shows:

- Static dispatch: ~1-2ns per call
- Dynamic dispatch: ~3-5ns per call

The difference is tiny for I/O-bound operations but can matter for tight inner loops. Profile before optimizing.

Pattern 4: Extension Traits

- **Problem:** You can't add methods to types from other crates (the "orphan rule"). You want to extend standard types like `Vec` or `String` with domain-specific helpers, but can't modify their source code.
- **Solution:** Define a new trait (an "extension trait") with the desired methods. Then, implement that trait for the external type.
- **Why It Matters:** This pattern allows you to extend any type you don't own in a modular, opt-in way. It solves the orphan rule problem cleanly.

Example: Basic Extension Trait

The orphan rule prevents implementing a foreign trait on a foreign type. However, you can implement your *own* trait on a foreign type. This is the core of the extension trait pattern. Here, we can't add a `sum` method to `Vec` directly, but we can define our own `SumExt` trait and implement it for `Vec<i32>`.

```

trait SumExt {
    fn sum_ext(&self) -> i32;
}

impl SumExt for Vec<i32> {
    fn sum_ext(&self) -> i32 {
        self.iter().sum()
    }
}

fn main() {
    let numbers = vec![1, 2, 3, 4, 5];
    // By bringing `SumExt` into scope, we "extend" Vec<i32>
    println!("Sum: {}", numbers.sum_ext());
}

```

Example: Blanket Iterator Extensions

This pattern is extremely powerful. By defining an extension trait `IteratorExt` and providing a blanket implementation for *all* types that implement `Iterator`, we can add new functionality to every iterator in our program.

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

trait IteratorExt: Iterator {
    // Count occurrences of each item in an iterator.
    fn counts(self) -> HashMap<Self::Item, usize>
    where
        Self: Sized,
        Self::Item: Eq + std::hash::Hash,
    {
        let mut map = HashMap::new();
        for item in self {
            *map.entry(item).or_insert(0) += 1;
        }
        map
    }
}

// Blanket implementation: this applies to any type that is an Iterator.
impl<I: Iterator> IteratorExt for I {}
```

fn main() {
 let words = vec!["apple", "banana", "apple", "cherry", "banana", "apple"];
 let counts = words.into_iter().counts();
 println!("{}:?", counts); // {"apple": 3, "banana": 2, "cherry": 1}
}

Example: Ergonomic Error Handling

Extension traits can make error handling more ergonomic by adding context or logging capabilities to the standard `Result` type. This `ResultExt` provides a `log_err` method that logs the error and its context before passing it up the call stack.

```
trait ResultExt<T> {
    fn log_err(self, context: &str) -> Self;
}

impl<T, E: std::error::Error> ResultExt<T> for Result<T, E> {
    fn log_err(self, context: &str) -> Self {
        self.map_err(|e| {
            eprintln!("[ERROR] {}: {}", context, e);
            e
        })
    }
}
```

```
fn main() {
    let _ = std::fs::read_to_string("non_existent_file.txt")
        .log_err("Failed to read config");
}
```

Example: Extending Standard Types

You can add domain-specific helper methods to standard library types like `String` and `str` to make your code more expressive.

```
trait StringExt {
    fn truncate_to(&self, max_len: usize) -> String;
}

impl<T: AsRef<str>> StringExt for T {
    fn truncate_to(&self, max_len: usize) -> String {
        let s = self.as_ref();
        if s.len() <= max_len {
            s.to_string()
        } else {
            format!("{}...", &s[..max_len.saturating_sub(3)])
        }
    }
}

fn main() {
    let long_string = "This is a very long string that needs truncation".to_string();
    let truncated = long_string.truncate_to(20);
    println!("{}", truncated); // "This is a very lo..."
}
```

Example: Conditional Extensions

An extension can be made conditional on the capabilities of the type it's extending. This `DebugExt` trait is implemented for any type `T` as long as `T` implements `Debug`, giving all debuggable types a handy `debug_print` method.

```
trait DebugExt {
    fn debug_print(&self);
}

impl<T: std::fmt::Debug> DebugExt for T {
    fn debug_print(&self) {
        println!("{:?}", self);
    }
}

fn main() {
    let numbers = vec![1, 2, 3];
    numbers.debug_print(); // Works because Vec<i32> implements Debug.
}
```

```
let name = "Alice";
name.debug_print(); // Works because &str implements Debug.
}
```

This pattern is incredibly powerful—one implementation provides functionality to infinite types.

Pattern 5: Sealed Traits

- **Problem:** As a library author, you want to publish a trait that users can depend on, but you want to prevent them from implementing it themselves. This allows you to add new methods to the trait later without it being a breaking change.
- **Solution:** Create a private `sealed` module with a public but un-implementable `Sealed` trait. Make your public trait a supertrait of `sealed::Sealed`.
- **Why It Matters:** This pattern gives you the freedom to evolve your API (e.g., add methods with default implementations to the trait) without breaking downstream users. It's a crucial tool for library authors who need to maintain long-term stability.

Example: Basic Sealed Trait

```
mod sealed {
    pub trait Sealed {}
}

pub trait MyTrait: sealed::Sealed {
    fn my_method(&self);

    // Can add new methods without breaking external code
    fn new_method(&self) {
        println!("Default implementation");
    }
}

struct MyType;

impl sealed::Sealed for MyType {}
impl MyTrait for MyType {
    fn my_method(&self) {
        println!("Internal implementation");
    }
}

// External crates can USE MyTrait but cannot IMPLEMENT it
// because they can't access sealed::Sealed
```

This pattern ensures you can evolve the trait API without semver-major version bumps.

Example: Dependency Injection with Traits

Use traits for testable, flexible architectures:

```

trait Database {
    fn get_user(&self, id: i32) -> Option<User>;
    fn save_user(&self, user: &User) -> Result<(), Error>;
}

trait EmailService {
    fn send_email(&self, to: &str, subject: &str, body: &str) -> Result<(), Error>;
}

struct UserService<D, E> {
    database: D,
    email: E,
}

impl<D: Database, E: EmailService> UserService<D, E> {
    fn new(database: D, email: E) -> Self {
        UserService { database, email }
    }

    fn register_user(&self, name: &str, email: &str) -> Result<User, Error> {
        let user = User {
            id: generate_id(),
            name: name.to_string(),
            email: email.to_string(),
        };

        self.database.save_user(&user)?;
        self.email.send_email(email, "Welcome!", "Thanks for signing up")?;

        Ok(user)
    }
}

// Production uses real implementations
// Tests use mocks

```

This pattern makes code testable and modular.

Summary

Key Takeaways: - Trait inheritance expresses capabilities: “to be A must be B” is declarative and composable - Associated types = one impl per type, inferred; generics = multiple impls, explicit choice - Dynamic dispatch = smaller binary, ~2-3ns overhead; static dispatch = optimized per-type - Extension traits extend types you don’t own via trait + impl - Sealed traits prevent external impls via private supertrait

Design Guidelines: - Supertraits for capability requirements: `trait Loggable: Debug + Display` - Associated types when output determined by type, generics when chosen by caller - Trait objects for heterogeneous collections, generics for performance - Extension traits for opt-in functionality on external types - Sealed traits when evolution/safety requires controlled implementations

Object Safety Rules (for `&dyn Trait`): - No generic methods (needs concrete type at compile-time) -
No Self: Sized bound (trait objects are !Sized) - Must have `&self`/`&mut self` receiver (needs object to call)
- No associated functions without self (can't call without type)

Common Patterns:

```
// Trait inheritance
trait Loggable: Debug + Display {
    fn log(&self) { println!("{}: {:?}", self); }
}

// Associated type (one impl per type)
trait Parser {
    type Output;
    fn parse(&self, input: &str) -> Self::Output;
}

// Generic (multiple impls possible)
trait From<T> {
    fn from(value: T) -> Self;
}

// Trait object (heterogeneous collection)
let shapes: Vec<Box<dyn Drawable>> = vec![
    Box::new(Circle { radius: 5.0 }),
    Box::new(Rectangle { width: 10.0, height: 20.0 }),
];

// Extension trait
trait StringExt {
    fn truncate_to(&self, max_len: usize) -> String;
}
impl StringExt for String { /* ... */ }

// Sealed trait (prevent external impl)
mod sealed { pub trait Sealed {} }
pub trait MyTrait: sealed::Sealed { /* ... */ }
```

Generics & Polymorphism

Generics are Rust's mechanism for writing code that works across multiple types while maintaining full type safety. Unlike dynamic languages that achieve polymorphism through runtime type checking, Rust's generics are resolved entirely at compile time through **monomorphization**—the compiler generates specialized versions of generic code for each concrete type used.

This approach delivers three critical benefits: 1. **Zero runtime overhead**: Generic code runs as fast as hand-written specialized code 2. **Type safety**: Errors caught at compile time, not runtime 3. **Code reuse**: Write once, use with any type that satisfies constraints

Rust's generics go beyond simple type parameters. Combined with the trait system, they enable: -

Bounded polymorphism: Require types to implement specific capabilities - **Associated types:**

Simplify trait APIs by fixing "output" types - **Const generics:** Parameterize by compile-time values, not just types - **Higher-kinded patterns:** Express relationships between types

Type System Foundation

```
// Basic generic syntax
fn foo<T>(x: T) -> T { x }                                // Generic function
struct Pair<T, U> { first: T, second: U }                      // Generic struct
enum Result<T, E> { Ok(T), Err(E) }                            // Generic enum
impl<T> Pair<T, T> { }                                         // Generic impl

// Trait bounds (constraints)
fn print<T: Display>(x: T) { }                                // Single bound
fn both<T: Debug + Clone>(x: T) { }                           // Multiple bounds
fn complex<T>(x: T) where T: Debug + Clone { }                // Where clause
fn lifetime<'a, T: 'a>(x: &'a T) { }                         // Lifetime bounds

// Associated types
trait Iterator { type Item; }                                     // Type determined by impl
trait Graph { type Node; type Edge; }                            // Multiple associated types

// Const generics
struct Array<T, const N: usize>([T; N]);                     // Parameterized by value
fn fixed<const N: usize>() -> [u8; N] { [0; N] }           // Const in return type

// Higher-ranked trait bounds
fn hrtb<F>(f: F) where F: for<'a> Fn(&'a str) { } // Works for any lifetime

// Phantom types
use std::marker::PhantomData;
struct Tagged<T, Tag> { value: T, _tag: PhantomData<Tag> }
```

Pattern 1: Type-Safe Generic Functions

- **Problem:** You need to write functions that perform the same operation on different types.
Without generics, you'd either duplicate code for each type (error-prone, unmaintainable) or use dynamic typing with runtime casts (unsafe, slow).
- **Solution:** Define functions with type parameters `<T>` that can be instantiated with any concrete type. Add trait bounds to constrain `T` to types that support required operations.
- **Why It Matters:** Monomorphization means generic code is as fast as hand-written specialized code—there's no vtable lookup, no boxing, no dynamic dispatch. The `Vec::push` you call on `Vec<i32>` is different compiled code than `Vec::push` on `Vec<String>`, each optimized for its type.

Examples

```
use std::fmt::Display;
use std::cmp::Ordering;
```

```

//=====
// Pattern: Basic generic function with type inference
//=====

fn identity<T>(x: T) -> T {
    x
}

let num = identity(42);           // T inferred as i32
let text = identity("hello");    // T inferred as &str

//=====
// Pattern: Generic function with trait bound for operations
//=====

fn largest<T: PartialOrd>(list: &[T]) -> Option<&T> {
    let mut largest = list.first()?;
    for item in list {
        if item > largest {
            largest = item;
        }
    }
    Some(largest)
}

let numbers = vec![34, 50, 25, 100, 65];
assert_eq!(largest(&numbers), Some(&100));

let chars = vec!['y', 'm', 'a', 'q'];
assert_eq!(largest(&chars), Some(&'y'));

//=====
// Pattern: Multiple trait bounds for complex requirements
//=====

fn print_sorted<T>(mut items: Vec<T>)
where
    T: Ord + Display,    // Sortable AND printable
{
    items.sort();
    for item in items {
        println!("{}", item);
    }
}

//=====
// Pattern: Generic function returning owned vs borrowed
//=====

// Returns owned value - caller gets ownership
fn create_default<T: Default>() -> T {
    T::default()
}

// Returns reference - borrows from input
fn first<T>(slice: &[T]) -> Option<&T> {

```

```

        slice.first()
    }

// Returns reference with explicit lifetime
fn longest<'a, T>(x: &'a [T], y: &'a [T]) -> &'a [T] {
    if x.len() > y.len() { x } else { y }
}

//=====
// Pattern: Turbofish syntax for explicit type specification
//=====

let parsed = "42".parse::<i32>().unwrap();           // Turbofish ::<i32>
let default = create_default::<String>();           // Explicit type
let collected: Vec<i32> = (0..10).collect();       // Type annotation alternative

//=====
// Pattern: Generic comparison with borrowing
//=====

fn compare<T: Ord>(a: &T, b: &T) -> Ordering {
    a.cmp(b)
}

fn min_ref<'a, T: Ord>(a: &'a T, b: &'a T) -> &'a T {
    if a <= b { a } else { b }
}

//=====
// Pattern: Generic function with default values
//=====

fn get_or_default<T: Default>(opt: Option<T>) -> T {
    opt.unwrap_or_default()
}

fn get_or<T>(opt: Option<T>, default: T) -> T {
    opt.unwrap_or(default)
}

fn get_or_else<T, F: FnOnce() -> T>(opt: Option<T>, f: F) -> T {
    opt.unwrap_or_else(f)
}

```

When to use generic functions: - Operations that apply uniformly across types - When you need zero-cost abstraction - When trait bounds express the required capabilities - When type inference reduces boilerplate

Performance characteristics: - Monomorphization: separate code generated per type - Zero runtime overhead vs hand-written specialized code - Increased compile time and binary size (trade-off) - Inlining opportunities for small generic functions

Pattern 2: Generic Structs and Enums

- **Problem:** Data structures need to store values of various types. Hard-coding types limits reusability—you'd need `IntStack`, `StringStack`, `FloatStack`, etc.
- **Solution:** Parameterize structs and enums over type parameters. `struct Stack<T>` can hold any type while remaining fully type-safe.
- **Why It Matters:** Generic data structures are the foundation of Rust's standard library. `Vec<T>`, `HashMap<K, V>`, `Option<T>`, `Result<T, E>`—these are all generic types.

Examples

```
//=====
// Pattern: Basic generic struct
//=====

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

impl<T> Point<T> {
    fn new(x: T, y: T) -> Self {
        Point { x, y }
    }
}

// Methods requiring specific traits
impl<T: Copy + std::ops::Add<Output = T>> Point<T> {
    fn add(&self, other: &Point<T>) -> Point<T> {
        Point {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}

let int_point = Point::new(5, 10);
let float_point = Point::new(1.0, 4.0);

//=====
// Pattern: Multiple type parameters
//=====

struct Pair<T, U> {
    first: T,
    second: U,
}

impl<T, U> Pair<T, U> {
    fn new(first: T, second: U) -> Self {
        Pair { first, second }
    }
}
```

```

// Method mixing type parameters
fn swap(self) -> Pair<U, T> {
    Pair {
        first: self.second,
        second: self.first,
    }
}

// Method with different generics than struct
impl<T, U> Pair<T, U> {
    fn mix<V, W>(self, other: Pair<V, W>) -> Pair<T, W> {
        Pair {
            first: self.first,
            second: other.second,
        }
    }
}

//=====
// Pattern: Generic enum with variants
//=====

enum BinaryTree<T> {
    Empty,
    Node {
        value: T,
        left: Box<BinaryTree<T>>,
        right: Box<BinaryTree<T>>,
    },
}

impl<T: Ord> BinaryTree<T> {
    fn new() -> Self {
        BinaryTree::Empty
    }

    fn insert(&mut self, new_value: T) {
        match self {
            BinaryTree::Empty => {
                *self = BinaryTree::Node {
                    value: new_value,
                    left: Box::new(BinaryTree::Empty),
                    right: Box::new(BinaryTree::Empty),
                };
            }
            BinaryTree::Node { value, left, right } => {
                if new_value <= *value {
                    left.insert(new_value);
                } else {
                    right.insert(new_value);
                }
            }
        }
    }
}

```

```

}

fn contains(&self, target: &T) -> bool {
    match self {
        BinaryTree::Empty => false,
        BinaryTree::Node { value, left, right } => {
            match target.cmp(value) {
                std::cmp::Ordering::Equal => true,
                std::cmp::Ordering::Less => left.contains(target),
                std::cmp::Ordering::Greater => right.contains(target),
            }
        }
    }
}

//=====
// Pattern: Generic wrapper with transformation
//=====

struct Wrapper<T> {
    value: T,
}

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

    fn into_inner(self) -> T {
        self.value
    }

    fn map<U, F: FnOnce(T) -> U>(self, f: F) -> Wrapper<U> {
        Wrapper { value: f(self.value) }
    }

    fn as_ref(&self) -> Wrapper<&T> {
        Wrapper { value: &self.value }
    }
}

//=====
// Pattern: Specialized impl for specific types
//=====

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

// Only available for f64
impl Point<f64> {
    fn distance_from_origin(&self) -> f64 {
}

```

```

        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}

// Only available for numeric types
impl<T: std::ops::Mul<Output = T> + Copy> Point<T> {
    fn scale(&self, factor: T) -> Point<T> {
        Point {
            x: self.x * factor,
            y: self.y * factor,
        }
    }
}

//=====
// Pattern: Generic struct with lifetime
//=====

struct Ref<'a, T> {
    value: &'a T,
}

impl<'a, T> Ref<'a, T> {
    fn new(value: &'a T) -> Self {
        Ref { value }
    }
}

struct MutRef<'a, T> {
    value: &'a mut T,
}

//=====
// Pattern: Default bounds for common functionality
//=====

#[derive(Debug, Clone, PartialEq)]
struct Container<T> {
    items: Vec<T>,
}

impl<T> Default for Container<T> {
    fn default() -> Self {
        Container { items: Vec::new() }
    }
}

impl<T> Container<T> {
    fn push(&mut self, item: T) {
        self.items.push(item);
    }

    fn pop(&mut self) -> Option<T> {
        self.items.pop()
    }
}

```

```
}
```

Design guidelines: - Prefer single type parameter when elements are homogeneous - Use multiple parameters for relationships (key-value, input-output) - Add trait bounds at impl level, not struct definition - Consider **Default** implementation for generic containers

Common patterns: - **Wrapper<T>**: Add behavior to any type - **Container<T>**: Store multiple values - **Tree<T>, Graph<T>**: Recursive structures - **Result<T, E>**: Success or error - **State<S, T>**: Type-state machines

Pattern 3: Trait Bounds and Constraints

- **Problem:** Generic functions need to perform operations on their type parameters, but **T** could be any type—how do you call `.clone()` on **T** if **T** might not implement **Clone**? Without constraints, you can only move, drop, or pass the value around.
- **Solution:** Use trait bounds to constrain type parameters to types implementing specific traits. Bounds can be inline (`fn foo<T: Clone>`) or in where clauses (`where T: Clone + Debug`).
- **Why It Matters:** Trait bounds are the contract between generic code and its callers. They specify exactly what capabilities are required, enabling the compiler to verify correctness.

Examples

```
use std::fmt::{Debug, Display};
use std::hash::Hash;
use std::collections::HashMap;

//=====
// Pattern: Single trait bound
//=====
fn print_debug<T: Debug>(value: T) {
    println!("{}: {:?}", value);
}

fn clone_it<T: Clone>(value: &T) -> T {
    value.clone()
}

//=====
// Pattern: Multiple trait bounds
//=====
// Using + syntax
fn compare_and_show<T: PartialOrd + Display>(a: T, b: T) {
    if a < b {
        println!("{} < {}", a, b);
    }
}

// Using where clause (cleaner for complex bounds)
fn complex_operation<T, U>(t: T, u: U)
```

```

where
    T: Clone + Debug + Default,
    U: AsRef<str> + Display,
{
    println!("T: {:?}", U: {}, t.clone(), u);
}

//=====
// Pattern: Bounds on associated types
//=====

fn sum_iterator<I>(iter: I) -> i32
where
    I: Iterator<Item = i32>,
{
    iter.sum()
}

fn print_all<I>(iter: I)
where
    I: Iterator,
    I::Item: Display, // Bound on associated type
{
    for item in iter {
        println!("{}", item);
    }
}

//=====
// Pattern: Lifetime bounds on type params
//=====

// T must live at least as long as 'a
fn store_ref<'a, T: 'a>(storage: &mut Option<&'a T>, value: &'a T) {
    *storage = Some(value);
}

// T must be 'static (no borrows or only 'static borrows)
fn spawn_task<T: Send + 'static>(value: T) {
    std::thread::spawn(move || {
        drop(value);
    });
}

//=====
// Pattern: Conditional method implementation
//=====

struct Wrapper<T>(T);

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

    fn into_inner(self) -> T {
}

```

```

        self.0
    }
}

// Only when T: Display
impl<T: Display> Wrapper<T> {
    fn display(&self) {
        println!("{}", self.0);
    }
}

// Only when T: Clone
impl<T: Clone> Wrapper<T> {
    fn duplicate(&self) -> Self {
        Wrapper(self.0.clone())
    }
}

// Only when T: Default
impl<T: Default> Default for Wrapper<T> {
    fn default() -> Self {
        Wrapper(T::default())
    }
}

//=====
// Pattern: Trait bounds for hashable keys
//=====

fn count_occurrences<T>(items: &[T]) -> HashMap<&T, usize>
where
    T: Hash + Eq,
{
    let mut counts = HashMap::new();
    for item in items {
        *counts.entry(item).or_insert(0) += 1;
    }
    counts
}

//=====
// Pattern: Sized and ?Sized bounds
//=====

// T must be Sized (default, known size at compile time)
fn takes_sized<T>(value: T) {
    drop(value);
}

// T can be unsized (like str, [u8], dyn Trait)
fn takes_unsized<T: ?Sized>(value: &T) {
    // Can only use through reference
    let _ = std::mem::size_of_val(value);
}

```

```

// Works with both String and str
fn print_str<T: AsRef<str> + ?Sized>(s: &T) {
    println!("{}", s.as_ref());
}

//=====
// Pattern: Supertraits (trait inheritance)
//=====

trait Printable: Display + Debug {
    fn print(&self) {
        println!("Display: {}, Debug: {:?}", self, self);
    }
}

// Implementing Printable requires Display + Debug
impl Printable for i32 {}
impl Printable for String {}

//=====
// Pattern: Generic bounds with Self
//=====

trait Duplicable: Sized {
    fn duplicate(&self) -> Self;
}

impl<T: Clone> Duplicable for T {
    fn duplicate(&self) -> Self {
        self.clone()
    }
}

//=====
// Pattern: Combining bounds with impl Trait
//=====

fn make_iterator() -> impl Iterator<Item = i32> {
    (0..10).filter(|x| x % 2 == 0)
}

fn transform<T>(items: impl IntoIterator<Item = T>) -> Vec<T>
where
    T: Clone,
{
    items.into_iter().collect()
}

```

Bound selection guidelines: - Start with minimal bounds, add as compiler requires - Prefer `AsRef<T>` over concrete types for flexibility - Use `?Sized` when working through references only - Consider `Send + Sync + 'static` for thread-spawning

Common bound combinations: - `Clone + Debug`: Development/testing - `Hash + Eq`: HashMap keys - `Ord`: Sortable, BTreeMap keys - `Serialize + DeserializeOwned`: Data serialization - `Send + Sync + 'static`: Thread safety

Pattern 4: Associated Types vs Generic Parameters

- **Problem:** When designing a trait, should you use `trait Foo<T>` (generic parameter) or `trait Foo { type T; }` (associated type)? Both allow types to vary by implementation, but they have different semantics and ergonomics.
- **Solution:** Use **associated types** for “output” types where each implementation specifies exactly one type (`Iterator::Item`). Use **generic parameters** for “input” types where the same type can implement the trait multiple times with different parameters (`From`).
- **Why It Matters:** Associated types simplify APIs dramatically. With `Iterator<Item = i32>`, you know the item type.

Examples

```
use std::ops::Add;

//=====
// Pattern: Associated type - one impl per type
//=====

trait Container {
    type Item; // Associated type

    fn get(&self, index: usize) -> Option<&Self::Item>;
    fn len(&self) -> usize;
}

impl<T> Container for Vec<T> {
    type Item = T; // Specified by impl

    fn get(&self, index: usize) -> Option<&T> {
        <[T]>::get(self, index)
    }

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

// Usage is clean - no extra type parameters
fn first_item<C: Container>(c: &C) -> Option<&C::Item> {
    c.get(0)
}

//=====
// Pattern: Generic parameter - multiple impls per type
//=====

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

// Same type can implement for multiple targets
impl Convertible<String> for i32 {
```

```

fn convert(&self) -> String {
    self.to_string()
}

impl Convertible<f64> for i32 {
    fn convert(&self) -> f64 {
        *self as f64
    }
}

// Must specify which conversion
let n: i32 = 42;
let s: String = Convertible::<String>::convert(&n);
let f: f64 = Convertible::<f64>::convert(&n);

//=====
// Pattern: Combining associated types and generics
//=====

trait Graph {
    type Node;
    type Edge;

    fn nodes(&self) -> Vec<&Self::Node>;
    fn edges(&self) -> Vec<&Self::Edge>;
}

// Generic over graph type, but Node/Edge are associated
fn count_nodes<G: Graph>(graph: &G) -> usize {
    graph.nodes().len()
}

//=====
// Pattern: Associated type with bounds
//=====

trait Summable {
    type Item: Add<Output = Self::Item> + Default + Copy;

    fn items(&self) -> &[Self::Item];

    fn sum(&self) -> Self::Item {
        self.items()
            .iter()
            .copied()
            .fold(Self::Item::default(), |acc, x| acc + x)
    }
}

//=====
// Pattern: Generic Associated Types (GAT)
//=====

trait LendingIterator {
    type Item<'a> where Self: 'a;           // GAT with lifetime
}

```

```

    fn next(&mut self) -> Option<Self::Item'_>;
}

// Allows returning references to self
struct WindowsMut<'a, T> {
    slice: &'a mut [T],
    pos: usize,
}

impl<'a, T> LendingIterator for WindowsMut<'a, T> {
    type Item<'b> = &'b mut [T] where Self: 'b;

    fn next(&mut self) -> Option<Self::Item'_> {
        if self.pos + 2 <= self.slice.len() {
            let window = &mut self.slice[self.pos..self.pos + 2];
            self.pos += 1;
            Some(window)
        } else {
            None
        }
    }
}

//=====
// Pattern: Type families with associated types
//=====

trait Family {
    type Member;
}

struct IntFamily;
impl Family for IntFamily {
    type Member = i32;
}

struct StringFamily;
impl Family for StringFamily {
    type Member = String;
}

fn create_member<F: Family>() -> F::Member
where
    F::Member: Default,
{
    F::Member::default()
}

//=====
// Pattern: Associated type vs generic - API comparison
//=====

// Associated type approach (cleaner)
trait Deref {

```

```

type Target: ?Sized;
fn deref(&self) -> &Self::Target;
}

// If it were generic (worse - ambiguous)
trait DerefGeneric<Target: ?Sized> {
    fn deref(&self) -> &Target;
}

// Usage difference:
// Associated: impl Deref for MyPtr { type Target = i32; ... }
//                 Only ONE target type per pointer type
//
// Generic:     impl DerefGeneric<i32> for MyPtr { ... }
//                 impl DerefGeneric<String> for MyPtr { ... }
//                 Ambiguous! Which deref to call?

```

Decision guide:

Use Associated Types When Use Generic Parameters When

| | |
|-----------------------------|-----------------------------------|
| One implementation per type | Multiple implementations per type |
| "Output" or "result" type | "Input" or "parameter" type |
| User shouldn't choose type | User chooses type at call site |
| Example: Iterator::Item | Example: From, Into |
| Example: Deref::Target | Example: Add |

Key insight: Associated types answer “what type does this produce?” Generic parameters answer “what type should this accept?”

Pattern 5: Blanket Implementations

- **Problem:** You want to provide trait implementations for all types meeting certain criteria, not just specific types. For example, any type implementing `Display` should automatically get `ToString`.
- **Solution:** Use blanket implementations: `impl<T: Bound> Trait for T`. This implements `Trait` for all types `T` that satisfy `Bound`.
- **Why It Matters:** Blanket implementations enable powerful trait composition. You define the relationship once, and it applies to all qualifying types—past, present, and future.

Examples

```

use std::fmt::{Debug, Display};

//=====
// Pattern: Blanket impl for all types implementing a trait
//=====

trait Printable {
    fn print(&self);
}

```

```

// Any type that implements Display gets Printable for free
impl<T: Display> Printable for T {
    fn print(&self) {
        println!("{}", self);
    }
}

// Now i32, String, etc. all have .print()
42.print();
"hello".print();

//=====
// Pattern: Extension trait with blanket impl
//=====

trait IteratorExt: Iterator {
    fn count_where<P>(self, predicate: P) -> usize
    where
        Self: Sized,
        P: FnMut(&Self::Item) -> bool;
}

// Blanket impl for all iterators
impl<I: Iterator> IteratorExt for I {
    fn count_where<P>(self, mut predicate: P) -> usize
    where
        P: FnMut(&Self::Item) -> bool,
    {
        self.filter(|item| predicate(item)).count()
    }
}

// Now any iterator has count_where
let evens = (0..10).count_where(|x| x % 2 == 0);

//=====
// Pattern: Reflexive implementation
//=====

trait AsRefExt<T: ?Sized> {
    fn as_ref_ext(&self) -> &T;
}

// Every T can be viewed as &T
impl<T> AsRefExt<T> for T {
    fn as_ref_ext(&self) -> &T {
        self
    }
}

//=====
// Pattern: Into from From (std library pattern)
//=====

// This is how std implements Into<U> for all T: From<U>
trait MyInto<T> {

```

```

fn my_into(self) -> T;
}

trait MyFrom<T> {
    fn my_from(value: T) -> Self;
}

// Blanket impl: if you can convert FROM T, you can convert INTO T
impl<T, U> MyInto<U> for T
where
    U: MyFrom<T>,
{
    fn my_into(self) -> U {
        U::my_from(self)
    }
}

//=====
// Pattern: Conditional blanket impl
//=====

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

// Only for types that are both Debug and Clone
impl<T> Summarizable for T
where
    T: Debug + Clone,
{
    fn summary(&self) -> String {
        format!("{} (cloneable)", self)
    }
}

//=====
// Pattern: Blanket impl with references
//=====

trait Process {
    fn process(&self) -> String;
}

impl Process for i32 {
    fn process(&self) -> String {
        format!("processing {}", self)
    }
}

// Blanket impl: if T: Process, then &T: Process
impl<T: Process> Process for &T {
    fn process(&self) -> String {
        (*self).process()
    }
}

```

```

// Blanket impl: if T: Process, then Box<T>: Process
impl<T: Process> Process for Box<T> {
    fn process(&self) -> String {
        (**self).process()
    }
}

//=====
// Pattern: Negative reasoning (orphan rules)
//=====
// You cannot do blanket impl for foreign trait on foreign type
// This prevents conflicting impls across crates

// ALLOWED: Your trait, blanket impl
trait MyTrait {
    fn my_method(&self);
}
impl<T: Debug> MyTrait for T {
    fn my_method(&self) {
        println!("{}: {:?}", self);
    }
}

// ALLOWED: Foreign trait, your type
struct MyType;
impl Display for MyType {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "MyType")
    }
}

// NOT ALLOWED: Foreign trait, foreign type
// impl Display for Vec<i32> { ... } // ERROR: orphan rule

```

Blanket impl patterns: - Extension traits: Add methods to foreign types - Adapter patterns: Auto-convert between traits - Forwarding: Implement for references/smart pointers - Conditional behavior: Based on bounds

Coherence and orphan rules: - One impl per type-trait combination globally - Blanket impls must be in trait's crate or type's crate - Can't impl foreign trait for foreign type

Pattern 6: Phantom Types and Type-Level State

- **Problem:** You want to track state or constraints at compile time without runtime overhead. For example, a file handle should only allow reading if opened in read mode, or a builder should require certain fields before building.
- **Solution:** Use phantom types—type parameters that exist only in the type signature, not in the data layout. `PhantomData<T>` is a zero-sized type marker that tells the compiler “pretend this struct contains a T.” Combined with zero-sized state marker types, this enables the type-state pattern.

- **Why It Matters:** Phantom types move invariants from runtime to compile time. Invalid states become unrepresentable—you literally cannot call `.read()` on a write-only handle because the method doesn’t exist for that type.

Examples

```

use std::marker::PhantomData;

//=====
// Pattern: Type-state for protocol states
//=====

struct Disconnected;
struct Connected;
struct Authenticated;

struct Connection<State> {
    socket: String, // Simplified; would be real socket
    _state: PhantomData<State>,
}

impl Connection<Disconnected> {
    fn new() -> Self {
        Connection {
            socket: String::new(),
            _state: PhantomData,
        }
    }

    fn connect(self, addr: &str) -> Connection<Connected> {
        Connection {
            socket: addr.to_string(),
            _state: PhantomData,
        }
    }
}

impl Connection<Connected> {
    fn authenticate(self, _credentials: &str) -> Connection<Authenticated> {
        Connection {
            socket: self.socket,
            _state: PhantomData,
        }
    }

    fn disconnect(self) -> Connection<Disconnected> {
        Connection {
            socket: String::new(),
            _state: PhantomData,
        }
    }
}

```

```

impl Connection<Authenticated> {
    fn send(&mut self, _data: &[u8]) {
        // Only authenticated connections can send
    }

    fn logout(self) -> Connection<Connected> {
        Connection {
            socket: self.socket,
            _state: PhantomData,
        }
    }
}

// Usage - compile-time state enforcement
let conn = Connection::<Disconnected>::new();
let conn = conn.connect("localhost:8080");
// conn.send(&[1,2,3]); // ERROR: no method `send` on Connected
let mut conn = conn.authenticate("secret");
conn.send(&[1, 2, 3]); // OK: Authenticated has send

//=====
// Pattern: Units of measure
//=====

struct Meters;
struct Feet;
struct Seconds;

struct Quantity<T, Unit> {
    value: T,
    _unit: PhantomData<Unit>,
}

impl<T, Unit> Quantity<T, Unit> {
    fn new(value: T) -> Self {
        Quantity { value, _unit: PhantomData }
    }

    fn value(&self) -> &T {
        &self.value
    }
}

// Can only add same units
impl<T: std::ops::Add<Output = T>, Unit> std::ops::Add for Quantity<T, Unit> {
    type Output = Quantity<T, Unit>;

    fn add(self, other: Self) -> Self::Output {
        Quantity::new(self.value + other.value)
    }
}

let m1: Quantity<f64, Meters> = Quantity::new(10.0);
let m2: Quantity<f64, Meters> = Quantity::new(5.0);

```

```

let m3 = m1 + m2; // OK: both Meters

let f1: Quantity<f64, Feet> = Quantity::new(3.0);
// let bad = m3 + f1; // ERROR: can't add Meters and Feet

//=====
// Pattern: Builder with required fields
//=====

struct NoName;
struct HasName;
struct NoEmail;
struct HasEmail;

struct UserBuilder<Name, Email> {
    name: Option<String>,
    email: Option<String>,
    age: Option<u32>,
    _name_state: PhantomData<Name>,
    _email_state: PhantomData<Email>,
}

impl UserBuilder<NoName, NoEmail> {
    fn new() -> Self {
        UserBuilder {
            name: None,
            email: None,
            age: None,
            _name_state: PhantomData,
            _email_state: PhantomData,
        }
    }
}

impl<Email> UserBuilder<NoName, Email> {
    fn name(self, name: &str) -> UserBuilder<HasName, Email> {
        UserBuilder {
            name: Some(name.to_string()),
            email: self.email,
            age: self.age,
            _name_state: PhantomData,
            _email_state: PhantomData,
        }
    }
}

impl<Name> UserBuilder<Name, NoEmail> {
    fn email(self, email: &str) -> UserBuilder<Name, HasEmail> {
        UserBuilder {
            name: self.name,
            email: Some(email.to_string()),
            age: self.age,
            _name_state: PhantomData,
            _email_state: PhantomData,
        }
    }
}

```

```

        }
    }

impl<Name, Email> UserBuilder<Name, Email> {
    fn age(mut self, age: u32) -> Self {
        self.age = Some(age);
        self
    }
}

struct User {
    name: String,
    email: String,
    age: Option<u32>,
}

// build() only available when both Name and Email are set
impl UserBuilder<HasName, HasEmail> {
    fn build(self) -> User {
        User {
            name: self.name.unwrap(),
            email: self.email.unwrap(),
            age: self.age,
        }
    }
}

// Must provide name and email
let user = UserBuilder::new()
    .name("Alice")
    .email("alice@example.com")
    .age(30)
    .build();

// let incomplete = UserBuilder::new().name("Bob").build(); // ERROR

//=====
// Pattern: FFI ownership marker
//=====

struct Owned;
struct Borrowed;

struct CString<Ownership> {
    ptr: *mut i8,
    _ownership: PhantomData<Ownership>,
}

impl CString<Owned> {
    fn new(s: &str) -> Self {
        // Allocate and copy string
        let ptr = Box::into_raw(s.to_string().into_boxed_str()) as *mut i8;
        CString { ptr, _ownership: PhantomData }
    }
}

```

```

    }

}

impl Drop for CString<Owned> {
    fn drop(&mut self) {
        // Only owned strings need to be freed
        unsafe {
            drop(Box::from_raw(self.ptr));
        }
    }
}

impl CString<Borrowed> {
    unsafe fn from_ptr(ptr: *mut i8) -> Self {
        CString { ptr, _ownership: PhantomData }
    }
    // No Drop impl - we don't own it
}

```

Phantom type benefits: - Zero runtime cost (ZST optimized away) - Compile-time state verification - Self-documenting API constraints - Impossible to misuse

Common phantom patterns: - State machines: Protocol, connection, auth states - Units: Meters, seconds, currencies (can't mix) - Permissions: ReadOnly, WriteOnly, ReadWrite - Ownership: Owned, Borrowed, Shared

Pattern 7: Higher-Ranked Trait Bounds (HRTBs)

- **Problem:** You want to accept a closure that works with references of any lifetime, not a specific one. For example, a function that calls a closure with temporary references created inside the function.
- **Solution:** Use higher-ranked trait bounds with `for<'a>` syntax: `F: for<'a> Fn(&'a str) -> &'a str`. This means “F implements Fn for all possible lifetimes ‘a.’” The closure must work regardless of what lifetime the references have.
- **Why It Matters:** HRTBs are essential for closure-heavy APIs. Without them, you couldn’t write functions like `Vec::sort_by` that accept comparison closures operating on temporary references.

Examples

```

//=====
// Pattern: Basic HRTB for closure with references
//=====

fn call_with_ref<F>(f: F)
where
    F: for<'a> Fn(&'a str) -> usize,
{
    let local = String::from("hello");
    let result = f(&local); // 'a is the lifetime of local
    println!("Length: {}", result);
}

```

```

call_with_ref(|s| s.len()); // Closure works for any lifetime

//=====
// Pattern: HRTB vs regular lifetime parameter
//=====

// Regular lifetime: caller chooses lifetime
fn with_lifetime<'a, F>(s: &'a str, f: F) -> &'a str
where
    F: Fn(&'a str) -> &'a str,
{
    f(s)
}

// HRTB: function chooses lifetime, closure must handle any
fn with_hrtb<F>(f: F) -> String
where
    F: for<'a> Fn(&'a str) -> &'a str,
{
    let local = String::from("hello world");
    f(&local).to_string() // f must work with local's lifetime
}

//=====
// Pattern: Fn trait bounds are sugar for HRTB
//=====

// These are equivalent:
fn takes_fn_sugar<F: Fn(&str)>(f: F) { f("hi"); }
fn takes_fn_explicit<F>(f: F) where F: for<'a> Fn(&'a str) { f("hi"); }

// The sugar version automatically uses HRTB for reference parameters

//=====
// Pattern: HRTB with multiple lifetimes
//=====

fn call_with_two<F>(f: F)
where
    F: for<'a, 'b> Fn(&'a str, &'b str) -> bool,
{
    let s1 = String::from("hello");
    let s2 = String::from("world");
    let result = f(&s1, &s2);
    println!("Equal: {}", result);
}

//=====
// Pattern: HRTB in trait definitions
//=====

trait Parser<Output> {
    fn parse<'a>(&self, input: &'a str) -> Option<(Output, &'a str)>;
}

// Store parser that works for any input lifetime

```

```

struct BoxedParser<Output> {
    parser: Box<dyn for<'a> Fn(&'a str) -> Option<(Output, &'a str)>>,
}

impl<Output> BoxedParser<Output> {
    fn new<F>(f: F) -> Self
    where
        F: for<'a> Fn(&'a str) -> Option<(Output, &'a str)> + 'static,
    {
        BoxedParser { parser: Box::new(f) }
    }

    fn parse<'a>(&self, input: &'a str) -> Option<(Output, &'a str)> {
        (self.parser)(input)
    }
}

//=====
// Pattern: HRTB for iterator adapters
//=====

trait IteratorExt: Iterator {
    fn for_each_ref<F>(self, f: F)
    where
        Self: Sized,
        F: for<'a> FnMut(&'a Self::Item);
}

impl<I: Iterator> IteratorExt for I {
    fn for_each_ref<F>(self, mut f: F)
    where
        F: for<'a> FnMut(&'a Self::Item),
    {
        for item in self {
            f(&item);
        }
    }
}

//=====
// Pattern: Callback storage with HRTB
//=====

type Callback = Box<dyn for<'a> Fn(&'a str)>;

struct EventEmitter {
    callbacks: Vec<Callback>,
}

impl EventEmitter {
    fn new() -> Self {
        EventEmitter { callbacks: Vec::new() }
    }

    fn on<F>(&mut self, callback: F)
}

```

```

where
    F: for'a Fn(&'a str) + 'static,
{
    self.callbacks.push(Box::new(callback));
}

fn emit(&self, event: &str) {
    for callback in &self.callbacks {
        callback(event);
    }
}
}

//=====
// Pattern: Where HRTB is NOT needed (common mistake)
//=====

// DON'T need HRTB: lifetime comes from parameter
fn map_ref<'a, T, U, F>(value: &'a T, f: F) -> U
where
    F: Fn(&'a T) -> U, // No HRTB needed, 'a from parameter
{
    f(value)
}

// DO need HRTB: lifetime created inside function
fn create_and_process<F>(f: F)
where
    F: for'a Fn(&'a str) -> &'a str, // HRTB needed
{
    let local = String::from("temp");
    let _ = f(&local); // local's lifetime unknown to caller
}

```

When to use HRTB: - Closure works with references created inside your function - Storing callbacks that will be called with various lifetimes - Parser combinators and similar patterns - When compiler suggests it

When NOT to use HRTB: - Lifetime comes from function parameter - Closure returns owned value - No references involved

Pattern 8: Const Generics

- **Problem:** You need to parameterize types by compile-time constant values, not just types. Arrays in Rust have their size in the type: `[i32; 5]` is different from `[i32; 10]`.
- **Solution:** Use const generics: `struct Array<T, const N: usize>`. The `N` is a compile-time constant that becomes part of the type.
- **Why It Matters:** Const generics enable zero-cost fixed-size abstractions. Matrix multiplication `Matrix<3, 4> * Matrix<4, 5> = Matrix<3, 5>` is checked at compile time.

Examples

```
//=====
// Pattern: Basic const generic array
//=====

struct Array<T, const N: usizeimpl<T: Default + Copy, const N: usize> Array<T, N> {
    fn new() -> Self {
        Array { data: [T::default(); N] }
    }
}

impl<T, const N: usize> Array<T, N> {
    fn len(&self) -> usize {
        N // Known at compile time
    }

    fn get(&self, index: usize) -> Option<&T> {
        if index < N {
            Some(&self.data[index])
        } else {
            None
        }
    }
}

let arr: Array<i32, 5> = Array::new();
assert_eq!(arr.len(), 5);

//=====
// Pattern: Compile-time size validation
//=====

struct NonEmpty<T, const N: usize> {
    data: [T; N],
}

impl<T, const N: usize> NonEmpty<T, N> {
    fn new(data: [T; N]) -> Self {
        // Compile-time check that N > 0
        const { assert!(N > 0, "NonEmpty requires N > 0") }
        NonEmpty { data }
    }

    fn first(&self) -> &T {
        &self.data[0] // Always safe, N > 0
    }
}

let valid: NonEmpty<i32, 3> = NonEmpty::new([1, 2, 3]);
```

```

// let invalid: NonEmpty<i32, 0> = NonEmpty::new([]); // Compile error

//=====
// Pattern: Matrix with dimension checking
//=====

struct Matrix<T, const ROWS: usize, const COLS: usize> {
    data: [[T; COLS]; ROWS],
}

impl<T: Default + Copy, const R: usize, const C: usize> Matrix<T, R, C> {
    fn new() -> Self {
        Matrix { data: [[T::default(); C]; R] }
    }

    fn rows(&self) -> usize { R }
    fn cols(&self) -> usize { C }
}

impl<T, const R: usize, const C: usize> Matrix<T, R, C> {
    fn get(&self, row: usize, col: usize) -> Option<&T> {
        self.data.get(row)?.get(col)
    }

    fn transpose(self) -> Matrix<T, C, R>
    where
        T: Default + Copy,
    {
        let mut result = Matrix::<T, C, R>;::new();
        for i in 0..R {
            for j in 0..C {
                result.data[j][i] = self.data[i][j];
            }
        }
        result
    }
}

// Matrix multiplication with compile-time dimension checking
impl<T, const M: usize, const N: usize> Matrix<T, M, N>
where
    T: Default + Copy + std::ops::Add<Output = T> + std::ops::Mul<Output = T>,
{
    fn multiply<const P: usize>(
        &self,
        other: &Matrix<T, N, P>,
    ) -> Matrix<T, M, P> {
        let mut result = Matrix::<T, M, P>;::new();
        for i in 0..M {
            for j in 0..P {
                let mut sum = T::default();
                for k in 0..N {
                    sum = sum + self.data[i][k] * other.data[k][j];
                }
            }
        }
        result
    }
}

```

```

                result.data[i][j] = sum;
            }
        }
    result
}
}

let a: Matrix<i32, 2, 3> = Matrix::new();
let b: Matrix<i32, 3, 4> = Matrix::new();
let c: Matrix<i32, 2, 4> = a.multiply(&b); // Compiles!

// let bad = a.multiply(&a); // ERROR: 2x3 * 2x3 dimension mismatch

//=====
// Pattern: Fixed-size buffer / ring buffer
//=====

struct RingBuffer<T, const N: usize> {
    buffer: [Option<T>; N],
    head: usize,
    tail: usize,
    len: usize,
}

impl<T, const N: usize> RingBuffer<T, N> {
    fn new() -> Self
    where
        T: Copy,
    {
        RingBuffer {
            buffer: [None; N],
            head: 0,
            tail: 0,
            len: 0,
        }
    }

    fn push(&mut self, value: T) -> Result<(), T> {
        if self.len == N {
            Err(value) // Buffer full
        } else {
            self.buffer[self.tail] = Some(value);
            self.tail = (self.tail + 1) % N;
            self.len += 1;
            Ok(())
        }
    }

    fn pop(&mut self) -> Option<T> {
        if self.len == 0 {
            None
        } else {
            let value = self.buffer[self.head].take();
            self.head = (self.head + 1) % N;
            Some(value)
        }
    }
}

```

```

        self.len == 1;
        value
    }

}

fn capacity(&self) -> usize { N }
fn len(&self) -> usize { self.len }

}

//=====
// Pattern: Const generics with expressions
//=====

fn double_array<T: Copy + Default, const N: usize>(
    arr: [T; N],
) -> [T; N * 2] {
    let mut result = [T::default(); N * 2];
    result[..N].copy_from_slice(&arr);
    result[N..].copy_from_slice(&arr);
    result
}

//=====
// Pattern: Const generic for protocol frames
//=====

struct Frame<const SIZE: usize> {
    header: [u8; 4],
    payload: [u8; SIZE],
    checksum: u32,
}

impl<const SIZE: usize> Frame<SIZE> {
    fn new(payload: [u8; SIZE]) -> Self {
        Frame {
            header: [0; 4],
            payload,
            checksum: 0,
        }
    }

    fn total_size(&self) -> usize {
        4 + SIZE + 4 // header + payload + checksum
    }
}

// Different frame types
type SmallFrame = Frame<64>;
type LargeFrame = Frame<1024>;
type JumboFrame = Frame<9000>

//=====
// Pattern: Const bounds and where clauses
//=====

fn requires_at_least_two<T, const N: usize>(arr: [T; N]) -> (T, T)

```

```

where
    [(); N - 2]:, // Assert N >= 2
{
    let mut iter = arr.into_iter();
    (iter.next().unwrap(), iter.next().unwrap())
}

fn split_array<T, const N: usize, const M: usize>(
    arr: [T; N],
) -> ([T; M], [T; N - M])
where
    [(); N - M]:, // Assert N >= M
{
    todo!() // Implementation omitted for brevity
}

```

Const generic benefits: - Compile-time size checking - Zero runtime overhead - Type-safe fixed-size containers - Dimension-aware math operations

Current limitations: - Only primitive types (integers, bool, char) - Complex const expressions still unstable - No const generic associated types yet - Const bounds syntax is awkward

Performance Summary

| Feature | Compile Overhead | Runtime Overhead | Binary Size |
|------------------|------------------|------------------|-------------|
| Monomorphization | High | None | Increases |
| Trait bounds | Low | None | None |
| Associated types | Low | None | None |
| PhantomData | None | None | None |
| Const generics | Medium | None | Varies |
| HRTBs | Low | None | None |

Quick Reference: Choosing Generic Patterns

```

// Need same operation on different types?
fn operation<T: Trait>(x: T) {} // Generic function

// Need reusable data structure?
struct Container<T> { data: T } // Generic struct

// Need to constrain type capabilities?
fn foo<T: Clone + Debug>(x: T) {} // Trait bounds

// One implementation per type?
trait Iter { type Item; } // Associated type

// Multiple implementations per type?
trait From<T> { fn from(t: T) -> Self; } // Generic parameter

```

```

// Implement for all types with property?
impl<T: Display> ToString for T { } // Blanket impl

// Compile-time state tracking?
struct State<S> { _marker: PhantomData<S> } // Phantom type

// Closure works with any lifetime?
F: for<'a> Fn(&'a str) // HRTB

// Parameterize by constant value?
struct Array<T, const N: usize> // Const generic

```

Common Anti-Patterns

```

// ✗ Over-constraining: bounds not needed by implementation
fn foo<T: Clone + Debug + Display + Default>(x: T) {
    println!("{}:?", x); // Only uses Debug!
}

// ✓ Minimal bounds
fn foo<T: Debug>(x: T) {
    println!("{}:?", x);
}

// ✗ Generic when concrete works
fn always_i32<T>(x: T) -> i32 where T: Into<i32> {
    x.into()
}

// ✓ Just accept i32
fn always_i32(x: impl Into<i32>) -> i32 {
    x.into()
}

// ✗ Associated type when generic needed
trait Converter {
    type Output; // Can only convert to ONE type!
    fn convert(&self) -> Self::Output;
}

// ✓ Generic parameter for multiple conversions
trait Converter<T> {
    fn convert(&self) -> T;
}

// ✗ Phantom type without purpose
struct Wrapper<T, Phantom> {
    value: T,
    _p: PhantomData<Phantom>, // Does nothing!
}

// ✓ Phantom type with state meaning

```

```

struct Connection<State> {
    socket: Socket,
    _state: PhantomData<State>, // Enforces protocol
}

// ✗ Missing HRTB - won't compile
fn broken<F>(f: F) where F: Fn(&str) -> &str {
    let local = String::from("temp");
    let _ = f(&local); // ERROR: lifetime mismatch
}

// ✓ HRTB when creating references inside
fn works<F>(f: F) where F: for'a Fn(&'a str) -> &'a str {
    let local = String::from("temp");
    let _ = f(&local); // OK
}

```

Key Takeaways

- Monomorphization is zero-cost: Generic code compiles to specialized machine code
- Trait bounds are contracts: Specify exactly what capabilities you need
- Associated types simplify APIs: Use for “output” types in traits
- Blanket impls enable composition: Implement once, apply everywhere
- Phantom types are free: Zero runtime cost for compile-time guarantees
- HRTBs unlock closures: Essential for callback-heavy APIs
- Const generics enable value-level types: Compile-time dimension checking
- Start minimal: Add bounds only when the compiler requires them

Builder & API Design

This chapter explores several powerful API design patterns in Rust:

- **Builder Pattern:** For flexible and readable complex object construction.
- **Typestate Pattern:** For compile-time state machine validation.
- **Fluent APIs:** Using method chaining for ergonomic code.
- **Generic Parameters:** Employing `Into`/`AsRef` for flexible function arguments.
- **`#[must_use]` attribute:** To prevent accidental misuse of important return values.

Pattern 1: Builder Pattern Variations

The Builder pattern provides a flexible and readable way to construct complex objects, especially those with multiple optional fields or a lengthy configuration process.

- **Problem:** Constructors with numerous parameters are confusing and error-prone. It’s hard to remember the order of arguments, and handling optional fields with `Option<T>` is verbose.
- **Solution:** Instead of creating an object in a single step, a builder object is used to configure the final object piece by piece through a series of method calls. A final `.build()` method then constructs the object.

- **Why it matters:** This pattern leads to self-documenting code (e.g., `.timeout(30)`) is clearer than a positional argument). It improves ergonomics with a fluent, chainable API.

- **Use cases:**

- Building HTTP requests (`reqwest::Client::get()`).
- Configuring database connections (`sqlx::postgres::PgConnectOptions`).
- Constructing complex UI components or application configuration objects.
- Creating test data with varying parameters.

Example: Basic Consuming Builder

This is the most common builder variant. Each setter method consumes the builder (`takes self`) and returns a new one, allowing for method chaining. The final `.build()` call consumes the builder and returns the constructed object. This ensures the builder cannot be accidentally reused.

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

#[derive(Debug)]
pub struct Request {
    url: String,
    method: String,
    headers: Vec<(String, String)>,
    body: Option<String>,
    timeout: Option<Duration>,
    retry_count: u32,
    follow_redirects: bool,
}

pub struct RequestBuilder {
    url: String,
    method: String,
    headers: Vec<(String, String)>,
    body: Option<String>,
    timeout: Option<Duration>,
    retry_count: u32,
    follow_redirects: bool,
}

impl Request {
    // Provide a convenient entry point to the builder.
    pub fn builder(url: impl Into<String>) -> RequestBuilder {
        RequestBuilder::new(url)
    }
}

impl RequestBuilder {
    // The `new` function sets defaults for all fields.
    pub fn new(url: impl Into<String>) -> Self {
        RequestBuilder {
            url: url.into(),
            method: "GET".to_string(),
            headers: Vec::new(),
            body: None,
            timeout: Some(Duration::from_secs(10)),
            retry_count: 3,
            follow_redirects: true,
        }
    }

    pub fn url(self, url: impl Into<String>) -> Self {
        Self { url: url.into(), ..self }
    }

    pub fn method(self, method: &str) -> Self {
        Self { method: method.to_string(), ..self }
    }

    pub fn headers(self, headers: Vec<(String, String)>) -> Self {
        Self { headers, ..self }
    }

    pub fn body(self, body: String) -> Self {
        Self { body: Some(body), ..self }
    }

    pub fn timeout(self, timeout: Duration) -> Self {
        Self { timeout: Some(timeout), ..self }
    }

    pub fn retry_count(self, count: u32) -> Self {
        Self { retry_count, ..self }
    }

    pub fn follow_redirects(self, follows: bool) -> Self {
        Self { follow_redirects, ..self }
    }

    pub fn build(self) -> Request {
        Request {
            url: self.url,
            method: self.method,
            headers: self.headers,
            body: self.body,
            timeout: self.timeout,
            retry_count: self.retry_count,
            follow_redirects: self.follow_redirects,
        }
    }
}
```

```

        headers: Vec::new(),
        body: None,
        timeout: None,
        retry_count: 0,
        follow_redirects: true,
    }
}

// Each method takes `self` and returns `self` to enable chaining.
pub fn method(mut self, method: impl Into<String>) -> Self {
    self.method = method.into();
    self
}

pub fn header(mut self, key: impl Into<String>, value: impl Into<String>) -> Self {
    self.headers.push((key.into(), value.into()));
    self
}

pub fn body(mut self, body: impl Into<String>) -> Self {
    self.body = Some(body.into());
    self
}

// The `build` method consumes the builder and creates the final object.
pub fn build(self) -> Request {
    Request {
        url: self.url,
        method: self.method,
        headers: self.headers,
        body: self.body,
        timeout: self.timeout,
        retry_count: self.retry_count,
        follow_redirects: self.follow_redirects,
    }
}
}

// The call site is now fluent, readable, and self-documenting.
let request = Request::builder("https://api.example.com")
    .method("POST")
    .header("Authorization", "Bearer token")
    .body("{\"data\": \"value\"}")
    .build();

println!("{:?}", request);

```

Example: Builder with Runtime Validation

When some fields are required, the builder can store them as `Option<T>` and the `.build()` method can return a `Result`. This moves validation from compile time to a single runtime check, ensuring no required fields are missed.

```

#[derive(Debug)]
pub struct Database {
    host: String,
    port: u16,
    username: String,
}

// The builder stores required fields as `Option`.
pub struct DatabaseBuilder {
    host: Option<String>,
    port: Option<u16>,
    username: Option<String>,
}

impl DatabaseBuilder {
    pub fn new() -> Self {
        DatabaseBuilder { host: None, port: None, username: None }
    }

    pub fn host(mut self, host: impl Into<String>) -> Self {
        self.host = Some(host.into());
        self
    }

    pub fn port(mut self, port: u16) -> Self {
        self.port = Some(port);
        self
    }

    pub fn username(mut self, username: impl Into<String>) -> Self {
        self.username = Some(username.into());
        self
    }

    // `build` returns a `Result` to enforce that required fields were set.
    pub fn build(self) -> Result<Database, String> {
        let host = self.host.ok_or("host is required")?;
        let port = self.port.ok_or("port is required")?;
        let username = self.username.ok_or("username is required")?;

        Ok(Database { host, port, username })
    }
}

// The caller must now handle the `Result`.
let db_result = DatabaseBuilder::new()
    .host("localhost")
    .port(5432)
    .username("admin")
    .build();
assert!(db_result.is_ok());
println!("{:?}", db_result.unwrap());

```

```
// This would fail at runtime because a required field is missing.
let db_fail = DatabaseBuilder::new().host("localhost").build();
assert!(!db_fail.is_err());
println!("{}", db_fail.unwrap_err());
```

Example: Non-Consuming (Mutable) Builder

For builders that you might want to reuse or incrementally build, methods can take a mutable reference (`&mut self`). This allows calling methods multiple times and creating multiple objects from the same builder instance.

```
#[derive(Debug)]
pub struct Email {
    to: Vec<String>,
    subject: String,
    body: String,
}

pub struct EmailBuilder {
    to: Vec<String>,
    subject: String,
    body: String,
}

impl EmailBuilder {
    pub fn new() -> Self {
        EmailBuilder { to: Vec::new(), subject: String::new(), body: String::new() }
    }

    // Methods take `&mut self` and return `&mut Self` for chaining.
    pub fn to(&mut self, email: impl Into<String>) -> &mut Self {
        self.to.push(email.into());
        self
    }

    pub fn subject(&mut self, subject: impl Into<String>) -> &mut Self {
        self.subject = subject.into();
        self
    }

    pub fn body(&mut self, body: impl Into<String>) -> &mut Self {
        self.body = body.into();
        self
    }

    // The build/send method now typically borrows the builder.
    pub fn build(&self) -> Email {
        Email {
            to: self.to.clone(),
            subject: self.subject.clone(),
            body: self.body.clone()
        }
    }
}
```

```

    }

}

pub fn clear(&mut self) {
    self.to.clear();
    self.subject.clear();
    self.body.clear();
}
}

let mut builder = EmailBuilder::new();

// Build the first email.
builder.to("a@example.com").subject("First").body("This is the first email.");
let email1 = builder.build();
println!("Built email 1: {:?}", email1);

// Reuse the builder for a second email after clearing it.
builder.clear();
builder.to("b@example.com").subject("Second").body("This is the second email.");
let email2 = builder.build();
println!("Built email 2: {:?}", email2);

```

Pattern 2: Typestate Pattern

The Typestate pattern encodes the state of an object into the type system itself. This makes invalid state transitions a compile-time error rather than a runtime panic.

- **Problem:** State machines are often implemented with enums and checked at runtime (e.g., `if self.state == State::Connected`). This is error-prone, as it's easy to forget a state, handle a transition incorrectly, or call a method in the wrong state, leading to panics.
- **Solution:** Represent each state with a distinct type. State transitions are functions that consume an object in one state and return a new object in another state.
- **Why it matters:** This pattern makes invalid states and transitions impossible to represent in code. It moves state machine validation from runtime to compile time, leveraging Rust's type system to create safer, self-documenting APIs.
- **Use cases:**
 - Database connections (`Unauthenticated -> Authenticated`).
 - File handles (`Open -> Written -> Closed`).
 - Protocol state machines (e.g., HTTP request/response cycle).
 - Builder patterns that require fields to be set in a specific order (see Example 2).
 - Resource lifecycle management (`Acquired -> Released`).

Example: Typestate for a Connection

This example models a connection that can be `Disconnected` or `Connected`. Methods like `send` are only available on a `Connection<Connected>`, which the compiler enforces.

```
use std::marker::PhantomData;
use std::io::{self, Write};
use std::net::TcpStream;

// State marker types are zero-sized structs.
#[derive(Debug)]
struct Disconnected;
#[derive(Debug)]
struct Connected;

// The Connection is generic over its state.
struct Connection<State> {
    stream: Option<TcpStream>,
    _state: PhantomData<State>,
}

// In the `Disconnected` state, we can only connect.
impl Connection<Disconnected> {
    fn new() -> Self {
        Connection { stream: None, _state: PhantomData }
    }

    fn connect(self, addr: &str) -> io::Result<Connection<Connected>> {
        let stream = TcpStream::connect(addr)?;
        println!("Connected to {}", addr);
        Ok(Connection { stream: Some(stream), _state: PhantomData })
    }
}

// In the `Connected` state, we can send data.
impl Connection<Connected> {
    fn send(&mut self, data: &[u8]) -> io::Result<()> {
        let stream = self.stream.as_mut().expect("Stream must exist in Connected state");
        stream.write_all(data)?;
        println!("Sent data: {}", String::from_utf8_lossy(data));
        Ok(())
    }

    fn close(self) -> Connection<Disconnected> {
        if let Some(stream) = self.stream {
            drop(stream); // Close the connection.
        }
        println!("Connection closed.");
        Connection { stream: None, _state: PhantomData }
    }
}
```

```
// The compiler enforces correct usage.
let conn = Connection::new();
// conn.send(b"hello"); // COMPILE ERROR: `send` is not a member of `Connection<Disconnected>`


match conn.connect("127.0.0.1:8080") {
    Ok(mut connected_conn) => {
        connected_conn.send(b"hello from typestate!").unwrap();
        let _disconnected_conn = connected_conn.close();
        // connected_conn.send(b"world"); // COMPILE ERROR: value used here after move
    }
    Err(e) => {
        println!("Failed to connect: {}", e);
    }
}
```

Example: Typestate Builder for Compile-Time Validation

The typestate pattern can be combined with a builder to enforce that required fields are set *at compile time*. The `.build()` method is only made available on the final state type, after all required setters have been called.

```
use std::marker::PhantomData;

// State markers for the builder
#[derive(Default)]
struct NoName;
#[derive(Default)]
struct HasName;
#[derive(Default)]
struct NoEmail;
#[derive(Default)]
struct HasEmail;

#[derive(Debug)]
struct User { name: String, email: String }

// The builder is generic over its name and email states.
struct UserBuilder<NameState, EmailState> {
    name: Option<String>,
    email: Option<String>,
    _name_state: PhantomData<NameState>,
    _email_state: PhantomData<EmailState>,
}

// Initial state: no name, no email.
impl Default for UserBuilder<NoName, NoEmail> {
    fn default() -> Self {
        UserBuilder { name: None, email: None, _name_state: PhantomData, _email_state: PhantomData }
    }
}
```

```

// Methods transition the builder to a new state by returning a new type.
impl<E> UserBuilder<NoName, E> {
    fn name(self, name: impl Into<String>) -> UserBuilder<HasName, E> {
        UserBuilder { name: Some(name.into()), email: self.email, _name_state: PhantomData,
        _email_state: PhantomData }
    }
}

impl<N> UserBuilder<N, NoEmail> {
    fn email(self, email: impl Into<String>) -> UserBuilder<N, HasEmail> {
        UserBuilder { name: self.name, email: Some(email.into()), _name_state: PhantomData,
        _email_state: PhantomData }
    }
}

// The `build` method is only available in the `HasName, HasEmail` state.
impl UserBuilder<HasName, HasEmail> {
    fn build(self) -> User {
        User {
            name: self.name.expect("Name is guaranteed by typestate"),
            email: self.email.expect("Email is guaranteed by typestate")
        }
    }
}

// The compiler enforces the builder logic.
let user = UserBuilder::default()
    .name("Alice")
    .email("alice@example.com")
    .build();
println!("Successfully built user: {:?}", user);

// The following line would fail to compile because `build()` is not available.
// The error message is clear: "no method named `build` found for struct `UserBuilder<HasName,
// NoEmail>`"
// let incomplete_user = UserBuilder::default().name("Bob").build();

```

Pattern 3: #[must_use] for Critical Return Values

This attribute signals that a function's return value is important and should not be ignored. The compiler will issue a warning if a value marked `#[must_use]` is discarded.

- **Problem:** Functions that return a `Result` or `Option` can have their return value silently ignored, leading to unhandled errors or logic bugs. Some types, like builders or transaction guards, are useless unless a final method (`.build()`, `.commit()`) is called.
- **Solution:** Apply the `#[must_use]` attribute to functions or types. This instructs the compiler to generate a warning if a returned value of that type is not “used” in some way (e.g., assigned to a variable, passed to another function, or having a method called on it).

- **Why it matters:** `#[must_use]` makes APIs safer by default. It prevents entire classes of bugs caused by silently ignoring errors or failing to complete a required workflow.

- **Use cases:**

- `Result` and `Option` types are the canonical examples.
- Builder patterns, to ensure `.build()` is called.
- Resource handles that must be explicitly closed or released.
- Transaction guards that must be `.commit()`-ed or rolled back.
- Futures, which do nothing unless they are `.await`-ed.

Example: Applying `#[must_use]` to Functions and Types

The standard library widely uses `#[must_use]`, for example on `Result` and `Option`. You can apply it to your own types and functions to guide users towards correct usage.

```
// Applying `#[must_use]` to a function's return value.
// A custom message explains why it's important.
#[must_use = "this `Result` may be an error, which should be handled"]
pub fn connect_to_db() -> Result<(), &'static str> {
    Err("Failed to connect")
}

// Applying `#[must_use]` to a type definition.
// Any function returning this type will implicitly be `must_use`.
#[must_use = "a builder does nothing unless you call ` `.build()` `"]
pub struct ConnectionBuilder;

impl ConnectionBuilder {
    pub fn new() -> Self { ConnectionBuilder }
    pub fn build(self) {}
}

fn main() {
    // If you call these functions without using their return values, you'll get a warning.

    // connect_to_db();
    // WARNING: unused `Result` that must be used

    // ConnectionBuilder::new();
    // WARNING: unused `ConnectionBuilder` that must be used

    // Correct usage:
    if let Err(e) = connect_to_db() {
        println!("Error: {}", e);
    }

    let builder = ConnectionBuilder::new();
    builder.build();
}
```

```
    println!("Builder was used correctly.");
}
```

Lifetime Patterns

This chapter explores essential lifetime patterns in Rust, covering how the compiler infers lifetimes through elision, how to use lifetime bounds for generic constraints, how higher-ranked trait bounds enable lifetime polymorphism for closures, the role of variance in subtyping, and how `Pin` makes self-referential structures safe.

Pattern 1: Named Lifetimes and Elision

Problem: The compiler needs to know that references are valid, but annotating every single reference with a lifetime would be extremely verbose. It's not always obvious when lifetimes are required versus when they are inferred, and how to specify the relationship between multiple references.

Solution: Rust uses **lifetime elision rules** to automatically infer lifetimes in common, unambiguous cases. For more complex scenarios, you use explicit lifetime annotations like `'a` to tell the compiler how the lifetimes of different references relate to each other.

Why it matters: Lifetimes are a zero-cost, compile-time feature that prevents an entire class of memory safety bugs, like use-after-free errors. The elision rules make this powerful feature ergonomic, covering over 90% of use cases so you only need to write explicit lifetimes when the relationships are ambiguous.

Use Cases: - Functions that take multiple references (e.g., finding the longest of two strings). - Structs that hold references to data. - Functions that return a reference derived from one of its inputs. - Methods on structs that return references to the struct's data.

Example: Why Lifetimes Exist

In languages like C, it's easy to accidentally return a pointer to memory that has been deallocated, leading to crashes.

```
// C code - compiles but crashes!
char* get_string() {
    char buffer[100];
    strcpy(buffer, "Hello");
    return buffer; // Returns a pointer to stack memory that is now invalid!
}
```

The function returns a pointer to stack memory that's immediately deallocated. Using this pointer is undefined behavior. Lifetimes prevent this entire class of bugs: [### Example: Lifetime Elision Rules](#)

```
// This Rust code will not compile.
// fn get_string() -> &str {
//     let s = String::from("Hello");
```

```
//     &s // Error: cannot return reference to local variable `s`
// }
```

The compiler sees that the returned reference doesn't outlive the function and rejects the code. Lifetimes encode "how long is this reference valid?" in the type system.

When a function takes multiple references and returns one, you must explicitly tell the compiler how the lifetimes are related. Here, `'a` ensures that the returned reference is valid for as long as the shorter of the two input references. **Example: Basic Lifetime Annotation**

When a function takes multiple references and returns one, you must explicitly tell the compiler how the lifetimes are related. Here, `'a` ensures that the returned reference is valid for as long as the shorter of the two input references.

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

fn example() {
    let string1 = String::from("long string");
    let string2 = String::from("short");

    let result = longest(&string1, &string2);
    println!("Longest: {}", result);
}
```

Example: Lifetime Elision Rules

To avoid boilerplate, the compiler applies three rules to infer lifetimes automatically. You only need to write annotations when these rules are not sufficient.

- **Rule 1:** Each elided lifetime in a function's parameters gets its own distinct lifetime parameter.
- **Rule 2:** If there is exactly one input lifetime, that lifetime is assigned to all output lifetimes.
- **Rule 3:** If one of the parameters is `&self` or `&mut self`, its lifetime is assigned to all output lifetimes.

```
// Elision Rule 2 applies here.
// The compiler infers that the output lifetime is the same as the input lifetime.
fn first_word(s: &str) -> &str {
    s.split_whitespace().next().unwrap_or(s)
}

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

```
// Elision Rule 3 applies here.
// The output lifetime is tied to the lifetime of `&self`.
impl<'a> MyString<'a> {
    fn get_text(&self) -> &str {
        self.text
    }
}
```

Example: The '`static` Lifetime

The '`static`' lifetime indicates that a reference is valid for the entire duration of the program. String literals are the most common example. Be cautious with '`static`', as it is rarely what you need for function inputs or outputs unless you are dealing with truly global data.

```
// `s` is a reference to data that is hardcoded into the program's binary.
let s: &'static str = "I have a static lifetime.";

// You can also create static data with `const`.
const STATIC_STRING: &'static str = "This is also a static string.>";
```

Pattern 2: Lifetime Bounds

Problem: When a generic type holds a reference, the compiler needs to ensure that the referenced data lives at least as long as the generic type itself. For example, in a `struct Wrapper<'a, T>`, how do we ensure `T` doesn't contain a reference that dies before '`a`'?

Solution: Use **lifetime bounds**. The syntax `T: 'a` means that the type `T` must "outlive" the lifetime '`a`.

Why it matters: Lifetime bounds are crucial for the safety of generic types that contain references. They ensure that you cannot create a generic struct holding a reference to data that might be destroyed while the struct is still in use.

Use Cases: - Generic structs that hold references, like caches or parsers. - Generic functions that work with borrowed data. - Traits that involve references in their method signatures or associated types.

Example: Lifetime Bound on a Generic Struct

A struct containing a generic type with a reference needs a lifetime bound. Here, `T: 'a` ensures that whatever type `T` is, it does not contain any references that live for a shorter time than '`a`'.

```
// `T: 'a` means `T` must outlive ``a``.
// In modern Rust, this bound is inferred from the `&'a T` field.
struct Wrapper<'a, T: 'a> {
    value: &'a T,
}
```

Example: `where` Clauses for Complex Bounds

For complex combinations of lifetime and trait bounds, a `where` clause can make the function signature much more readable.

```
fn process_and_debug<'a, T>(items: &'a [T])
where
    T: std::fmt::Debug + 'a, // T must be debug-printable and outlive 'a
{
    for item in items {
        println!("Item: {:?}", item);
    }
}
```

1. Implementing traits with lifetime parameters

```
trait Parser {
    fn parse<'a>(&self, input: &'a str) -> Option<&'a str>;
}
```

Pattern 3: Higher-Ranked Trait Bounds (for Lifetimes)

Problem: You need to write a function that accepts a closure, but the closure must work for *any* lifetime, not just one specific lifetime that you can name. This is common for iterator adapters or any function that calls a closure with locally created references.

Solution: Use a **Higher-Ranked Trait Bound (HRTB)** with the `for<'a>` syntax. The bound `F: for<'a> Fn(&'a str)` means that the closure `F` must work for a reference `&str` of *any* possible lifetime `'a`.

Why it matters: HRTBs are the key to Rust's powerful and flexible functional programming patterns. They allow you to write generic, higher-order functions that accept closures operating on borrowed data, without needing to tie the closure to a single, specific lifetime.

Use Cases: - Iterator adapters like `map`, `filter`, and `for_each` when working with references. - Functions that accept callbacks or event handlers. - Parser combinator libraries.

Example: A Function Accepting a Lifetime-Generic Closure

The `call_on_hello` function creates a local string and calls a closure on a reference to it. The closure must be able to handle this local, temporary lifetime. The `for<'a>` bound ensures this.

```
// The HRTB `for<'a> Fn(&'a str)` ensures `f` works for any lifetime.
fn call_on_hello<F>(f: F)
where
    F: for<'a> Fn(&'a str),
{
    let s = String::from("hello");
    f(&s); // The closure is called with a reference local to this function.
```

```

}

// This closure works for any &str, so it can be passed to `call_on_hello`.
let print_it = |s: &str| println!("{}", s);
call_on_hello(print_it);

```

Example: Trait with a Higher-Ranked Method

You can use HRTBs in traits to define methods that are generic over lifetimes. This is common in “streaming iterator” or “lending iterator” patterns.

```

trait Processor {
    // This method must work for any input lifetime 'a.
    fn process<'a>(&self, data: &'a str) -> &'a str;
}

struct Trimmer;

impl Processor for Trimmer {
    fn process<'a>(&self, data: &'a str) -> &'a str {
        data.trim()
    }
}

```

Pattern 4: Self-Referential Structs and Pin

Problem: It is normally impossible to create a struct that holds a reference to one of its own fields. The borrow checker forbids this because if the struct were moved, the internal reference would become invalid (dangling).

Solution: Use `Pin<T>`. A `Pin` “pins” a value to its location in memory, guaranteeing that it will not be moved.

Why it matters: `Pin` is the cornerstone that makes `async/await` in Rust work safely and efficiently. Futures in `async` Rust are often self-referential, and `Pin` ensures that they can be polled without their internal references being invalidated.

Use Cases: - Async `Futures`, which store state across `.await` points. - Generators and other coroutines. - Intrusive data structures like linked lists where nodes are embedded within other objects.

Example: The Problem with Self-Reference

This code demonstrates why safe Rust disallows self-referential structs. You cannot create a reference to a field before the struct is fully constructed and moved into its final memory location.

```

// This will not compile.
// struct SelfReferential<'a> {
//     data: String,
//     // This reference is supposed to point to `data`.

```

```
//     reference: &'a str,  
// }
```

Example: A Safe Alternative Using Indices

Instead of direct references, you can use indices into a collection. This avoids the self-reference problem because indices remain valid even if the collection is moved.

```
// A graph where nodes reference each other via indices, not pointers.  
struct Node {  
    name: String,  
    edges: Vec<u32>, // Indices of other nodes in the graph's `nodes` vector.  
}  
  
struct Graph {  
    nodes: Vec<Node>,  
}
```

Example: A Pinned, Self-Referential Struct (Unsafe)

This example shows how `Pin` and `unsafe` can be used to create a truly self-referential struct. This is an advanced technique and should be used with great care.

```
use std::pin::Pin;  
use std::marker::PhantomPinned;  
  
struct Unmovable {  
    data: String,  
    slice: *const str, // A raw pointer, not a reference  
    _pin: PhantomPinned,  
}  
  
impl Unmovable {  
    fn new(data: String) -> Pin<Box<Self>> {  
        let res = Unmovable {  
            data,  
            // Can't initialize `slice` yet, as `data` is not pinned.  
            slice: std::ptr::null(),  
            _pin: PhantomPinned,  
        };  
        let mut boxed = Box::pin(res);  
  
        // Now that the data is pinned, we can create a pointer to it.  
        let slice = &boxed.data[...] as *const str;  
        // And update the `slice` field with the correct pointer.  
        unsafe {  
            let mut_ref: Pin<&mut Self> = Pin::as_mut(&mut boxed);  
            Pin::get_unchecked_mut(mut_ref).slice = slice;  
        }  
        boxed  
    }  
}
```

```

}

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

fn reference(&self) -> &str {
    unsafe { &*self.reference }
}

}

```

`Pin` guarantees the struct won't move, making the self-reference safe. This is advanced and requires `unsafe`.

Solution 3: Rental Crates

Libraries like `ouroboros` provide safe abstractions:

```

// Using ouroboros crate
use ouroboros::self_referencing;

#[self_referencing]
struct SelfRef {
    data: String,
    #[borrows(data)]
    reference: &'this str,
}

fn example() {
    let s = SelfRefBuilder {
        data: "hello".to_string(),
        reference_builder: |data| &data[..],
    }.build();

    s.with_reference(|r| println!("{}", r));
}

```

The `ouroboros` crate uses macros and `Pin` internally to provide a safe interface.

Solution 4: Restructure the Design

Often, self-references indicate a design problem. Consider alternatives:

```

// Instead of self-referential:
struct BadDesign<'a> {
    data: String,
    view: &'a str,
}

// Use two separate types:
struct Data {

```

```

        content: String,
    }

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

impl Data {
    fn view(&self) -> View {
        View {
            data: self,
            window: &self.content[...],
        }
    }
}

```

This separates ownership from borrowing, eliminating the self-reference.

When Self-References Are Actually Needed

Rare cases that truly need self-references:

- 1. Async runtimes:** Futures containing references to their own data
- 2. Parsers:** Holding both input buffer and views into it
- 3. Game engines:** Scene graphs with parent-child relationships

For these, use **Pin**, arena allocation, or specialized crates.

Pattern 5: Variance and Subtyping

Problem: Lifetime subtyping rules unclear—when can **&'long** be used where **&'short** expected?
Covariant vs contravariant vs invariant confusing.

Solution: Covariant types accept longer lifetimes: **&'a T** is covariant in **'a**, so **&'long T** usable where **&'short T** expected. Invariant types require exact lifetime: **&mut 'a T** is invariant—can't substitute.

Why It Matters: Enables flexible lifetime assignments—longer lifetime works where shorter needed.
Immutable reference covariance allows ergonomic code: can pass long-lived reference to function expecting short.

Use Cases: Reference wrappers (determining if **Wrapper<'a>** covariant), iterator chains (covariant iterators compose naturally), function pointers (contravariant arguments, covariant returns), trait objects (variance of **dyn Trait<'a>**), smart pointers with references (**Arc<&'a T>** variance), custom pointer types (controlling subtyping behavior with **PhantomData**), phantom data usage (adding variance markers to generic types).

```

fn example() {
    let outer: &'static str = "hello";

    {

```

```
    let inner: &str = outer; // OK: 'static is subtype of shorter lifetime
}
}
```

If '`long`: `short`' (read: "'long outlives 'short"), then '`long`' is a subtype of '`short`'. You can use a longer lifetime where a shorter one is expected.

Variance Categories

Types have variance with respect to their lifetime and type parameters:

Covariant: Subtyping flows in the same direction

```
// &'a T is covariant over 'a
// If 'a: 'b, then &'a T <: &'b T

fn covariant_example() {
    let long: &'static str = "hello";
    let short: &str = long; // OK
}
```

Invariant: No subtyping allowed

```
// &'a mut T is invariant over 'a
// Cannot substitute different lifetimes

fn invariant_example<'a, 'b>(x: &'a mut i32, y: &'b mut i32)
where
    'a: 'b,
{
    // Cannot assign x to y even though 'a: 'b
    // let z: &'b mut i32 = x; // Error!
}
```

Contravariant: Subtyping flows in opposite direction (rare in Rust)

```
// Function arguments are contravariant over lifetimes
// fn(&'a T) is contravariant over 'a
```

Why Variance Matters

Variance determines when types are compatible:

```
// Covariance allows this:
fn take_short(x: &str) {}

fn example() {
    let s: &'static str = "hello";
    take_short(s); // OK: can pass 'static where shorter lifetime expected
}
```

```
// Invariance prevents this:
fn swap<'a, 'b>(x: &'a mut &'static str, y: &'b mut &'a str) {
    // std::mem::swap(x, y); // Error! Invariance prevents swapping
}
```

Variance in Practice

Common types and their variance:

```
// Covariant:
// &'a T
// *const T
// fn() -> T
// Vec<T>, Box<T>, etc.

// Invariant:
// &'a mut T
// *mut T
// Cell<T>, UnsafeCell<T>

// Contravariant (rare):
// fn(T) -> ()
```

Example showing practical impact:

```
struct Producer<T> {
    produce: fn() -> T, // Covariant over T
}

struct Consumer<T> {
    consume: fn(T), // Contravariant over T
}

fn example() {
    // Can use Producer<&'static str> where Producer<&'a str> expected
    let p: Producer<&'static str> = Producer { produce: || "hello" };
    let _p2: Producer<&str> = p; // OK

    // Contravariance with Consumer
    let c: Consumer<&str> = Consumer { consume: |_s| {} };
    // let _c2: Consumer<&'static str> = c; // Would be error if uncommented
}
```

Interior Mutability and Invariance

Interior mutability types are invariant:

```
use std::cell::Cell;
```

```

fn example() {
    let cell: Cell<&'static str> = Cell::new("hello");

    // Cannot do this even though 'static: 'a
    // fn take_cell<'a>(c: Cell<&'a str>) {}
    // take_cell(cell); // Error! Cell is invariant
}

```

Invariance prevents creating references with incorrect lifetimes through interior mutability.

PhantomData and Variance

Control variance explicitly with `PhantomData`:

```

use std::marker::PhantomData;

// Covariant over T
struct Covariant<T> {
    _marker: PhantomData<T>,
}

// Invariant over T
struct Invariant<T> {
    _marker: PhantomData<Cell<T>>,
}

// Contravariant over T (rare)
struct Contravariant<T> {
    _marker: PhantomData<fn(T)>,
}

```

Use `PhantomData` when you need to control variance without storing the actual type.

Subtyping and Higher-Rank Trait Bounds

HRTBs interact with variance:

```

// Works because of variance
fn accepts_any_lifetime<F>(f: F)
where
    F: for<'a> Fn(&'a str) -> String,
{
    f("hello");
}

fn example() {
    // This closure works with any lifetime
    accepts_any_lifetime(|s: &str| s.to_uppercase());
}

```

The HRTB ensures the function works with any lifetime, leveraging variance.

Lifetime Subtyping in APIs

Design APIs to work with variance:

```
// Good: covariant, flexible
struct GoodReader<'a> {
    data: &'a [u8],
}

impl<'a> GoodReader<'a> {
    fn read(&self) -> &'a [u8] {
        self.data
    }
}

// Can use GoodReader<'static> where GoodReader<'a> expected

// Bad: invariant, inflexible
struct BadReader<'a> {
    data: &'a mut [u8],
}

// Cannot substitute different lifetimes
```

Prefer immutable references for flexibility unless mutation is necessary.

Pattern 6: Advanced Lifetime Patterns

Let's explore some sophisticated patterns that combine these concepts.

Lifetime Bounds with Closures

```
fn process_with_context<'a, F, T>(
    data: &'a str,
    context: &'a T,
    f: F,
) -> String
where
    F: Fn(&'a str, &'a T) -> String,
{
    f(data, context)
}
```

The closure receives references tied to the input lifetimes.

Streaming Iterators

Iterator items that borrow from the iterator itself:

```

trait StreamingIterator {
    type Item<'a> where Self: 'a;

    fn next(&mut self) -> Option<Self::Item<'_>>;
}

struct WindowIter<'a> {
    data: &'a [i32],
    window_size: usize,
    position: usize,
}

impl<'a> StreamingIterator for WindowIter<'a> {
    type Item<'b> = &'b [i32] where Self: 'b;

    fn next(&mut self) -> Option<Self::Item<'_>> {
        if self.position + self.window_size <= self.data.len() {
            let window = &self.data[self.position..self.position + self.window_size];
            self.position += 1;
            Some(window)
        } else {
            None
        }
    }
}

```

This pattern allows iterators to yield references tied to the iterator's lifetime.

Lifetime Elision in Impl Blocks

```

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

impl<'a> Parser<'a> {
    // Elided: fn parse(&self) -> Option<&str>
    // Actual: fn parse(&'a self) -> Option<&'a str>
    fn parse(&self) -> Option<&str> {
        Some(self.input)
    }

    // Multiple lifetimes when needed
    fn parse_with<'b>(&self, other: &'b str) -> (&'a str, &'b str) {
        (self.input, other)
    }
}

```

Anonymous Lifetimes

Use `'_` for clarity without naming:

```

impl<'a> Parser<'a> {
    fn peek(&self) -> Option<&'_ str> {
        // '_' = 'a in this context
        Some(self.input)
    }
}

// Generic context
fn get_first<T>(vec: &Vec<T>) -> Option<&'_ T> {
    vec.first()
}

```

Anonymous lifetimes improve readability when the specific lifetime name doesn't matter.

Summary

This chapter covered lifetime patterns for ensuring reference validity:

- Named Lifetimes and Elision:** Three elision rules infer common cases, explicit '`a`' for complex relationships
- Lifetime Bounds and Where Clauses:** `T: 'a` (`T` outlives '`a`'), `'b: 'a` ('`b` outlives '`a`'), implied bounds
- Higher-Ranked Trait Bounds:** `for<'a> Fn(&'a str)` for lifetime-polymorphic closures
- Self-Referential Structs and Pin:** `Pin` enables safe self-references, essential for `async`
- Variance and Subtyping:** Covariant (`&'a T`), invariant (`&mut 'a T`), determines lifetime substitution

Key Takeaways: - Elision rules cover 90%+ cases: let compiler infer when possible - Lifetimes prevent use-after-free: references can't outlive data - Zero runtime cost: lifetimes compile-time only, erased after checking - HRTBs enable flexible closures: `for<'a>` means "for all lifetimes" - `Pin` makes `async` possible: self-referential futures safe when pinned

Lifetime Elision Rules: 1. Each elided lifetime gets distinct parameter: `fn foo(x: &i32) -> fn foo<'a>(x: &'a i32)` 2. Single input lifetime → all output lifetimes: `fn foo(x: &str) -> &str -> fn foo<'a>(x: &'a str) -> &'a str` 3. `&self` lifetime → all output lifetimes: `fn get(&self) -> &T -> fn get(&'a self) -> &'a T`

Variance Rules: - Covariant: `&'a T` accepts longer lifetimes ('long usable where 'short expected') - Invariant: `&mut 'a T` requires exact lifetime (no substitution) - Contravariant: function arguments (rare, opposite direction)

Common Patterns:

```

// Explicit lifetime for multiple parameters
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}

// Struct with lifetime
struct Parser<'a> {
    input: &'a str,
}

```

```

}

// Lifetime bounds
fn process<'a, T: 'a>(data: &'a T) -> &'a T { data }

// Higher-ranked trait bound
fn apply<F>(f: F) where F: for<'a> Fn(&'a str) -> String {
    f("hello");
}

// Variance example
let long: &static str = "hello";
let short: &str = long; // OK: 'static is subtype of shorter

```

Summary

When to Use What: - Elision: Most function signatures (let compiler infer) - Explicit 'a: Multiple references, unclear relationships - Multiple lifetimes: Independent lifetimes (Context<'user, 'session>) - 'static: Program-duration data (string literals, leaked allocations) - T: 'a bounds: Generic types with references - for<'a>: Closures accepting references - Pin: Self-referential types, async futures

Debugging Lifetime Errors: - Read error carefully: compiler explains what outlives what - Draw lifetime diagram: visualize scope of each reference - Simplify: remove complexity to isolate issue - Check elision: explicit annotations when elision fails - Use '_ placeholder: anonymous lifetime for clarity - Compiler suggestions: often provide exact fix

Performance: - Lifetimes: zero runtime cost, compile-time only - Bounds checking: compile-time, no runtime impact - HRTB: monomorphization, no runtime overhead - Pin: zero-cost abstraction when used correctly - Variance: compile-time subtyping rules, no cost

Anti-Patterns: - Overusing 'static (rarely needed, only for program-duration) - Fighting the borrow checker (restructure instead) - Cloning to avoid lifetimes (understand issue first) - Multiple unnecessary lifetimes (use one when possible) - Ignoring compiler suggestions (they're usually correct)

Pattern Matching & Destructuring

Pattern matching enables you to write clear and efficient code for handling complex data structures. Unlike simple switch statements in other languages, Rust's pattern matching provides deep destructuring, guards, bindings, and compile-time exhaustiveness checking.

The key insight is that pattern matching isn't just about control flow. It's a way to encode invariants, state transitions, and data transformations directly in your program's structure.

Pattern Matching Basics

```

// Basic patterns
match value {
    literal => { /* exact match */ }
    _ => { /* wildcard */ }
}

```

```

}

// Destructuring patterns
match tuple {
  (x, y) => { /* bind variables */ }
  (0, _) => { /* ignore parts */ }
}

// Enum patterns
match option {
  Some(x) => { /* extract value */ }
  None => { /* handle absence */ }
}

// Guards and bindings
match value {
  x if x > 0 => { /* conditional */ }
  x @ 1..=10 => { /* range with binding */ }
  _ => { /* default */ }
}

// Reference patterns
match &value {
  &x => { /* dereference */ }
  ref x => { /* create reference */ }
}

// Or patterns and multiple cases
match ch {
  'a' | 'e' | 'i' | 'o' | 'u' => { /* vowel */ }
  _ => { /* consonant */ }
}

```

Pattern 1: Advanced Match Patterns

Problem: Simple if-else chains for numeric ranges become unwieldy (checking temperature ranges requires 8+ nested conditions). Extracting and testing values simultaneously requires separate steps (check status code, then extract it).

Solution: Use range patterns (`1..=10`) for concise numeric matching. Combine `@` bindings with guards to capture values while testing conditions (`x @ 100..=200 if expensive(x)`).

Why It Matters: Range patterns reduce 20 lines of if-else to a single clear match expression. The `@` binding eliminates temporary variables—capture and test in one step.

Use Cases: Numeric classification (temperature ranges, HTTP status codes, port numbers), token parsing (keywords, operators, literals), request routing (method + path combinations), user categorization (age/premium/activity), validation with capture (valid ranges that you need to use).

Example: Range Matching

Range patterns provide a concise way to match against a range of values, which is far more readable than a long chain of `if-else` statements.

```
fn classify_temperature(temp: i32) -> &'static str {
    match temp {
        i32::MIN..=-40 => "extreme cold",
        -39..=-20 => "very cold",
        -19..=0 => "cold",
        1..=15 => "cool",
        16..=25 => "comfortable",
        26..=35 => "warm",
        36..=45 => "hot",
        46..=i32::MAX => "extreme heat",
    }
}
```

Example: Guards for Complex Conditions

Match guards (`if ...`) allow you to add arbitrary boolean expressions to a pattern. This is useful when the condition for a match is more complex than a simple value comparison.

```
struct Response;
struct Error;

fn process_request(status: u16, body: &str) -> Result<Response, Error> {
    match (status, body.len()) {
        (200, len) if len > 0 => Ok(Response), // Guard checks the length
        (200, _) => Err(Error), // Handles 200 but with empty body
        (status @ 400..=499, _) => Err(Error), // Bind status and match range
        (status @ 500..=599, _) => Err(Error),
        _ => Err(Error),
    }
}
```

Example: @ Bindings to Capture and Test

The `@` symbol lets you bind a value to a variable while simultaneously testing it against a more complex pattern. This avoids the need to re-bind the variable inside the match arm.

```
enum Port { WellKnown(u16), Registered(u16), Dynamic(u16) }
struct ValidationError;

fn validate_port(port: u16) -> Result<Port, ValidationError> {
    match port {
        p @ 1..=1023 => Ok(Port::WellKnown(p)), // Bind to `p` and test range
        p @ 1024..=49151 => Ok(Port::Registered(p)),
        p @ 49152..=65535 => Ok(Port::Dynamic(p)),
        _ => Err(ValidationError),
    }
}
```

```
}
```

Example: Nested Destructuring

Pattern matching allows you to “look inside” nested data structures, binding to only the fields you care about and ignoring the rest with ...

```
struct Point { x: i32, y: i32 }
enum Shape {
    Circle { center: Point, radius: f64 },
    Rectangle { top_left: Point, bottom_right: Point },
}

fn contains_origin(shape: &Shape) -> bool {
    match shape {
        // Destructure directly to the inner `x` and `y` fields.
        Shape::Circle { center: Point { x: 0, y: 0 }, .. } => true,
        Shape::Rectangle {
            top_left: Point { x: x1, y: y1 },
            bottom_right: Point { x: x2, y: y2 },
            // A guard can be used with destructured values.
            } if *x1 <= 0 && *x2 >= 0 && *y1 <= 0 && *y2 >= 0 => true,
            _ => false,
    }
}
```

Pattern 2: Exhaustiveness and Match Ergonomics

Problem: In many languages, if you forget to handle a new enum variant in a `switch` statement, the program may compile but crash at runtime. Wildcard branches (`_` or `default`) can silently swallow new variants, leading to logical bugs.

Solution: Rust’s `match` expressions are **exhaustive**, meaning the compiler guarantees that every possible case is handled. If you add a new variant to an enum, your code will not compile until you update all `match` expressions that use it.

Why It Matters: Exhaustiveness checking is one of Rust’s most powerful safety features. It makes refactoring and evolving codebases dramatically safer.

Use Cases: - State machines where new states must be handled correctly everywhere. - Protocol implementations that need to support multiple versions. - Command parsers that must handle every possible command. - API error types, ensuring that callers handle all possible failure modes.

Example: Exhaustive Matching for Safety

If we add a new variant to `RequestState`, the `state_duration` function will no longer compile until the new variant is handled in the `match` expression. This prevents us from forgetting to update critical logic.

```

enum RequestState {
    Pending,
    InProgress { started_at: u64 },
    Completed { result: String },
    Failed { error: String },
}

fn state_duration(state: &RequestState, now: u64) -> Option<u64> {
    match state {
        RequestState::Pending => None,
        RequestState::InProgress { started_at } => Some(now - started_at),
        RequestState::Completed { .. } => None,
        RequestState::Failed { .. } => None,
        // If a new variant, e.g., `Throttled`, were added to `RequestState`,
        // this match would produce a compile-time error.
    }
}

```

Example: Match Ergonomics

Rust's match ergonomics automatically handle references for you in most cases, reducing visual noise. When you match on a `&Option<String>`, the inner value is automatically a `&String`, not a `String`.

```

fn process_option(opt: &Option<String>) {
    match opt {
        // `s` is automatically `&String`, not `String`.
        Some(s) => println!("Got a string: {}", s),
        None => println!("Got nothing."),
    }
}

```

Example: The `#[non_exhaustive]` Attribute

For public libraries, you may want to add new enum variants without it being a breaking change for your users. The `#[non_exhaustive]` attribute tells the compiler that this enum may have more variants in the future, forcing users to include a wildcard (`_`) arm in their `match` expressions.

```

#[non_exhaustive]
pub enum HttpVersion {
    Http11,
    Http2,
}

// A user of this library must include a wildcard match.
fn handle_version(version: HttpVersion) {
    match version {
        HttpVersion::Http11 => println!("Using HTTP/1.1"),
        HttpVersion::Http2 => println!("Using HTTP/2"),
        // This is required. If the library adds `Http3`, this code won't break.
        _ => println!("Using an unknown future version"),
    }
}

```

```
}
```

Exhaustiveness principles:

1. **Avoid wildcards in application code:** Forces you to handle new variants
2. **Use non_exhaustive for public library enums:** Allows adding variants without breaking changes
3. **Leverage compiler errors:** Let the compiler tell you what you forgot to handle
4. **Prefer match over if-let for complex enums:** Makes missing cases obvious
5. **Group similar cases carefully:** Don't hide distinct behavior behind wildcards

Pattern 3: if let, while let, and let-else

Problem: A full `match` expression can be verbose if you only care about one or two cases. Nested `if-let` statements for a sequence of validations can lead to a “pyramid of doom” that is hard to read.

Solution: - Use `if let` to handle a single match case without the boilerplate of a full `match`. - Use `if let` chains (Rust 1.65+) to combine multiple patterns and conditions without nesting.

Why It Matters: These constructs make control flow more ergonomic and readable. `if let` chains flatten complex validation logic.

Use Cases: - Authentication flows (checking a header, parsing a token, validating claims). - Configuration parsing with layered fallbacks. - Processing items from a queue or stream until it is empty. - Navigating nested `Option` or `Result` types.

Example: if let and if let chains

An `if let` chain can replace nested `if let` statements, making validation logic much flatter and easier to read.

```
struct Claims;
struct Token;
struct Request { authorization: Option<String> }

fn parse_token(auth: &str) -> Result<Token, ()> { Ok(Token) }

fn validate_token(token: &Token) -> Result<Claims, ()> { Ok(Claims) }

fn handle_request(req: &Request) {
    // Before if-let chains, this would be nested.
    if let Some(auth) = &req.authorization {
        if let Ok(token) = parse_token(auth) {
            if let Ok(claims) = validate_token(&token) {
                // ... process request with claims
            }
        }
    }
}

// With if-let chains, it's much cleaner.
if let Some(auth) = &req.authorization
    && let Ok(token) = parse_token(auth)
```

```

    && let Ok(claims) = validate_token(&token)
{
    // ... process request with claims
}

```

Example: **let-else** for Early Returns

let-else is perfect for guard clauses at the beginning of a function. It allows you to destructure a value and **return** or **break** if the pattern doesn't match.

```

struct Claims { user_id: u64 }
enum Error { MissingAuth, InvalidToken }

fn get_user_id(request: &Request) -> Result<u64, Error> {
    let Some(auth_header) = &request.authorization else {
        return Err(Error::MissingAuth);
    };

    let Ok(claims) = validate_token(&Token) else {
        return Err(Error::InvalidToken);
    };

    Ok(claims.user_id)
}

```

Example: **while let** for Iteration

while let is the idiomatic way to loop as long as a pattern continues to match. It's often used to process items from an iterator or a queue.

```

use std::collections::VecDeque;

// This loop continues as long as `pop_front` returns `Some(task)` .
fn drain_queue(queue: &mut VecDeque<String>) {
    while let Some(task) = queue.pop_front() {
        println!("Processing task: {}", task);
    }
}

```

If-let and while-let guidelines:

- 1. Use if-let for single pattern:** More concise than match with one arm
- Avoid deep nesting:** Switch to match for complex cases
- Let-else for early returns:** Cleaner than if-let with nested code
- While-let for iterators:** Natural for consuming data structures
- If-let chains for validation sequences:** Replaces nested if-let

Pattern 4: State Machines with Type-State Pattern

Problem: In many architectures, business logic is scattered across different services or classes. Adding a new operation can require hunting through the codebase to make updates.

Solution: Model the core operations, events, and states of your application as **enums with associated data**. Use `match` expressions to implement behavior in a centralized and exhaustive way.

Why It Matters: This architecture centralizes your business logic and leverages Rust's exhaustiveness checking for maintainability. When you add a new command, the compiler forces you to handle it in your `execute` function.

Use Cases: - CQRS (Command Query Responsibility Segregation) systems. - Event sourcing, where application state is derived from a sequence of events. - Message-passing systems, like actor models or service buses. - Defining explicit, type-safe API response structures.

Example: Command Pattern with Enums

Instead of an object-oriented command pattern, you can define all possible operations as variants of a single `Command` enum. A central `execute` method then dispatches on the variant.

```
enum Command {
    CreateUser { username: String, email: String },
    DeleteUser { user_id: u64 },
}

fn execute_command(command: Command) {
    match command {
        Command::CreateUser { username, email } => {
            println!("Creating user {} with email {}", username, email);
        }
        Command::DeleteUser { user_id } => {
            println!("Deleting user {}", user_id);
        }
    }
}
```

Example: Event Sourcing with Enums

In an event-sourcing system, state is rebuilt by applying a series of events. Using an enum for events ensures that every event is a well-defined, typed structure, and that your state aggregates can handle all of them.

```
// All possible events in the system are defined here.
enum UserEvent {
    UserRegistered { user_id: u64, username: String },
    EmailVerified { user_id: u64 },
}

struct User {
    id: u64,
    username: String,
    is_verified: bool,
}

impl User {
```

```

// A user's state is built by applying events in order.
fn from_events(events: &[UserEvent]) -> Self {
    let mut user = User { id: 0, username: "".into(), is_verified: false };
    for event in events {
        user.apply(event);
    }
    user
}

fn apply(&mut self, event: &UserEvent) {
    match event {
        UserEvent::UserRegistered { user_id, username } => {
            self.id = *user_id;
            self.username = username.clone();
        }
        UserEvent::EmailVerified { .. } => {
            self.is_verified = true;
        }
    }
}

```

Destructuring best practices: 1. **Destructure in parameter position:** More concise than extracting in body 2. **Use .. to ignore irrelevant fields:** Makes intent clear 3. **Rename fields for context:** Use @ or field: name syntax 4. **Match slices with patterns:** Handle different lengths explicitly 5. **Be mindful of moves:** Use ref/&mut when needed to avoid consuming values

Summary

Pattern matching in Rust is a powerful tool that goes beyond simple control flow. By leveraging advanced patterns, exhaustiveness checking, state machines, and enum-driven architecture, you can:

- **Encode invariants at compile time:** Make invalid states unrepresentable
- **Eliminate entire classes of bugs:** Exhaustiveness prevents missing cases
- **Express complex logic clearly:** Pattern matching documents behavior
- **Build robust state machines:** Type-state pattern enforces valid transitions
- **Design better APIs:** Enums make interfaces explicit and type-safe

Key takeaways: 1. Use range patterns and guards for expressive numeric matching 2. Leverage exhaustiveness checking—avoid wildcards in application code 3. Apply if-let chains and let-else for ergonomic validation sequences 4. Encode state machines in types to prevent invalid transitions 5. Design subsystems around enums to make illegal states unrepresentable 6. Destructure deeply to extract exactly what you need

Iterator Patterns & Combinators

Iterators are providing a unified interface for processing sequences of data. Unlike loops in many languages, Rust iterators are zero-cost abstractions: they compile down to the same machine code as hand-written loops, yet offer composability, expressiveness, and safety.

The key insight is that iterators aren't just for collections—they're a design pattern for lazy, composable computation that can model streaming algorithms, state machines, and complex data transformations.

Pattern 1: Custom Iterators and `IntoIterator`

Problem: You have a custom data structure (like a tree, graph, or a special-purpose buffer) and you want to allow users to loop over it using a standard `for` loop. Returning a `Vec` of items is inefficient as it requires allocating memory for all items at once.

Solution: Implement the `Iterator` trait for a helper struct that holds the iteration state. Then, implement the `IntoIterator` trait for your main data structure, which creates and returns an instance of your iterator struct.

Why It Matters: This pattern provides a clean, idiomatic, and efficient way to expose the contents of your data structures. Because iterators are lazy, no computation or allocation happens until the caller actually starts consuming items.

Use Cases: - Custom collections like trees, graphs, or ring buffers. - Infinite or procedurally generated sequences (e.g., Fibonacci numbers, prime numbers). - Stateful generators that compute values on the fly. - Adapters for external data sources or APIs.

Example: A Basic Custom Iterator

This `Counter` struct demonstrates the simplest form of a custom iterator. It iterates from 0 up to a maximum value.

```
struct Counter {
    current: u32,
    max: u32,
}

impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        if self.current < self.max {
            let result = self.current;
            self.current += 1;
            Some(result)
        } else {
            None
        }
    }
}

// You can now use `Counter` with any iterator methods.
let sum: u32 = Counter { current: 0, max: 5 }.sum();
assert_eq!(sum, 10); // 0 + 1 + 2 + 3 + 4
```

Example: Implementing IntoIterator

To make a custom collection work with `for` loops, you need to implement `IntoIterator`. Here, we implement it for a `RingBuffer` for owned, borrowed, and mutably borrowed iteration.

```
struct RingBuffer<T> {
    data: Vec<T>,
}

// For `for item in my_buffer` (consumes the buffer)
impl<T> IntoIterator for RingBuffer<T> {
    type Item = T;
    type IntoIter = std::vec::IntoIter<T>;
    fn into_iter(self) -> Self::IntoIter {
        self.data.into_iter()
    }
}

// For `for item in &my_buffer` (borrows the buffer)
impl<'a, T> IntoIterator for &'a RingBuffer<T> {
    type Item = &'a T;
    type IntoIter = std::slice::Iter<'a, T>;
    fn into_iter(self) -> Self::IntoIter {
        self.data.iter()
    }
}
```

Example: An Infinite Iterator

Iterators can represent infinite sequences because they are lazy. This `Fibonacci` iterator will produce numbers forever until the sequence overflows or is stopped by an adapter like `.take()`.

```
struct Fibonacci {
    current: u64,
    next: u64,
}

impl Iterator for Fibonacci {
    type Item = u64;

    fn next(&mut self) -> Option<Self::Item> {
        let result = self.current;
        // Use `checked_add` to handle potential overflow gracefully.
        let new_next = self.current.checked_add(self.next)?;
        self.current = self.next;
        self.next = new_next;
        Some(result)
    }
}

// We can take the first 10 Fibonacci numbers.
```

```
let fibs: Vec<_> = Fibonacci { current: 0, next: 1 }.take(10).collect();
assert_eq!(fibs, vec![0, 1, 1, 2, 3, 5, 8, 13, 21, 34]);
```

Custom iterator guidelines: 1. **Implement size_hint when possible:** Enables optimizations in collect() and other consumers 2. **Use checked arithmetic:** Prevent overflow in stateful iterators 3. **Provide IntoIterator for all three forms:** Owned, borrowed, and mutably borrowed 4. **Extension traits for chainability:** Make custom adapters feel native 5. **Lazy evaluation:** Only compute values when next() is called

Pattern 2: Zero-Allocation Iteration

Problem: When processing large datasets, chaining operations like `map` and `filter` can be inefficient if each step allocates a new, intermediate collection. This can lead to high memory usage and poor cache performance.

Solution: Chain iterator adapters together without calling `.collect()` until the very end. Iterators in Rust are “lazy,” meaning they don’t do any work until a “consuming” method like `collect()`, `sum()`, or `count()` is called.

Why It Matters: This pattern is fundamental to writing high-performance data processing code in Rust. It allows you to write high-level, declarative code that is just as fast as a hand-written, low-level loop.

Use Cases: - Data processing pipelines (e.g., in ETL jobs or data analysis). - Filtering, transforming, and aggregating data from large files or databases. - High-performance code in hot paths, such as in network servers or game engines. - Parsing and stream processing.

Example: Chaining Adapters without Intermediate Collections

This function processes a slice of numbers by filtering positive numbers, squaring them, filtering again, and finally summing the result. No intermediate `Vec` is created.

```
fn process_numbers(input: &[i32]) -> i32 {
    input
        .iter()
        .filter(|&&x| x > 0)
        .map(|&x| x * x)
        .filter(|&x| x < 1000)
        .sum() // The iterator is consumed only at the end.
}
```

Example: Using `windows` for Sliding Window

The `.windows()` method creates an iterator that yields overlapping slices of the original data. This is a zero-allocation way to implement sliding window algorithms.

```
fn moving_average(data: &[f64], window_size: usize) -> Vec<f64> {
    data.windows(window_size)
        .map(|window| window.iter().sum():<f64>() / window_size as f64)
```

```
    .collect()
}
```

Example: `fold` for Custom Reductions

Instead of creating a new collection just to count items, `fold` can be used to perform custom aggregations in a single pass with no allocations.

```
fn count_long_strings(strings: &[&str]) -> usize {
    strings
        .iter()
        .fold(0, |count, &s| if s.len() > 10 { count + 1 } else { count })
}
```

Zero-allocation principles: 1. **Chain adapters instead of collecting:** Each adapter adds minimal overhead 2. **Use `fold/try_fold` for custom reduction:** Avoid `filter().count()` patterns 3. **Leverage from_fn for stateful generation:** No need for custom iterator types 4. **Prefer borrowed iteration:** Use `.iter()` over `.into_iter()` when possible 5. **Iterator::flatten and flat_map:** Flatten nested structures without intermediate Vec

Pattern 3: Advanced Iterator Composition

Problem: Standard iterator adapters cover many use cases, but sometimes you need more specialized logic, such as stateful transformations, lookahead, or custom grouping, without resorting to manual loops.

Solution: Use more advanced adapters like `.scan()`, `.peekable()`, and `.flat_map()` to build complex, declarative data processing pipelines. `.scan()` is perfect for stateful transformations like cumulative sums.

Why It Matters: These advanced tools allow you to express complex logic while staying within the iterator paradigm. This keeps your code declarative, composable, and often more performant than a manual implementation, as the compiler can still optimize the entire iterator chain.

Use Cases: - Log analysis and data transformation pipelines. - Parsers that need lookahead. - Grouping and aggregating data by a key. - Generating cumulative statistics or running totals. - Interleaving or merging multiple data streams.

Example: Complex filtering and transformation pipeline

This iterator chain filters log lines twice and folds the remainder into a per-level counter without building temporary collections.

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

#[derive(Debug, Clone)]
struct LogEntry {
    timestamp: u64,
    level: String,
```

```

        message: String,
    }

fn analyze_logs(logs: &[LogEntry]) -> HashMap<String, usize> {
    logs.iter()
        .filter(|entry| entry.level == "ERROR" || entry.level == "WARN")
        .filter(|entry| entry.timestamp > 1_000_000)
        .map(|entry| &entry.level)
        .fold(HashMap::new(), |mut map, level| {
            *map.entry(level.clone()).or_insert(0) += 1;
            map
        })
}

```

Example: Chunking with stateful iteration

`from_fn` keeps the stateful `Vec` and drains `chunk_size` elements at a time, yielding batches lazily.

```

fn process_in_chunks<T>(items: Vec<T>, chunk_size: usize) -> impl Iterator<Item = Vec<T>> {
    let mut items = items;
    std::iter::from_fn(move || {
        if items.is_empty() {
            None
        } else {
            let drain_end = chunk_size.min(items.len());
            Some(items.drain(..drain_end).collect())
        }
    })
}

```

Example: Interleaving iterators

This custom iterator alternates between two inputs while remaining lazy and falling back to whichever still has data.

```

struct Interleave<I, J> {
    a: I,
    b: J,
    use_a: bool,
}

impl<I, J> Iterator for Interleave<I, J>
where
    I: Iterator,
    J: Iterator<Item = I::Item>,
{
    type Item = I::Item;

    fn next(&mut self) -> Option<Self::Item> {
        if self.use_a {
            self.use_a = false;
            Some(self.a.next())
        } else {
            self.use_a = true;
            Some(self.b.next())
        }
    }
}

```

```

        self.a.next().or_else(|| self.b.next())
    } else {
        self.use_a = true;
        self.b.next().or_else(|| self.a.next())
    }
}

fn interleave<I, J>(a: I, b: J) -> Interleave<I::IntoIter, J::IntoIter>
where
    I: IntoIterator,
    J: IntoIterator<Item = I::Item>,
{
    Interleave {
        a: a.into_iter(),
        b: b.into_iter(),
        use_a: true,
    }
}

```

Example: Cartesian product

`flat_map` composes the nested iteration so every pair of elements from the two slices is produced without intermediate vectors.

```

fn cartesian_product<T: Clone>(a: &[T], b: &[T]) -> impl Iterator<Item = (T, T)> + '_ {
    a.iter()
        .flat_map(move |x| b.iter().map(move |y| (x.clone(), y.clone())))
}

```

Example: Group by key

One pass and a `HashMap` are enough to accumulate elements into key buckets defined by any closure.

```

use std::collections::HashMap;

fn group_by<K, V, F>(items: Vec<V>, key_fn: F) -> HashMap<K, Vec<V>>
where
    K: Eq + std::hash::Hash,
    F: Fn(&V) -> K,
{
    items.into_iter().fold(HashMap::new(), |mut map, item| {
        map.entry(key_fn(&item)).or_insert_with(Vec::new).push(item);
        map
    })
}

```

Example: Scan for cumulative operations

`.scan` threads mutable state to emit a running total as each value is consumed.

```

fn cumulative_sum(numbers: &[i32]) -> Vec<i32> {
    numbers
        .iter()
        .scan(0, |state, &x| {
            *state += x;
            Some(*state)
        })
        .collect()
}

```

Example: Take while and skip while for prefix/suffix operations

Two iterator adapters split a slice of lines at the first blank row, yielding header and body views without copying.

```

fn extract_header_body(lines: &[String]) -> (Vec<&String>, Vec<&String>) {
    let header: Vec<_> = lines.iter().take_while(|line| !line.is_empty()).collect();
    let body: Vec<_> = lines
        .iter()
        .skip_while(|line| !line.is_empty())
        .skip(1)
        .collect();
    (header, body)
}

```

Example: Peekable for lookahead

Wrapping `chars()` in `peekable()` enables a tokenizer that can inspect the next character before deciding how to advance.

```

fn parse_tokens(input: &str) -> Vec<Token> {
    let mut chars = input.chars().peekable();
    let mut tokens = Vec::new();

    while let Some(&ch) = chars.peek() {
        match ch {
            '0'..='9' => {
                let num: String = chars
                    .by_ref()
                    .take_while(|c| c.is_ascii_digit())
                    .collect();
                tokens.push(Token::Number(num.parse().unwrap()));
            }
            '+' | '-' | '*' | '/' => {
                tokens.push(Token::Op(ch));
                chars.next();
            }
            _ => {
                chars.next();
            }
        }
    }
}

```

```

        _ => {
            chars.next();
        }
    }

    tokens
}

#[derive(Debug)]
enum Token {
    Number(i32),
    Op(char),
}

```

Adapter composition principles: 1. **Build pipelines incrementally:** Each step should be independently testable 2. **Use scan for stateful transformations:** More expressive than manual state tracking 3. **Peekable for lookahead:** Essential for parsers and state machines 4. **Prefer iterator methods over manual loops:** More composable and optimizable 5. **fold for custom aggregations:** More flexible than sum/collect

Pattern 4: Streaming Algorithms

Problem: Loading entire files or datasets into memory causes OOM errors with large data (multi-GB log files, database dumps). Computing statistics requires multiple passes over data, multiplying I/O cost.

Solution: Process data one element at a time using iterators, maintaining only essential state (aggregates, sliding windows, top-K heaps). Use `BufReader::lines()` for line-by-line file processing.

Why It Matters: Streaming algorithms enable processing datasets larger than RAM—a 100GB log file processes with constant 1MB memory. Single-pass algorithms are dramatically faster: computing average + variance in one pass vs two passes halves I/O time.

Use Cases: Log file analysis (grep-like filtering, statistics), database ETL (processing query results), real-time analytics (streaming averages, alerting), sensor data processing, network packet analysis, infinite sequences (event streams, live data feeds), CSV/JSON parsing of large files.

Example: Line-by-line file processing

`BufReader::lines` streams through a file lazily, letting you filter and count matches without loading the whole file.

```

use std::fs::File;
use std::io::{BufRead, BufReader};

fn count_lines_matching(path: &str, pattern: &str) -> std::io::Result<usize> {
    let file = File::open(path)?;
    let reader = BufReader::new(file);

    Ok(reader

```

```

    .lines()
    .filter_map(Result::ok)
    .filter(|line| line.contains(pattern))
    .count()
}

}

```

Example: Streaming average calculation

A lightweight accumulator keeps just the sum and count so you can compute an average in one pass over any iterator.

```

struct StreamingAverage {
    sum: f64,
    count: usize,
}

impl StreamingAverage {
    fn new() -> Self {
        StreamingAverage { sum: 0.0, count: 0 }
    }

    fn add(&mut self, value: f64) {
        self.sum += value;
        self.count += 1;
    }

    fn average(&self) -> Option<f64> {
        if self.count == 0 {
            None
        } else {
            Some(self.sum / self.count as f64)
        }
    }
}

fn compute_streaming_average(numbers: impl Iterator<Item = f64>) -> Option<f64> {
    let mut avg = StreamingAverage::new();
    for num in numbers {
        avg.add(num);
    }
    avg.average()
}

```

Example: Top-K elements without sorting

A `BinaryHeap` tracks just the best `k` candidates, avoiding a full sort regardless of input size.

```

use std::cmp::Reverse;
use std::collections::BinaryHeap;

fn top_k<T: Ord>(iter: impl Iterator<Item = T>, k: usize) -> Vec<T> {

```

```

let mut heap = BinaryHeap::new();

for item in iter {
    if heap.len() < k {
        heap.push(Reverse(item));
    } else if let Some(&Reverse(ref min)) = heap.peek() {
        if &item > min {
            heap.pop();
            heap.push(Reverse(item));
        }
    }
}

heap.into_iter().map(|Reverse(x)| x).collect()
}

```

Example: Sliding window statistics

A reusable `SlidingWindow` struct with a `VecDeque` tracks the latest `window_size` values and emits sums as it slides forward.

```

struct SlidingWindow<T> {
    window: std::collections::VecDeque<T>,
    capacity: usize,
}

impl<T> SlidingWindow<T> {
    fn new(capacity: usize) -> Self {
        SlidingWindow {
            window: std::collections::VecDeque::with_capacity(capacity),
            capacity,
        }
    }

    fn push(&mut self, value: T) -> Option<T> {
        if self.window.len() == self.capacity {
            let removed = self.window.pop_front();
            self.window.push_back(value);
            removed
        } else {
            self.window.push_back(value);
            None
        }
    }

    fn iter(&self) -> impl Iterator<Item = &T> {
        self.window.iter()
    }
}

fn sliding_window_sum(
    numbers: impl Iterator<Item = i32>,

```

```

    window_size: usize,
) -> impl Iterator<Item = i32> {
    let mut window = SlidingWindow::new(window_size);
    let mut sum = 0;
    let mut initialized = false;

    numbers.filter_map(move |num| {
        if let Some(old) = window.push(num) {
            sum = sum - old + num;
            Some(sum)
        } else {
            sum += num;
            if window.window.len() == window_size {
                initialized = true;
            }
            if initialized {
                Some(sum)
            } else {
                None
            }
        }
    })
}
}

```

Example: Streaming deduplication

A `HashSet` remembers which values have been seen so duplicates are filtered out lazily as the iterator advances.

```

use std::collections::HashSet;

fn deduplicate_stream<T>(iter: impl Iterator<Item = T>) -> impl Iterator<Item = T>
where
    T: Eq + std::hash::Hash + Clone,
{
    let mut seen = HashSet::new();
    iter.filter(move |item| seen.insert(item.clone()))
}

```

Example: Rate limiting iterator

`RateLimited` wraps any iterator and sleeps as needed so successive items are yielded no faster than the configured interval.

```

use std::time::{Duration, Instant};

struct RateLimited<I> {
    iter: I,
    interval: Duration,
    last_yield: Option<Instant>,
}

```

```

impl<I: Iterator> RateLimited<I> {
    fn new(iter: I, interval: Duration) -> Self {
        RateLimited {
            iter,
            interval,
            last_yield: None,
        }
    }
}

impl<I: Iterator> Iterator for RateLimited<I> {
    type Item = I::Item;

    fn next(&mut self) -> Option<Self::Item> {
        if let Some(last) = self.last_yield {
            let elapsed = last.elapsed();
            if elapsed < self.interval {
                std::thread::sleep(self.interval - elapsed);
            }
        }

        let item = self.iter.next()?;
        self.last_yield = Some(Instant::now());
        Some(item)
    }
}

```

Example: Buffered batch processing

Accumulate elements in a buffer and flush them to a callback once `batch_size` is reached, reducing per-item overhead.

```

fn process_in_batches<T, F>(
    iter: impl Iterator<Item = T>,
    batch_size: usize,
    mut process_batch: F,
) where
    F: FnMut(Vec<T>),
{
    let mut batch = Vec::with_capacity(batch_size);

    for item in iter {
        batch.push(item);
        if batch.len() == batch_size {
            process_batch(std::mem::replace(
                &mut batch,
                Vec::with_capacity(batch_size),
            ));
        }
    }
}

```

```

    if !batch.is_empty() {
        process_batch(batch);
    }
}

```

Example: Streaming merge of sorted iterators

Two ordered inputs are merged into a new iterator by manually peeking at the next value from each source.

```

fn merge_sorted<T: Ord>(
    mut a: impl Iterator<Item = T>,
    mut b: impl Iterator<Item = T>,
) -> impl Iterator<Item = T> {
    let mut a_next = a.next();
    let mut b_next = b.next();

    std::iter::from_fn(move || match (&a_next, &b_next) {
        (Some(a_val), Some(b_val)) => {
            if a_val <= b_val {
                let result = a_next.take();
                a_next = a.next();
                result
            } else {
                let result = b_next.take();
                b_next = b.next();
                result
            }
        }
        (Some(_), None) => {
            let result = a_next.take();
            a_next = a.next();
            result
        }
        (None, Some(_)) => {
            let result = b_next.take();
            b_next = b.next();
            result
        }
        (None, None) => None,
    })
}

```

Example: CSV parsing with streaming

Each CSV line is read, split, and trimmed on the fly so no intermediate representation is needed.

```

use std::io::BufRead;

fn parse_csv_stream(reader: impl BufRead) -> impl Iterator<Item = Vec<String>> {
    reader.lines().filter_map(Result::ok).map(|line| {

```

```

        line.split(',')
            .map(|s| s.trim().to_string())
            .collect()
    })
}

```

Example: Lazy transformation chain

This pipeline reads a file lazily, normalizes each line, and folds it into a frequency map without ever materializing the whole dataset.

```

use std::collections::HashMap;
use std::fs::File;
use std::io::{BufRead, BufReader};

fn process_large_file(path: &str) -> std::io::Result<Vec<(String, usize)>> {
    let file = File::open(path)?;
    let reader = BufReader::new(file);

    let result = reader
        .lines()
        .filter_map(Result::ok)
        .filter(|line| !line.is_empty())
        .map(|line| line.to_lowercase())
        .flat_map(|line| {
            line.split_whitespace()
                .map(|s| s.to_string())
                .collect::<Vec<_>>()
        })
        .fold(HashMap::new(), |mut map, word| {
            *map.entry(word).or_insert(0) += 1;
            map
        })
        .into_iter()
        .collect();

    Ok(result)
}

```

Pattern 4: Parallel Iteration with Rayon

Problem: Processing a large collection sequentially can be slow, leaving multiple CPU cores idle. Manually writing multi-threaded code to parallelize such tasks is complex, error-prone, and requires careful synchronization.

Solution: Use the **Rayon** library. By simply changing `.iter()` to `.par_iter()`, Rayon can automatically parallelize your iterator chain across all available CPU cores.

Why It Matters: Rayon offers a massive performance boost for data-parallel tasks with minimal code changes. For a CPU-bound task, you can often achieve a near-linear speedup with the number of cores.

Use Cases: - Data-intensive computations like image processing or scientific simulations. - Batch processing of large numbers of files or database records. - Sorting and searching very large datasets. - Any “embarrassingly parallel” task that can be broken down into independent chunks.

Example: Basic Parallel Iteration

By changing `.iter()` to `.par_iter()`, this function becomes a parallel operation. Rayon handles all the complexity of threading and data distribution.

```
use rayon::prelude::*;

// This function will run in parallel across multiple cores.
fn parallel_sum_of_squares(numbers: &[i64]) -> i64 {
    numbers
        .par_iter() // The only change needed for parallelization!
        .map(|&x| x * x)
        .sum()
}
```

Example: Parallel sort

`par_sort_unstable` drops into Rayon’s divide-and-conquer sorter so large vectors are sorted across all cores.

```
use rayon::prelude::*;

fn parallel_sort(mut data: Vec<i32>) -> Vec<i32> {
    data.par_sort_unstable();
    data
}
```

Example: Parallel chunked processing

`par_chunks` divides the slice evenly and lets Rayon process each chunk independently before collecting the results.

```
use rayon::prelude::*;

fn parallel_chunk_processing(data: &[u8], chunk_size: usize) -> Vec<u32> {
    data.par_chunks(chunk_size)
        .map(|chunk| chunk.iter().map(|&b| b as u32).sum())
        .collect()
}
```

Example: Parallel file processing

Each path is read and counted concurrently so filesystem-bound workloads keep all cores busy.

```
use rayon::prelude::*;

fn parallel_process_files(paths: &[String]) -> Vec<u32> {
    paths
        .par_iter()
        .map(|path| {
            std::fs::read_to_string(path)
                .map(|content| content.lines().count())
                .unwrap_or(0)
        })
        .collect()
}
```

Example: Parallel find (early exit)

`position_any` searches in parallel and returns as soon as any worker hits the desired value.

```
use rayon::prelude::*;

fn parallel_find_first(numbers: &[i32], target: i32) -> Option<u32> {
    numbers.par_iter().position_any(|&x| x == target)
}
```

Example: Parallel fold with combiner

Map each line to a count in parallel, then use Rayon's `sum` combiner to aggregate in a thread-safe way.

```
use rayon::prelude::*;

fn parallel_word_count(lines: &[String]) -> u32 {
    lines
        .par_iter()
        .map(|line| line.split_whitespace().count())
        .sum()
}
```

Example: Parallel partition

`partition` splits even and odd numbers in one pass using `into_par_iter` for ownership and parallelism.

```
use rayon::prelude::*;

fn parallel_partition(numbers: Vec<i32>) -> (Vec<i32>, Vec<i32>) {
    numbers.into_par_iter().partition(|&x| x % 2 == 0)
}
```

Example: Parallel nested iteration with `flat_map`

Precompute the columns of `b` and use `par_iter` to multiply each row independently for cache-friendly matrix multiplication.

```
use rayon::prelude::*;

fn parallel_matrix_multiply(a: &[Vec<f64>], b: &[Vec<f64>]) -> Vec<Vec<f64>> {
    let b_cols: Vec<Vec<f64>> = (0..b[0].len())
        .map(|col| b.iter().map(|row| row[col]).collect())
        .collect();

    a.par_iter()
        .map(|row| {
            b_cols
                .iter()
                .map(|col| row.iter().zip(col.iter()).map(|(a, b)| a * b).sum())
                .collect()
        })
        .collect()
}
```

Example: Parallel bridge for converting sequential to parallel

`par_bridge` consumes a standard iterator (here, an MPSC receiver) and fans it into Rayon workers.

```
use rayon::prelude::*;
use std::sync::mpsc::channel;

fn parallel_bridge_example() {
    let (sender, receiver) = channel();

    std::thread::spawn(move || {
        for i in 0..1000 {
            sender.send(i).unwrap();
        }
    });

    let sum: i32 = receiver.into_iter().par_bridge().map(|x| x * x).sum();
    println!("Sum: {}", sum);
}
```

Example: Controlling parallelism with scope

`rayon::scope` lets you spawn child tasks over disjoint slices while borrowing data safely.

```
fn parallel_with_scope(data: &mut [i32]) {
    rayon::scope(|s| {
        let mid = data.len() / 2;
        let (left, right) = data.split_at_mut(mid);
```

```

        s.spawn(|_| {
            for x in left.iter_mut() {
                *x *= 2;
            }
        });

        s.spawn(|_| {
            for x in right.iter_mut() {
                *x *= 3;
            }
        });
    });
}

```

Example: Parallel map-reduce pattern

`fold` builds per-thread hash maps that `reduce` later merges, avoiding contention while counting words.

```

use rayon::prelude::*;
use std::collections::HashMap;

fn parallel_map_reduce(data: &[String]) -> HashMap<String, usize> {
    data.par_iter()
        .fold(
            || HashMap::new(),
            |mut map, line| {
                for word in line.split_whitespace() {
                    *map.entry(word.to_string()).or_insert(0) += 1;
                }
                map
            },
        )
        .reduce(
            || HashMap::new(),
            |mut a, b| {
                for (key, count) in b {
                    *a.entry(key).or_insert(0) += count;
                }
                a
            },
        ),
}

```

Example: Parallel pipeline with multiple stages

Chaining `map`, `filter`, and another `map` on a `par_iter` builds a streaming parallel pipeline.

```

use rayon::prelude::*;

fn parallel_pipeline(data: &[i32]) -> Vec<i32> {

```

```

    data.par_iter()
        .map(|&x| x * 2)      // Stage 1: multiply
        .filter(|&x| x > 100) // Stage 2: filter
        .map(|x| x / 3)       // Stage 3: divide
        .collect()
}

```

Example: Custom parallel iterator

A bespoke range type can later implement Rayon traits to expose fine-grained control over splitting behavior.

```

struct ParallelRange {
    start: usize,
    end: usize,
}

impl ParallelRange {
    fn new(start: usize, end: usize) -> Self {
        ParallelRange { start, end }
    }
}

```

Example: Joining parallel computations

`rayon::join` runs two closures concurrently and returns both results, ideal for independent aggregations.

```

use rayon::prelude::*;

fn parallel_join_example(data: &[i32]) -> (i32, i32) {
    let (sum, product) = rayon::join(
        || data.par_iter().sum(),
        || data.par_iter().product(),
    );
    (sum, product)
}

```

Parallel iteration principles: 1. **Use `par_iter` for parallel iteration:** Drop-in replacement for `.iter()`

2. **Automatic work stealing:** Rayon balances load across threads 3. **fold + reduce for aggregation:**

Parallel-friendly accumulation 4. **Chunk size matters:** Use `par_chunks` for better cache locality 5.

Measure before parallelizing: Overhead can exceed benefits for small datasets

Key takeaways:

1. Implement custom iterators for domain-specific iteration patterns
2. Chain adapters instead of collecting intermediate results
3. Use `fold`/`scan`/`try_fold` for stateful transformations
4. Stream data with `BufReader` and lazy iterators for large files

5. Apply rayon's `par_iter` for automatic parallelization
6. Prefer iterator methods over manual loops for compositability

Error Handling Architecture

Unlike exceptions in languages like Java or Python, Rust uses explicit return types (`Result<T, E>`) to force handling of errors at compile time. This approach eliminates entire classes of bugs: forgotten error checks, unexpected exception propagation, and unclear error boundaries.

The key insight is that error handling in Rust is not just about reporting failures—it's about encoding your program's error domain in the type system, making impossible states unrepresentable, and providing excellent diagnostics when things go wrong.

Pattern 1: Custom Error Enums for Libraries

Problem: Returning simple strings or a generic `Box<dyn Error>` from a library is not ideal. A `String` error loses all type information, making it impossible for a caller to programmatically handle different kinds of failures.

Solution: Define a custom `enum` for your library's errors. Each variant of the enum represents a distinct failure mode.

Why It Matters: A custom error enum makes your library's API transparent and robust. It allows users to `match` on specific error variants and handle them appropriately—for example, retrying a `Timeout` error but aborting on a `PermissionDenied` error.

Use Cases: - Public libraries where callers need to distinguish between different failure modes.
- Systems with complex or domain-specific errors, such as network protocols, database clients, or parsers.
- Any situation where an error is an expected part of the program's control flow.

Example: A Simple Custom Error Enum

This shows a basic error enum for a parsing function. Each variant represents a specific reason the parsing could fail.

```
use thiserror::Error;

#[derive(Error, Debug, PartialEq)]
pub enum ParseError {
    #[error("the input was empty")]
    EmptyInput,
    #[error("the input format was invalid")]
    InvalidFormat,
    #[error("the number was too large")]
    NumberTooLarge,
}

fn parse_number(input: &str) -> Result<i32, ParseError> {
    if input.is_empty() {
```

```

        return Err(ParseError::EmptyInput);
    }

    input.parse().map_err(|_| ParseError::InvalidFormat)
}

#[derive(Error, Debug)]
pub enum DataError {
    #[error("failed to read data")]
    Io(#[from] io::Error),
    #[error("failed to parse number")]
    Parse(#[from] ParseIntError),
}
}

fn load_and_parse_number(path: &str) -> Result<i32, DataError> {
    // The `?` operator will automatically convert `io::Error` and `ParseIntError`
    // into `DataError` because of the `#[from]` attributes.
    let content = std::fs::read_to_string(path)?;
    let number = content.trim().parse()?;
    Ok(number)
}

```

Example: `#[non_exhaustive]` for Library Stability

When publishing a library, you may want to add new error variants in the future without it being a breaking change. The `#[non_exhaustive]` attribute tells users of your library that they must include a wildcard `_` arm in their match statements, ensuring their code won't break if you add a new variant.

```

#[non_exhaustive]
#[derive(Debug, Error)]
pub enum ApiError {
    #[error("the network request failed")]
    NetworkError,
    #[error("the request timed out")]
    Timeout,
    // A new variant could be added here in a future version.
}

```

Pattern 2: `anyhow` for Application-Level Errors

Problem: In application code (as opposed to library code), you often don't need to handle each specific error type. Your main goal is to understand *why* an operation failed and report it to the user or a logging service.

Solution: Use the `anyhow` crate. `anyhow::Result` is a type alias for `Result<T, anyhow::Error>`, where `anyhow::Error` is a dynamic error type that can hold any error that implements `std::error::Error`.

Why It Matters: `anyhow` provides the convenience of a single, easy-to-use error type for your application while preserving the full chain of underlying causes. It strikes a balance between ease of use and detailed diagnostics, which is perfect for the top levels of an application.

Use Cases: - The main logic of CLI tools, web servers, and other applications. - Any situation where you care more about the error *message* and *context* than the specific error *type*. - Prototyping and writing examples where detailed error handling is not the main focus.

Example: Using `anyhow::Result` and `Context`

This function shows how `anyhow` can be used to handle errors from different libraries (`std::fs` and `serde_json`) and add context to them.

```
use anyhow::{Context, Result};

struct Config;

//=====
// Pattern: Structured error with multiple fields
//=====

#[derive(Error, Debug)]
#[error("HTTP request failed: {method} {url} (status: {status})")]
pub struct HttpError {
    pub method: String,
    pub url: String,
    pub status: u16,
    pub body: Option<String>,
    #[source]
    pub source: Option<Box<dyn std::error::Error + Send + Sync>>,
}

//=====
// Pattern: Error builder for complex errors
//=====

pub struct HttpErrorBuilder {
    method: String,
    url: String,
    status: u16,
    body: Option<String>,
    source: Option<Box<dyn std::error::Error + Send + Sync>>,
}

impl HttpErrorBuilder {
    pub fn new(method: impl Into<String>, url: impl Into<String>, status: u16) -> Self {
        HttpErrorBuilder {
            method: method.into(),
            url: url.into(),
            status,
            body: None,
            source: None,
        }
    }
}
```

```

pub fn body(mut self, body: String) -> Self {
    self.body = Some(body);
    self
}

pub fn source(mut self, err: Box) -> Self {
    self.source = Some(err);
    self
}

pub fn build(self) -> HttpError {
    HttpError {
        method: self.method,
        url: self.url,
        status: self.status,
        body: self.body,
        source: self.source,
    }
}
}

```

Error type design principles: 1. **Specific variants:** Each error variant represents a distinct failure mode 2. **Include context:** Path, line number, user input that caused error 3. **Chain sources:** Preserve underlying errors via `source()` 4. **Display for users, Debug for developers:** Display should be readable, Debug should be complete 5. **Non-exhaustive for libraries:** Allow adding variants without breaking changes 6. **Derive when possible:** Use `thiserror` to reduce boilerplate

Library vs Application errors: - **Libraries:** Specific error types, no context loss, caller decides handling - **Applications:** Opaque errors (anyhow), focus on diagnostics, fail fast

Pattern 2: Error Propagation Strategies

Problem: Explicit error handling with `match` and `if let` at every fallible call creates deeply nested code and obscures business logic. Transforming errors manually (wrapping `io::Error` in your `AppError`) is repetitive.

Solution: Use the `?` operator for concise error propagation—it early-returns `Err` and unwraps `Ok`. Implement `From` trait to enable automatic error conversion with `?`.

Why It Matters: The `?` operator reduces error handling from 5+ lines per call to a single character, making error paths as readable as success paths. Automatic error conversion via `From` eliminates boilerplate while preserving type safety.

Use Cases: Application code with mixed error types (I/O, parsing, validation), batch processing that needs to collect all errors, network code requiring retries, operations that can fall back to alternatives, data pipelines with lenient error handling.

Examples

```
//=====
// Pattern: Basic error propagation with ?
//=====

fn read_username(path: &str) -> Result<String, std::io::Error> {
    let content = std::fs::read_to_string(path)?; // Returns early on error
    let username = content.trim().to_string();
    Ok(username)
}

//=====
// Pattern: Error type conversion with ?
//=====

#[derive(Error, Debug)]
enum AppError {
    #[error("IO error")]
    Io(#[from] std::io::Error),

    #[error("Parse error")]
    Parse(#[from] ParseError),
}

fn process_file(path: &str) -> Result<i32, AppError> {
    let content = std::fs::read_to_string(path)?; // Converts io::Error to AppError
    let number = parse_number(&content)?;           // Converts ParseError to AppError
    Ok(number)
}

//=====
// Pattern: Manual error mapping
//=====

fn read_and_validate(path: &str) -> Result<String, IoError> {
    std::fs::read_to_string(path)
        .map_err(|e| IoError::ReadFailed {
            path: path.to_string(),
            source: e,
        })?;

    Ok("valid".to_string())
}

//=====
// Pattern: Fallible iterator processing
//=====

fn parse_all_numbers(lines: Vec<&str>) -> Result<Vec<i32>, ParseError> {
    lines
        .into_iter()
        .map(parse_number)
        .collect() // Collects Result<Vec<T>, E> - stops at first error
}
```

```

//=====
// Pattern: Collect successes, log failures
//=====

fn parse_all(lenient: Vec<&str>) -> Vec<i32> {
    lenient
        .into_iter()
        .filter_map(|line| {
            parse_number(line)
                .map_err(|e| eprintln!("Failed to parse '{}': {}", line, e))
                .ok()
        })
        .collect()
}

//=====
// Pattern: Early return with multiple error types
//=====

fn complex_operation(path: &str) -> Result<String, anyhow::Error> {
    let data = std::fs::read_to_string(path)
        .context("Failed to read input file")?;

    let parsed: Config = serde_json::from_str(&data)
        .context("Failed to parse JSON")?;

    validate_config(&parsed)
        .context("Config validation failed")?;

    Ok("success".to_string())
}

fn validate_config(_config: &Config) -> Result<(), ValidationError> {
    Ok(())
}

#[derive(Error, Debug)]
#[error("validation error")]
struct ValidationError;

//=====
// Pattern: Recovering from specific errors
//=====

fn read_or_default(path: &str) -> Result<String, std::io::Error> {
    match std::fs::read_to_string(path) {
        Ok(content) => Ok(content),
        Err(e) if e.kind() == std::io::ErrorKind::NotFound => {
            Ok("default config".to_string())
        }
        Err(e) => Err(e),
    }
}

//=====
// Pattern: Retry logic with error inspection

```

```

//=====
fn retry_on_timeout<F, T, E>(mut f: F, max_attempts: usize) -> Result<T, E>
where
    F: FnMut() -> Result<T, E>,
    E: std::fmt::Display,
{
    let mut attempts = 0;
    loop {
        attempts += 1;
        match f() {
            Ok(value) => return Ok(value),
            Err(e) if attempts < max_attempts => {
                eprintln!("Attempt {} failed: {}, retrying...", attempts, e);
                std::thread::sleep(std::time::Duration::from_millis(100));
            }
            Err(e) => return Err(e),
        }
    }
}

//=====
// Pattern: Error context accumulation
//=====

fn process_with_context(path: &str) -> Result<i32> {
    let content = std::fs::read_to_string(path)
        .with_context(|| format!("Failed to read file: {}", path))?;

    let number = parse_number(&content)
        .with_context(|| format!("Failed to parse content from {}", path))?;

    if number < 0 {
        anyhow::bail!("Number must be positive, got: {}", number);
    }

    Ok(number)
}

//=====
// Pattern: Partition results into successes and failures
//=====

fn partition_results<T, E>(results: Vec<Result<T, E>>) -> (Vec<T>, Vec<E>) {
    results.into_iter().partition_map(|r| match r {
        Ok(v) => itertools::Either::Left(v),
        Err(e) => itertools::Either::Right(e),
    })
}

use itertools::Itertools;

```

Propagation strategies: - **Immediate propagation** (?): Most common, fail fast - **Map errors**: Add context before propagating - **Collect errors**: Accumulate multiple failures - **Recover**: Handle specific

errors, propagate others - **Retry**: Attempt operation multiple times - **Log and continue**: Record failure but don't propagate

When to use each: - Libraries: Specific error types, minimal context - Applications: Rich context (anyhow), helpful diagnostics - Batch processing: Collect all errors - Network operations: Retry with backoff

Pattern 3: Custom Error Types with Context

Problem: Generic errors like "parse error" or "database query failed" provide no actionable information. Was it line 47 or line 1832?

Solution: Enrich errors with context at the point of failure using `anyhow::Context` for applications or structured fields in custom error types for libraries. Include: what operation failed, what input caused it, where in the input (line/column for parsers, row for databases), timing information, suggestions for fixing.

Why It Matters: "Parse error at line 847, column 23: expected '}', got EOF. Suggestion: check for unclosed braces" points directly to the bug.

Use Cases: Parsers and compilers (provide line/column and code snippet), configuration validation (suggest valid values), database operations (include query and parameters), file I/O (include paths and operations attempted), network requests (include URL, method, status).

Examples

```
//=====
// Pattern: Error with location tracking
//=====

#[derive(Error, Debug)]
#[error("Parse error at line {line}, column {column}: {message}")]
pub struct ParseErrorWithLocation {
    pub line: usize,
    pub column: usize,
    pub message: String,
    pub snippet: Option<String>,
}

impl ParseErrorWithLocation {
    pub fn new(line: usize, column: usize, message: String) -> Self {
        ParseErrorWithLocation {
            line,
            column,
            message,
            snippet: None,
        }
    }

    pub fn with_snippet(mut self, snippet: String) -> Self {
        self.snippet = Some(snippet);
        self
    }
}
```

```

    }

//=====
// Pattern: Error with stack trace (using backtrace)
//=====

#[derive(Debug)]
pub struct DetailedError {
    message: String,
    context: Vec<String>,
    backtrace: std::backtrace::Backtrace,
}

impl DetailedError {
    pub fn new(message: impl Into<String>) -> Self {
        DetailedError {
            message: message.into(),
            context: Vec::new(),
            backtrace: std::backtrace::Backtrace::capture(),
        }
    }

    pub fn with_context(mut self, ctx: impl Into<String>) -> Self {
        self.context.push(ctx.into());
        self
    }
}

impl std::fmt::Display for DetailedError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "{}", self.message)?;
        for ctx in &self.context {
            write!(f, "\n  in {}", ctx)?;
        }
        write!(f, "\n\nBacktrace:\n{}", self.backtrace)?;
        Ok(())
    }
}

impl std::error::Error for DetailedError {}

//=====
// Pattern: Error with structured metadata
//=====

#[derive(Error, Debug)]
#[error("Database query failed")]
pub struct QueryError {
    pub query: String,
    pub duration_ms: u64,
    pub row_count: Option<usize>,
    pub parameters: Vec<String>,
    #[source]
    pub source: Box<dyn std::error::Error + Send + Sync>,
}

```

```

}

impl QueryError {
    pub fn display_detailed(&self) -> String {
        format!(
            "Query failed after {}ms\nQuery: {}\nParameters: {:?}\nError: {}",
            self.duration_ms, self.query, self.parameters, self.source
        )
    }
}

//=====
// Pattern: Context wrapper
//=====

pub struct ErrorContext<E> {
    error: E,
    context: Vec<String>,
}

impl<E: std::error::Error> ErrorContext<E> {
    pub fn new(error: E) -> Self {
        ErrorContext {
            error,
            context: Vec::new(),
        }
    }

    pub fn add_context(mut self, ctx: impl Into<String>) -> Self {
        self.context.push(ctx.into());
        self
    }
}

impl<E: std::error::Error> std::fmt::Display for ErrorContext<E> {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        writeln!(f, "{}", self.error)?;
        for ctx in &self.context {
            writeln!(f, "  Context: {}", ctx)?;
        }
        Ok(())
    }
}

impl<E: std::error::Error + 'static> std::error::Error for ErrorContext<E> {
    fn source(&self) -> Option<&(dyn std::error::Error + 'static)> {
        Some(&self.error)
    }
}

//=====
// Pattern: Error with suggestions
//=====

#[derive(Error, Debug)]

```

```

pub enum ConfigError {
    #[error("Missing required field: {field}\n Suggestion: {suggestion}")]
    MissingField { field: String, suggestion: String },

    #[error("Invalid value for {field}: {value}\n Expected: {expected}")]
    InvalidValue {
        field: String,
        value: String,
        expected: String,
    },
}

impl ConfigError {
    pub fn missing_field(field: impl Into<String>) -> Self {
        let field = field.into();
        let suggestion = match field.as_str() {
            "database_url" => "Add DATABASE_URL to your .env file".to_string(),
            "api_key" => "Set API_KEY environment variable".to_string(),
            _ => format!("Add {} to configuration", field),
        };
        ConfigError::MissingField { field, suggestion }
    }
}

//=====
// Pattern: Error aggregation
//=====

#[derive(Error, Debug)]
#[error("Multiple errors occurred")]
pub struct MultiError {
    errors: Vec<Box<dyn std::error::Error + Send + Sync>>,
}

impl MultiError {
    pub fn new() -> Self {
        MultiError { errors: Vec::new() }
    }

    pub fn add(&mut self, error: Box<dyn std::error::Error + Send + Sync>) {
        self.errors.push(error);
    }

    pub fn is_empty(&self) -> bool {
        self.errors.is_empty()
    }

    pub fn into_result<T>(self, value: T) -> Result<T, Self> {
        if self.errors.is_empty() {
            Ok(value)
        } else {
            Err(self)
        }
    }
}

```

```

}

impl std::fmt::Display for MultiError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        writeln!(f, "Multiple errors occurred ({}):", self.errors.len())?;
        for (i, err) in self.errors.iter().enumerate() {
            writeln!(f, "  {}. {}", i + 1, err)?;
        }
        Ok(())
    }
}

// Usage
fn validate_all(items: Vec<Item>) -> Result<(), MultiError> {
    let mut errors = MultiError::new();

    for item in items {
        if let Err(e) = validate_item(&item) {
            errors.add(Box::new(e));
        }
    }

    errors.into_result()
}

struct Item;

#[derive(Error, Debug)]
#[error("validation failed")]
struct ItemValidationError;

fn validate_item(_item: &Item) -> Result<(), ItemValidationError> {
    Ok(())
}

```

Context best practices: 1. **Include inputs:** What data caused the error? 2. **Include operation:** What were you trying to do? 3. **Include location:** File, line, function 4. **Include timing:** When did it happen? How long did it take? 5. **Include suggestions:** How can the user fix it? 6. **Chain sources:** Preserve the full error chain

Context anti-patterns: - Redundant information already in source error - Sensitive data (passwords, tokens) in error messages - Too much context (stack of 20+ context strings) - Context that's obvious from source code

Pattern 4: Recoverable vs Unrecoverable Errors

Problem: Using `Result` for everything forces callers to handle programmer errors (bugs) that should never occur, cluttering code with defensive checks. Using `panic!` for recoverable errors (file not found, network timeout) makes software brittle—crashes instead of graceful degradation.

Solution: Use `Result` for expected failures that callers should handle (file not found, parse errors, network failures). Use `panic!` for programmer errors and invariant violations (out-of-bounds indexing, assertion failures, contract violations).

Why It Matters: Using `panic!` for programmer errors catches bugs immediately in development—if an array index is out of bounds, the program crashes with a clear error rather than returning a `Result` that might be ignored. Using `Result` for external failures enables graceful degradation—a web server can return 503 for database timeout instead of crashing.

Use Cases: Libraries use `Result` (let caller decide), applications can `panic!` at startup for missing config, long-running services use `Result` with fallbacks, embedded systems may `panic!` on OOM, test code uses `unwrap()` liberally, FFI boundaries must catch panics with `catch_unwind`.

Examples

```
//=====
// Pattern: Panic for programmer errors
//=====

fn get_user(users: &[User], index: usize) -> &User {
    &users[index] // Panics on out-of-bounds – caller bug
}

//=====
// Pattern: Result for expected failures
//=====

fn find_user(users: &[User], id: u64) -> Option<&User> {
    users.iter().find(|u| u.id == id) // None is expected
}

struct User {
    id: u64,
    name: String,
}

//=====
// Pattern: Expect with informative message
//=====

fn initialize() {
    let config = load_config()
        .expect("Failed to load config: ensure config.toml exists in working directory");
}

fn load_config() -> Result<String, std::io::Error> {
    std::fs::read_to_string("config.toml")
}

//=====
// Pattern: Unwrap in tests
//=====

#[cfg(test)]
mod tests {
```

```

#[test]
fn test_parse() {
    let result = parse_number("42").unwrap(); // OK in tests
    assert_eq!(result, 42);
}

use super::*;

//=====
// Pattern: Debug assertions
//=====

fn compute_checksum(data: &[u8]) -> u32 {
    debug_assert!(!data.is_empty(), "Data must not be empty");
    data.iter().map(|&b| b as u32).sum()
}

//=====
// Pattern: Fail fast at startup
//=====

fn main() -> Result<()> {
    let config = load_config()?;
    let db = connect_database(&config)?;

    // Application starts only if initialization succeeds
    run_server(db)?;
    Ok(())
}

fn connect_database(_config: &str) -> Result<Database> {
    Ok(Database)
}

fn run_server(_db: Database) -> Result<()> {
    Ok(())
}

struct Database;

//=====
// Pattern: Graceful degradation
//=====

fn get_user_with_fallback(id: u64) -> User {
    match fetch_user_from_cache(id) {
        Ok(user) => user,
        Err(_) => {
            eprintln!("Cache miss for user {}, fetching from DB", id);
            fetch_user_from_db(id).unwrap_or_else(|_| User {
                id,
                name: "Unknown".to_string(),
            })
        }
    }
}

```

```

}

fn fetch_user_from_cache(_id: u64) -> Result<User, CacheError> {
    Err(CacheError)
}

fn fetch_user_from_db(_id: u64) -> Result<User, DbError> {
    Err(DbError)
}

#[derive(Error, Debug)]
#[error("cache error")]
struct CacheError;

#[derive(Error, Debug)]
#[error("db error")]
struct DbError;

//=====
// Pattern: Poisoning vs panicking
//=====

use std::sync::{Mutex, PoisonError};

fn update_counter(counter: &Mutex<i32>) -> Result<(), String> {
    match counter.lock() {
        Ok(mut c) => {
            *c += 1;
            Ok(())
        }
        Err(poisoned) => {
            // Mutex poisoned due to panic in another thread
            eprintln!("Mutex poisoned, recovering...");
            let mut c = poisoned.into_inner();
            *c += 1;
            Ok(())
        }
    }
}

//=====
// Pattern: Abort on critical errors
//=====

fn write_checkpoint(data: &[u8]) -> Result<()> {
    std::fs::write("checkpoint.dat", data).map_err(|e| {
        eprintln!("CRITICAL: Failed to write checkpoint: {}", e);
        eprintln!("Data integrity cannot be guaranteed. Aborting.");
        std::process::abort();
    })
}

//=====
// Pattern: Catch unwind for FFI boundaries
//=====
```

```

use std::panic::{catch_unwind, AssertUnwindSafe};

#[no_mangle]
pub extern "C" fn safe_compute(input: i32) -> i32 {
    match catch_unwind(AssertUnwindSafe(|| risky_computation(input))) {
        Ok(result) => result,
        Err(_) => {
            eprintln!("Computation panicked, returning default");
            0
        }
    }
}

fn risky_computation(input: i32) -> i32 {
    if input < 0 {
        panic!("Negative input!");
    }
    input * 2
}

```

Decision tree: Result vs Panic

Use **Result** when: - Failure is expected (file not found, network timeout) - Caller should handle the error - Error can be recovered - Library code (let caller decide)

Use **panic!** when: - Programmer error (contract violation) - Invariant violated (impossible state) - Continuing would corrupt data - Prototype/example code

Use **Option** when: - Absence is a valid state (empty collection) - No error context needed - Simpler than Result<T, ()>

Pattern 5: Error Handling in Async Contexts

Problem: Async operations introduce failure modes absent in synchronous code: timeouts (operation took too long), cancellation (task dropped before completion), concurrent failures (10 out of 100 requests failed), and cascading failures (one service down brings down dependent services). Naive async error handling leads to unbounded waits, resource leaks from cancelled operations, and unclear error reporting when multiple concurrent operations fail.

Solution: Wrap all I/O operations in timeouts using `tokio::time::timeout`. Use `try_join!` or `try_join_all` to propagate first error from concurrent operations.

Why It Matters: Without timeouts, a single slow dependency can hang your entire service—one database query taking 30 seconds blocks all concurrent requests. Without proper cancellation handling, dropped tasks can leave files partially written or transactions uncommitted.

Use Cases: Web servers handling concurrent requests, microservices with service-to-service calls, batch processing with concurrent workers, real-time systems with latency requirements, streaming data pipelines, distributed systems requiring fault tolerance.

Examples

```
use tokio;
use std::time::Duration;

//=====
// Pattern: Async error propagation
//=====

async fn fetch_user_data(id: u64) -> Result<User> {
    let response = make_http_request(id).await?;
    let user = parse_response(response).await?;
    Ok(user)
}

async fn make_http_request(_id: u64) -> Result<String> {
    Ok("response".to_string())
}

async fn parse_response(_response: String) -> Result<User> {
    Ok(User { id: 1, name: "Alice".to_string() })
}

//=====
// Pattern: Timeout with context
//=====

async fn fetch_with_timeout(id: u64) -> Result<User> {
    tokio::time::timeout(Duration::from_secs(5), fetch_user_data(id))
        .await
        .map_err(|_| anyhow::anyhow!("Timeout fetching user {}", id))?
}

//=====
// Pattern: Concurrent operations with try_join
//=====

async fn fetch_multiple_users(ids: Vec<u64>) -> Result<Vec<User>> {
    let futures = ids.into_iter().map(fetch_user_data);

    futures::future::try_join_all(futures)
        .await
        .context("Failed to fetch all users")
}

//=====
// Pattern: Race multiple operations
//=====

async fn fetch_with_fallback(id: u64) -> Result<User> {
    tokio::select! {
        result = fetch_from_primary(id) => result,
        result = fetch_from_secondary(id) => result,
    }
}
```

```

async fn fetch_from_primary(_id: u64) -> Result<User> {
    tokio::time::sleep(Duration::from_secs(1)).await;
    Ok(User { id: 1, name: "Primary".to_string() })
}

async fn fetch_from_secondary(_id: u64) -> Result<User> {
    tokio::time::sleep(Duration::from_secs(2)).await;
    Ok(User { id: 1, name: "Secondary".to_string() })
}

//=====
// Pattern: Error recovery in stream processing
//=====

use futures::StreamExt;

async fn process_stream(mut stream: impl futures::Stream<Item = Result<i32> + Unpin>) {
    while let Some(result) = stream.next().await {
        match result {
            Ok(value) => println!("Processed: {}", value),
            Err(e) => {
                eprintln!("Error in stream: {}", e);
                // Continue processing despite errors
            }
        }
    }
}

//=====
// Pattern: Aggregating errors from concurrent tasks
//=====

async fn parallel_validation(items: Vec<Item>) -> Result<(), MultiError> {
    let handles: Vec<_> = items
        .into_iter()
        .map(|item| tokio::spawn(async move { validate_item(&item) }))
        .collect();

    let mut errors = MultiError::new();

    for handle in handles {
        if let Ok(Err(e)) = handle.await {
            errors.add(Box::new(e));
        }
    }

    errors.into_result()
}

//=====
// Pattern: Graceful shutdown on error
//=====

async fn run_worker(mut shutdown: tokio::sync::broadcast::Receiver<()>) -> Result<()> {
    loop {
        tokio::select! {

```

```

        _ = shutdown.recv() => {
            println!("Shutting down gracefully");
            return Ok(());
        }
    result = do_work() => {
        if let Err(e) = result {
            eprintln!("Work failed: {}", e);
            // Decide whether to continue or stop
            if is_fatal(&e) {
                return Err(e);
            }
        }
    }
}

async fn do_work() -> Result<()> {
    tokio::time::sleep(Duration::from_millis(100)).await;
    Ok(())
}

fn is_fatal(_error: &anyhow::Error) -> bool {
    false
}

//=====
// Pattern: Retry with exponential backoff
//=====

async fn retry_with_backoff<F, T, Fut>(f: F, max_attempts: usize) -> Result<T>
where
    F: Fn() -> Fut,
    Fut: std::future::Future<Output = Result<T>>,
{
    let mut attempt = 0;
    let mut delay = Duration::from_millis(100);

    loop {
        attempt += 1;
        match f().await {
            Ok(value) => return Ok(value),
            Err(e) if attempt >= max_attempts => {
                return Err(e.context(format!("Failed after {} attempts", attempt)));
            }
            Err(e) => {
                eprintln!("Attempt {} failed: {}, retrying in {:?}", attempt, e, delay);
                tokio::time::sleep(delay).await;
                delay *= 2; // Exponential backoff
            }
        }
    }
}

```

```

//=====
// Pattern: Cancellation-safe operations
//=====

async fn cancellation_safe_write(data: &[u8]) -> Result<()> {
    let temp_path = "temp_file.tmp";
    let final_path = "final_file.dat";

    // Write to temp file (cancellation here is OK)
    tokio::fs::write(temp_path, data).await
        .context("Failed to write temp file")?;

    // Atomic rename (fast, unlikely to be cancelled)
    tokio::fs::rename(temp_path, final_path).await
        .context("Failed to rename file")?;

    Ok(())
}

//=====
// Pattern: Circuit breaker pattern
//=====

use std::sync::Arc;
use std::sync::atomic::{AtomicUsize, Ordering};

struct CircuitBreaker {
    failure_count: Arc<AtomicUsize>,
    threshold: usize,
}

impl CircuitBreaker {
    fn new(threshold: usize) -> Self {
        CircuitBreaker {
            failure_count: Arc::new(AtomicUsize::new(0)),
            threshold,
        }
    }

    async fn call<F, T, Fut>(&self, f: F) -> Result<T>
    where
        F: FnOnce() -> Fut,
        Fut: std::future::Future<Output = Result<T>>,
    {
        let failures = self.failure_count.load(Ordering::Relaxed);

        if failures >= self.threshold {
            anyhow::bail!("Circuit breaker open: too many failures");
        }

        match f().await {
            Ok(value) => {
                self.failure_count.store(0, Ordering::Relaxed);
                Ok(value)
            }
        }
    }
}

```

```

        Err(e) => {
            self.failure_count.fetch_add(1, Ordering::Relaxed);
            Err(e)
        }
    }
}

```

Async error handling principles: 1. **Timeouts everywhere:** Network calls should have timeouts

2. **Graceful degradation:** Don't fail entire operation for one sub-task 3. **Cancellation safety:** Ensure operations can be cancelled safely

4. **Error aggregation:** Collect errors from concurrent operations 5.

Circuit breakers: Fail fast when downstream is unavailable 6. **Retry with backoff:** Transient failures should retry with exponential backoff

Pattern 6: Error Handling Anti-Patterns

```

// ✗ Swallowing errors
fn bad_error_handling(path: &str) {
    let _ = std::fs::read_to_string(path); // Error ignored!
}

// ✓ Log or propagate
fn good_error_handling(path: &str) -> Result<String> {
    std::fs::read_to_string(path)
        .context("Failed to read file")
}

// ✗ Using unwrap in library code
pub fn parse_config(path: &str) -> Config {
    let content = std::fs::read_to_string(path).unwrap(); // Will panic!
    serde_json::from_str(&content).unwrap()
}

// ✓ Return Result, let caller decide
pub fn parse_config_safe(path: &str) -> Result<Config> {
    let content = std::fs::read_to_string(path)?;
    Ok(serde_json::from_str(&content)?)
}

// ✗ Generic error messages
fn load_data() -> Result<Vec<u8>> {
    std::fs::read("data.bin")
        .map_err(|e| anyhow::anyhow!("Error: {}", e)) // Unhelpful
}

// ✓ Specific, actionable messages
fn load_data_better() -> Result<Vec<u8>> {
    std::fs::read("data.bin")
        .context("Failed to read data.bin: ensure file exists and is readable")
}

```

```

// ✗ Catching and re-panicking
fn bad_panic_handling() {
    if let Err(e) = risky_operation() {
        panic!("Operation failed: {}", e); // Why not just return Result?
    }
}

fn risky_operation() -> Result<()> {
    Ok(())
}

// ✓ Propagate errors normally
fn good_panic_handling() -> Result<()> {
    risky_operation()?;
    Ok(())
}

// ✗ Over-broad error types
fn overly_generic() -> Result<i32, Box<dyn std::error::Error>> {
    Ok(42) // Caller doesn't know what errors to expect
}

// ✓ Specific error types
#[derive(Error, Debug)]
enum ParseError2 {
    #[error("Io error")]
    Io(#[from] std::io::Error),
    #[error("Parse error")]
    Parse(#[from] std::num::ParseIntError),
}

fn specific_errors() -> Result<i32, ParseError2> {
    Ok(42) // Clear what can fail
}

```

Error Handling Comparison

| Approach | Use Case | Pros | Cons |
|---------------------------------|-------------------|-------------------------|------------------------------|
| <code>Result<T, E></code> | Libraries | Type-safe, composable | Verbose, requires error type |
| <code>anyhow::Result</code> | Applications | Ergonomic, rich context | Dynamic type, less precise |
| <code>Option<T></code> | Simple absence | Lightweight | No error info |
| <code>panic!</code> | Programmer errors | Impossible to ignore | Cannot recover |
| <code>thiserror</code> | Custom errors | Reduces boilerplate | Compile-time cost |

Key Takeaways

1. Libraries: specific errors, applications: opaque errors
2. Add context at error sites, not at error definitions
3. Use ? for clean propagation
4. panic! for programmer errors, Result for runtime errors
5. Preserve error chains with source()
6. Include actionable information in error messages
7. Async: timeouts, retries, graceful degradation
8. Profile error handling overhead (usually negligible)
9. Use thiserror for custom errors, anyhow for applications
10. Document error conditions in function signatures

Vec & Slice Manipulation

Vectors and slices are the workhorses of Rust data processing. `Vec<T>` provides dynamic, heap-allocated arrays with amortized O(1) append operations, while slices (`&[T]`, `&mut [T]`) provide views into contiguous sequences without ownership. Understanding how to efficiently manipulate these types is essential for writing high-performance Rust code.

The key insight is that careful capacity management, zero-copy operations, and algorithmic choices can dramatically impact performance—often by orders of magnitude.

Pattern 1: Capacity Management and Amortization

Problem: Growing vectors incrementally triggers repeated reallocations—each doubling copies all existing elements. Building a 100K-element vector without pre-allocation causes ~17 reallocations and copies 200K elements total.

Solution: Use `Vec::with_capacity(n)` when size is known upfront. Call `reserve(n)` before bulk operations to pre-allocate space.

Why It Matters: Pre-allocation can improve performance by 10-100x for vector construction. A data pipeline building 1M-element results: naive approach does ~20 reallocations copying ~2M elements.

Use Cases: Batch processing (pre-allocate for batch size), collecting query results (reserve based on estimated count), temporary buffers in loops (reuse with clear), building large datasets (with_capacity), long-lived lookup tables (shrink_to_fit after construction).

Example: Pre-allocate When Size is Known

When you know how many elements you'll add to a vector, pre-allocating with `with_capacity` eliminates all reallocations during construction. This is the single most impactful optimization for vector building.

```
fn process_batch(items: &[Item]) -> Vec<ProcessedItem> {
    let mut results = Vec::with_capacity(items.len());
    for item in items {
```

```
        results.push(process(item));
    }
    results
}
```

Example: Reserve Before Iterative Construction

When building vectors from multiple sources, estimate the total size upfront and reserve space once. This avoids multiple reallocations as the vector grows.

```
fn build_result_set(queries: &[Query]) -> Vec<Result> {
    let mut results = Vec::new();

    // Estimate total size to avoid multiple reallocations
    let estimated_total: usize = queries.iter()
        .map(|q| q.estimated_results())
        .sum();

    results.reserve(estimated_total);

    for query in queries {
        for result in execute_query(query) {
            results.push(result);
        }
    }

    results
}
```

Example: Reuse Vectors to Avoid Allocation

In loops where you build temporary vectors repeatedly, reuse a single buffer by calling `clear()` between iterations. This retains the allocated capacity and eliminates allocation overhead entirely.

```
fn batch_processor(batches: &[Batch]) -> Vec<Vec<Output>> {
    let mut buffer = Vec::with_capacity(1000);
    let mut all_results = Vec::with_capacity(batches.len());

    for batch in batches {
        buffer.clear(); // Retains capacity

        for item in &batch.items {
            buffer.push(process_item(item));
        }

        // Clone only the used portion
        all_results.push(buffer.clone());
    }
}
```

```
    all_results
}
```

Example: Track Amortized Growth

Monitoring allocation patterns helps identify performance problems. This wrapper tracks how many reallocations occur during vector growth, revealing whether pre-allocation is needed.

```
struct GrowableBuffer<T> {
    data: Vec<T>,
    allocations: usize,
}

impl<T> GrowableBuffer<T> {
    fn new() -> Self {
        GrowableBuffer {
            data: Vec::new(),
            allocations: 0,
        }
    }

    fn push(&mut self, value: T) {
        let old_cap = self.data.capacity();
        self.data.push(value);
        let new_cap = self.data.capacity();

        if new_cap > old_cap {
            self.allocations += 1;
        }
    }

    fn stats(&self) -> (usize, usize, usize) {
        (self.data.len(), self.data.capacity(), self.allocations)
    }
}
```

Example: Shrink Long-Lived Data Structures

When a vector is over-allocated and will remain in memory for a long time, use `shrink_to_fit` to reclaim the excess capacity. This is particularly important for lookup tables and cached data.

```
fn build_lookup_table(entries: &[Entry]) -> Vec<IndexEntry> {
    let mut table = Vec::with_capacity(entries.len() * 2); // Over-estimate

    for entry in entries {
        if entry.should_index() {
            table.push(IndexEntry::from(entry));
        }
    }
    // Reclaim unused space for long-lived data
    table.shrink_to_fit();
```

```
    table
}
```

Example: Use Iterator Size Hints

When collecting from iterators, leverage the `size_hint` to pre-allocate optimal capacity. This is especially useful when the iterator provides accurate bounds.

```
fn collect_filtered(items: impl Iterator<Item = i32>) -> Vec<i32> {
    // When collecting from iterators, size_hint can optimize allocation
    let (lower, upper) = items.size_hint();

    let mut result = if let Some(upper) = upper {
        Vec::with_capacity(upper)
    } else {
        Vec::with_capacity(lower)
    };

    result.extend(items);
    result
}
```

Example: Batch Insertion with Extend

When merging multiple vectors into one, calculate the total size upfront and reserve all needed space at once. This prevents reallocations during the merge operation.

```
fn merge_results(target: &mut Vec<String>, sources: &[Vec<String>]) {
    let total: usize = sources.iter().map(|v| v.len()).sum();
    target.reserve(total);

    for source in sources {
        target.extend_from_slice(source);
    }
}
```

Example: Building Large Datasets Efficiently

For datasets where you know the exact size, pre-allocation ensures zero reallocations during construction. The assertion at the end verifies that no reallocation occurred.

```
fn generate_dataset(n: usize) -> Vec<DataPoint> {
    let mut data = Vec::with_capacity(n);

    for i in 0..n {
        data.push(DataPoint {
            id: i,
            value: compute_value(i),
            metadata: generate_metadata(i),
        });
    }

    assert_eq!(data.capacity(), n);
    assert!(data.is_contiguous());
    assert!(data.as_ptr() == data.data());
}
```

```

    });
}

assert_eq!(data.len(), data.capacity()); // No reallocation occurred
data
}

```

Capacity management principles: 1. **Pre-allocate when size is known:** Use `with_capacity` to avoid reallocations 2. **Reserve before bulk operations:** Prevent multiple reallocations during growth 3. **Reuse vectors with `clear()`:** Retains capacity for the next use 4. **Shrink long-lived vectors:** Use `shrink_to_fit` to reclaim memory 5. **Track growth for performance analysis:** Monitor reallocation frequency in hot paths

Pattern 2: Slice Algorithms

Problem: Linear search through large sorted arrays is $O(N)$ when $O(\log N)$ binary search is possible. Sorting entire datasets to find median or top-K wastes $O(N \log N)$ time.

Solution: Use `binary_search` and `partition_point` for $O(\log N)$ searches on sorted data. Apply `select_nth_unstable` for $O(N)$ median/top-K finding without full sort.

Why It Matters: Algorithm choice dramatically affects performance. Finding an element: linear search $O(N)$ vs binary search $O(\log N)$ is 1000x difference for 1M elements.

Use Cases: Database query optimization (binary search on sorted indices), statistics computation (median, percentiles with `select_nth`), data deduplication (sort + dedup), priority queues (partition by priority), cyclic buffers (rotate operations), filtering with memory constraints (retain vs filter+collect).

Example: Binary Search on Sorted Data

Binary search provides $O(\log N)$ lookup on sorted slices, dramatically faster than linear search for large datasets. The slice must be sorted by the search key for correct results.

```

fn find_user_by_id(users: &[User], id: u64) -> Option<&User> {
    // users must be sorted by id
    users.binary_search_by_key(&id, |u| u.id)
        .ok()
        .map(|idx| &users[idx])
}

```

Example: Partition Point for Range Queries

`partition_point` finds the index where a predicate transitions from true to false, enabling efficient range queries on sorted data. This is particularly useful for database-style queries.

```

fn find_range(sorted: &[i32], min: i32, max: i32) -> &[i32] {
    let start = sorted.partition_point(|&x| x < min);
    let end = sorted.partition_point(|&x| x <= max);

```

```
    &sorted[start..end]
}
```

Example: Partition by Predicate

Partitioning separates elements based on a condition without allocating a new vector. This returns mutable references to both the matching and non-matching segments.

```
fn separate_valid_invalid(items: &mut [Item]) -> (&mut [Item], &mut [Item]) {
    let pivot = items.iter().partition_point(|item| item.is_valid());
    items.split_at_mut(pivot)
}
```

Example: Custom Sorting with Comparators

Complex sorting criteria can be expressed with `sort_by`, chaining multiple comparisons. This example sorts by priority (descending) with timestamp as a tiebreaker (ascending).

```
fn sort_by_priority(tasks: &mut [Task]) {
    tasks.sort_by(|a, b| {
        // Sort by priority descending, then by timestamp ascending
        b.priority.cmp(&a.priority)
            .then_with(|| a.timestamp.cmp(&b.timestamp))
    });
}
```

Example: Unstable Sort for Performance

When the relative order of equal elements doesn't matter, `sort_unstable` runs significantly faster than stable sort. This is ideal for primitive types and performance-critical code.

```
fn sort_large_dataset(data: &mut [f64]) {
    // sort_unstable is faster than sort for primitive types
    data.sort_unstable_by(|a, b| a.partial_cmp(b).unwrap());
}
```

Example: Finding Median and Top-K Elements

`select_nth_unstable` performs partial sorting in O(N) time, making it perfect for finding medians, percentiles, and top-K elements without sorting the entire array.

```
fn find_median(values: &mut [f64]) -> f64 {
    let mid = values.len() / 2;
    let (_, median, _) = values.select_nth_unstable(mid);
    *median
}

fn top_k_elements(values: &mut [i32], k: usize) -> &[i32] {
```

```
    let (_, _, right) = values.select_nth_unstable(values.len() - k);
    right
}
```

Example: Efficient Cyclic Rotation

`rotate_left` and `rotate_right` perform cyclic shifts efficiently without temporary buffers. This is essential for ring buffers and circular data structures.

```
fn rotate_buffer(buffer: &mut [u8], offset: usize) {
    buffer.rotate_left(offset % buffer.len());
}
```

Example: Deduplication on Sorted Data

For removing duplicates, sort first then call `dedup`. This is $O(N \log N)$ for the sort plus $O(N)$ for dedup, much faster than checking each element against all others.

```
fn unique_sorted(items: &mut Vec<i32>) {
    items.sort_unstable();
    items.dedup();
}
```

Example: In-Place Filtering with Retain

`retain` removes elements that don't match a predicate without allocating a new vector. This is more efficient than `filter().collect()` when you want to modify in place.

```
fn remove_invalid(items: &mut Vec<Item>) {
    items.retain(|item| item.is_valid());
}
```

Example: Reverse Operations

Reversing slices in-place is $O(N/2)$ swaps. Combined with chunking, you can reverse segments of data efficiently.

```
fn reverse_segments(data: &mut [u8], segment_size: usize) {
    for chunk in data.chunks_mut(segment_size) {
        chunk.reverse();
    }
}
```

Example: Filling Slices

`fill` sets all elements to a value, while `fill_with` uses a closure to generate values. This is useful for initialization and resetting buffers.

```

fn initialize_buffer(buffer: &mut [u8], pattern: u8) {
    buffer.fill(pattern);
}

fn initialize_with_indices(buffer: &mut [u8]) {
    let mut counter = 0;
    buffer.fill_with(|| {
        let val = counter;
        counter += 1;
        val
    });
}

```

Example: Swapping Slice Ranges

`swap_with_slice` exchanges the contents of two mutable slices in place without allocation. This is useful for data rearrangement and buffer management.

```

fn swap_halves(data: &mut [u8]) {
    let mid = data.len() / 2;
    let (left, right) = data.split_at_mut(mid);
    let min_len = left.len().min(right.len());
    left[..min_len].swap_with_slice(&mut right[..min_len]);
}

```

Example: Pattern Matching with Starts/Ends

Checking for prefixes and suffixes is a common pattern in protocol parsing and file format detection.

```

fn has_magic_header(data: &[u8]) -> bool {
    const MAGIC: &[u8] = b"PNG\x89";
    data.starts_with(MAGIC)
}

```

Example: Finding Subsequences

Searching for a pattern within a slice can be done efficiently using windows combined with position finding.

```

fn find_pattern(haystack: &[u8], needle: &[u8]) -> Option<u8> {
    haystack.windows(needle.len())
        .position(|window| window == needle)
}

```

Slice algorithm guidelines: 1. **Use binary_search for sorted data:** O(log n) vs O(n) for linear search 2. **Prefer sort_unstable for primitives:** Faster than stable sort when order doesn't matter 3. **Use select_nth_unstable for top-k:** O(n) vs O(n log n) for full sort 4. **Partition instead of filter+collect:** Avoids allocation 5. **Use rotate for cyclic shifts:** More efficient than manual copying

Pattern 3: Chunking and Windowing

Problem: Processing large datasets element-by-element is slow due to function call overhead and poor cache locality. Batch operations require manual index calculation and are error-prone.

Solution: Use `.chunks(n)` for non-overlapping fixed-size batches. Use `.windows(n)` for overlapping subsequences (moving averages, pattern detection).

Why It Matters: Chunking improves cache locality—processing 1000-element chunks instead of individual elements can be 10-50x faster. Window operations enable signal processing algorithms (FFT, convolution) without collecting intermediate vectors—zero allocation for moving statistics.

Use Cases: Batch processing (database inserts, API requests), signal processing (moving averages, FFT windows), image processing (tile-based operations), parallel computation (divide work across threads), network packet assembly (fixed-size frames), time-series analysis (rolling statistics).

Example: Fixed-Size Chunking

`chunks` divides a slice into non-overlapping segments of a specified size. The last chunk may be smaller if the slice length isn't evenly divisible.

```
fn process_in_batches(data: &[u8], batch_size: usize) -> Vec<ProcessedBatch> {
    data.chunks(batch_size)
        .map(|chunk| process_batch(chunk))
        .collect()
}
```

Example: Mutable Chunks for In-Place Transformation

`chunks_mut` provides mutable access to each chunk, enabling in-place transformations without copying data. This is ideal for batch normalization and similar operations.

```
fn normalize_batches(data: &mut [f64], batch_size: usize) {
    for chunk in data.chunks_mut(batch_size) {
        let sum: f64 = chunk.iter().sum();
        let mean = sum / chunk.len() as f64;

        for value in chunk {
            *value -= mean;
        }
    }
}
```

Example: Exact Chunks with Remainder Handling

`as_chunks` splits a slice into fixed-size arrays with compile-time size checking, returning both the aligned chunks and the remainder. This is essential for SIMD-optimized code.

```

fn process_with_remainder(data: &[u8], chunk_size: usize) {
    let (chunks, remainder) = data.as_chunks::<8>();

    for chunk in chunks {
        // Process full 8-byte chunks
        process_aligned_chunk(chunk);
    }

    if !remainder.is_empty() {
        // Handle remaining bytes
        process_partial_chunk(remainder);
    }
}

```

Example: Sliding Windows for Moving Averages

`windows` creates overlapping views of the slice, perfect for computing rolling statistics without allocating intermediate buffers.

```

fn moving_average(values: &[f64], window_size: usize) -> Vec<f64> {
    values.windows(window_size)
        .map(|window| {
            let sum: f64 = window.iter().sum();
            sum / window.len() as f64
        })
        .collect()
}

```

Example: Pairwise Operations

Windows of size 2 enable efficient computation of differences, ratios, or other pairwise operations between adjacent elements.

```

fn compute_deltas(values: &[i32]) -> Vec<i32> {
    values.windows(2)
        .map(|pair| pair[1] - pair[0])
        .collect()
}

```

Example: Pattern Matching in Overlapping Windows

Windows combined with filtering enable detection of consecutive sequences or patterns spanning multiple elements.

```

fn find_consecutive_sequences(data: &[i32], target: i32) -> Vec<usize> {
    data.windows(3)
        .enumerate()
        .filter_map(|(i, window)| {
            if window.iter().all(|&x| x == target) {

```

```

        Some(i)
    } else {
        None
    }
})
.collect()
}

```

Example: Parallel Processing with Chunks

Chunking naturally partitions work for parallel processing. Each thread processes one chunk independently.

```

fn parallel_sum(data: &[i64]) -> i64 {
    use std::thread;

    let chunk_size = data.len() / num_cpus::get();
    let handles: Vec<_> = data.chunks(chunk_size)
        .map(|chunk| {
            let chunk = chunk.to_vec();
            thread::spawn(move || chunk.iter().sum::<i64>())
        })
        .collect();

    handles.into_iter()
        .map(|h| h.join().unwrap())
        .sum()
}

```

Example: Reverse Chunking

`rchunks` processes chunks from the end of the slice backward, useful for parsing formats where metadata appears at the end.

```

fn process_backwards(data: &[u8], chunk_size: usize) {
    for chunk in data.rchunks(chunk_size) {
        process_chunk(chunk);
    }
}

```

Example: Exact Chunks vs Regular Chunks

`chunks_exact` guarantees all chunks (except the explicit remainder) have the exact size, simplifying algorithms that require uniform blocks.

```

fn encode_blocks(data: &[u8], block_size: usize) -> Vec<EncodedBlock> {
    let mut encoded = Vec::new();

    // Process full blocks

```

```

for block in data.chunks_exact(block_size) {
    encoded.push(encode_full_block(block));
}

// Handle remainder
let remainder = &data[data.len() - (data.len() % block_size)..];
if !remainder.is_empty() {
    encoded.push(encode_partial_block(remainder));
}

encoded
}

```

Example: Strided Access with Step By

Combining iteration with `step_by` enables sampling every Nth element, useful for downsampling data.

```

fn sample_every_nth(data: &[f64], n: usize) -> Vec<f64> {
    data.iter()
        .step_by(n)
        .copied()
        .collect()
}

```

Example: Splitting into Equal Parts

Dividing data into N roughly equal parts is common for load balancing across workers.

```

fn split_into_n_parts(data: &[u8], n: usize) -> Vec<&[u8]> {
    let chunk_size = (data.len() + n - 1) / n; // Ceiling division
    data.chunks(chunk_size).collect()
}

```

Example: Signal Processing with Overlapping Windows

Advanced windowing combines `step_by` for hop size with manual slicing for overlapping FFT windows in spectrograms.

```

fn compute_spectrogram(signal: &[f32], window_size: usize, hop_size: usize) -> Vec<Vec<f32>> {
    let mut result = Vec::new();

    for i in (0..signal.len()).step_by(hop_size) {
        if i + window_size <= signal.len() {
            let window = &signal[i..i + window_size];
            let spectrum = fft(window);
            result.push(spectrum);
        }
    }
}

```

```
    result  
}
```

Chunking and windowing principles: 1. **Use chunks for batch processing:** Non-overlapping segments 2. **Use windows for sliding operations:** Moving averages, pattern detection 3. **Handle remainders explicitly:** chunks_exact makes remainder handling clear 4. **Prefer chunks over manual indexing:** More idiomatic and less error-prone 5. **Consider rchunks for reverse processing:** Cleaner than reversing then chunking

Pattern 4: Zero-Copy Slicing

Problem: Parsing structured data by extracting fields into owned `String/Vec` causes allocation for every field—parsing 1M CSV records with 10 fields allocates 10M strings. Returning data from functions forces cloning entire vectors.

Solution: Return borrowed slices (`&[T]`, `&str`) that reference the original data. Use `split()`, `split_at()`, and range indexing to create views.

Why It Matters: Zero-copy parsing can be 10-100x faster than allocating approach. Parsing a 100MB CSV file: allocating approach needs gigabytes of temporary memory and causes GC pressure.

Use Cases: CSV/JSON parsing (return field slices), network protocol parsers (split packets into views), text processing (split without allocation), binary format parsing (frame headers/payloads), configuration file parsing, streaming data processors.

Example: Return Slices Instead of Cloning

By returning borrowed slices instead of owned strings, CSV parsing avoids allocating memory for every field, dramatically improving performance.

```
fn find_field<'a>(record: &'a [u8], field_index: usize) -> &'a [u8] {  
    let mut start = 0;  
    let mut current_field = 0;  
  
    for (i, &byte) in record.iter().enumerate() {  
        if byte == b',' {  
            if current_field == field_index {  
                return &record[start..i];  
            }  
            current_field += 1;  
            start = i + 1;  
        }  
    }  
  
    if current_field == field_index {  
        &record[start..]  
    } else {  
        &[]  
    }  
}
```

```
    }  
}
```

Example: Split Without Allocation

The `split` iterator creates string slices on the fly without allocating a vector until `collect` is called. Each slice references the original string.

```
fn parse_csv_line(line: &str) -> Vec<&str> {  
    line.split(',')  
        .map(|s| s.trim())  
        .collect()  
}
```

Example: Multiple Slices from One Allocation

A struct can hold multiple slices all pointing into a single backing buffer, enabling zero-copy frame parsing for network protocols.

```
struct Frame<'a> {  
    header: &'a [u8],  
    payload: &'a [u8],  
    checksum: &'a [u8],  
}  
  
impl<'a> Frame<'a> {  
    fn parse(data: &'a [u8]) -> Result<Self, ParseError> {  
        if data.len() < 10 {  
            return Err(ParseError::TooShort);  
        }  
  
        Ok(Frame {  
            header: &data[0..4],  
            payload: &data[4..data.len() - 4],  
            checksum: &data[data.len() - 4..],  
        })  
    }  
}
```

Example: Split At for Header/Body Separation

`split_at` divides a slice at a specific index, creating two non-overlapping views perfect for fixed-size header parsing.

```
fn process_header_and_body(data: &[u8]) -> Result<(Header, Vec<Item>), Error> {  
    let (header_bytes, body) = data.split_at(HEADER_SIZE);  
    let header = parse_header(header_bytes)?;  
    let items = parse_body(body)?;
```

```
    Ok((header, items))
}
```

Example: Copy-on-Write with Cow

Cow (Clone on Write) enables APIs that only allocate when modification is needed, borrowing otherwise. Perfect for conditional string encoding fixes.

```
use std::borrow::Cow;

fn decode_field(field: &[u8]) -> Cow<str> {
    match std::str::from_utf8(field) {
        Ok(s) => Cow::Borrowed(s),
        Err(_) => {
            // Only allocate when we need to fix encoding
            Cow::Owned(String::from_utf8_lossy(field).into_owned())
        }
    }
}
```

Example: Split First and Last for Protocol Parsing

`split_first` and `split_last` extract single elements while returning a slice of the remainder, ideal for version bytes and checksums.

```
fn parse_packet(data: &[u8]) -> Result<Packet, ParseError> {
    let (&version, rest) = data.split_first()
        .ok_or(ParseError::Empty)?;

    let (payload, &checksum) = rest.split_last()
        .ok_or(ParseError::NoChecksum)?;

    Ok(Packet { version, payload, checksum })
}
```

Example: Iterating Without Collecting

When the final result doesn't need individual slices, process them in the iterator pipeline without allocating a vector.

```
fn sum_valid_numbers(data: &str) -> i32 {
    data.split(',')
        .filter_map(|s| s.trim().parse::<i32>().ok())
        .sum()
}
```

Example: Slicing During Iteration

Manual range-based slicing combined with iteration enables custom chunk processing without the constraints of fixed-size chunks.

```
fn process_blocks(data: &[u8], block_size: usize) -> Vec<BlockResult> {
    (0..data.len())
        .step_by(block_size)
        .map(|i| {
            let end = (i + block_size).min(data.len());
            process_block(&data[i..end])
        })
        .collect()
}
```

Example: Grouping by Delimiter

Manual delimiter-based splitting provides more control than the built-in `split` method, useful for binary data or custom delimiters.

```
fn group_by_delimiter(data: &[u8], delimiter: u8) -> Vec<&[u8]> {
    let mut groups = Vec::new();
    let mut start = 0;

    for (i, &byte) in data.iter().enumerate() {
        if byte == delimiter {
            groups.push(&data[start..i]);
            start = i + 1;
        }
    }

    if start < data.len() {
        groups.push(&data[start..]);
    }

    groups
}
```

Example: In-Place Mutable Operations

Mutable slices enable in-place transformations like byte swapping without any allocation.

```
fn swap_bytes_in_place(data: &mut [u8]) {
    for pair in data.chunks_exact_mut(2) {
        pair.swap(0, 1);
    }
}
```

Example: Complete Zero-Copy HTTP Parser

This comprehensive example shows how an entire HTTP request parser can work with zero allocations, storing only slices into the original buffer.

```
struct HttpRequest<'a> {
    method: &'a str,
    path: &'a str,
    headers: Vec<(&'a str, &'a str)>,
    body: &'a [u8],
}

impl<'a> HttpRequest<'a> {
    fn parse(data: &'a [u8]) -> Result<Self, ParseError> {
        let data_str = std::str::from_utf8(data)
            .map_err(|_| ParseError::InvalidUtf8)?;

        let (head, body) = data_str.split_once("\r\n\r\n")
            .ok_or(ParseError::NoBodySeparator)?;

        let mut lines = head.lines();
        let request_line = lines.next().ok_or(ParseError::Empty)?;

        let mut parts = request_line.split_whitespace();
        let method = parts.next().ok_or(ParseError::NoMethod)?;
        let path = parts.next().ok_or(ParseError::NoPath)?;

        let headers: Vec<_> = lines
            .filter_map(|line| line.split_once(": "))
            .collect();

        Ok(HttpRequest {
            method,
            path,
            headers,
            body: body.as_bytes(),
        })
    }
}
```

Zero-copy principles: 1. **Return slices when possible:** Avoid cloning unless necessary 2. **Use split methods:** split, split_at, split_once for parsing 3. **Leverage Cow for conditional cloning:** Only allocate when modification is needed 4. **Parse in-place:** Use iterators and slices instead of collecting 5. **Design APIs for borrowing:** Accept &[T] instead of Vec when ownership isn't needed

Pattern 5: SIMD Operations

Problem: Processing large arrays element-by-element leaves CPU vector units idle—modern CPUs can process 4-16 elements per instruction but scalar code uses only one. Image processing, audio encoding, numerical computing, and data compression are bottlenecked by sequential processing.

Solution: Use portable SIMD through `std::simd` (nightly) or crates like `packed_simd2`. Process data in SIMD-width chunks (4/8/16 elements).

Why It Matters: SIMD provides 4-16x speedups for data-parallel operations. Processing 1M floats: scalar takes 1M operations, SIMD with 8-wide vectors takes 125K operations.

Use Cases: Image/video processing (filters, transformations), audio processing (effects, encoding), numerical computing (matrix operations, scientific simulations), compression algorithms, checksums and hashing, database query execution, machine learning inference.

Example: Manual SIMD-Friendly Chunking

Processing data in fixed-size chunks enables the compiler to auto-vectorize, and provides a clear structure for manual SIMD optimization.

```
fn sum_bytes(data: &[u8]) -> u64 {
    let (chunks, remainder) = data.as_chunks::<8>();

    let mut sum = 0u64;

    // Process 8 bytes at a time
    for chunk in chunks {
        for &byte in chunk {
            sum += byte as u64;
        }
    }

    // Handle remainder
    for &byte in remainder {
        sum += byte as u64;
    }

    sum
}
```

Example: Aligned Data Structures

Proper memory alignment is crucial for SIMD performance. Using `#[repr(align(N))]` ensures data is aligned for vector instructions.

```
#[repr(align(32))]
struct AlignedBuffer([f32; 8]);

fn process_aligned(data: &[AlignedBuffer]) -> Vec<f32> {
    data.iter()
        .flat_map(|buf| buf.0.iter())
        .map(|&x| x * 2.0)
        .collect()
}
```

Example: Portable SIMD with std::simd

Rust's portable SIMD (nightly) provides safe, cross-platform vector operations that compile to optimal CPU instructions.

```
#[cfg(feature = "portable_simd")]
fn add_vectors_simd(a: &[f32], b: &[f32], result: &mut [f32]) {
    use std::simd::*;

    let lanes = 4;
    let (a_chunks, a_remainder) = a.as_chunks::<4>();
    let (b_chunks, b_remainder) = b.as_chunks::<4>();
    let (result_chunks, result_remainder) = result.as_chunks_mut::<4>();

    // SIMD processing
    for ((a_chunk, b_chunk), result_chunk) in
        a_chunks.iter().zip(b_chunks).zip(result_chunks)
    {
        let a_simd = f32x4::from_array(*a_chunk);
        let b_simd = f32x4::from_array(*b_chunk);
        let sum = a_simd + b_simd;
        *result_chunk = sum.to_array();
    }

    // Handle remainder
    for ((a, b), result) in
        a_remainder.iter().zip(b_remainder).zip(result_remainder)
    {
        *result = a + b;
    }
}
```

Example: SIMD Search Operations

Searching through large buffers can benefit from SIMD parallelism by checking multiple elements simultaneously.

```
fn find_byte_simd(haystack: &[u8], needle: u8) -> Option<usize> {
    let (chunks, remainder) = haystack.as_chunks::<16>();

    for (i, chunk) in chunks.iter().enumerate() {
        for (j, &byte) in chunk.iter().enumerate() {
            if byte == needle {
                return Some(i * 16 + j);
            }
        }
    }

    let offset = chunks.len() * 16;
    remainder.iter()
        .position(|&b| b == needle)
```

```
.map(|pos| offset + pos)
}
```

Example: SIMD Reduction Operations

Reduction operations like sum benefit from SIMD by accumulating multiple lanes in parallel before the final reduction.

```
fn sum_f32_vectorized(data: &[f32]) -> f32 {
    let (chunks, remainder) = data.as_chunks::<8>();

    let mut sums = [0.0f32; 8];

    for chunk in chunks {
        for (i, &value) in chunk.iter().enumerate() {
            sums[i] += value;
        }
    }

    let chunk_sum: f32 = sums.iter().sum();
    let remainder_sum: f32 = remainder.iter().sum();

    chunk_sum + remainder_sum
}
```

Example: Combining Parallelism with SIMD

Rayon's parallel iterators combined with SIMD operations enable multi-core data parallelism for maximum throughput.

```
#[cfg(feature = "rayon")]
fn parallel_simd_transform(data: &mut [f32]) {
    use rayon::prelude::*;

    data.par_chunks_mut(1024)
        .for_each(|chunk| {
            for value in chunk {
                *value = value.sqrt();
            }
        });
}
```

Example: Auto-Vectorization

Simple loops are often auto-vectorized by the compiler. Writing clear, simple code can be as fast as manual SIMD.

```
fn scale_values(data: &mut [f32], scale: f32) {
    // Compiler can auto-vectorize this loop
```

```

    for value in data {
        *value *= scale;
    }
}

```

Example: Dot Product with Chunking

Dot products are fundamental linear algebra operations that benefit significantly from SIMD processing.

```

fn dot_product_chunks(a: &[f32], b: &[f32]) -> f32 {
    let (a_chunks, a_rem) = a.as_chunks::<4>();
    let (b_chunks, b_rem) = b.as_chunks::<4>();

    let mut sum = 0.0;

    // Process 4 elements at a time
    for (a_chunk, b_chunk) in a_chunks.iter().zip(b_chunks) {
        for i in 0..4 {
            sum += a_chunk[i] * b_chunk[i];
        }
    }

    // Handle remainder
    for (a_val, b_val) in a_rem.iter().zip(b_rem) {
        sum += a_val * b_val;
    }

    sum
}

```

Example: Image Processing Pipeline

Converting RGB to grayscale is embarrassingly parallel and benefits from SIMD processing of pixel data.

```

fn grayscale_simd(rgb_data: &[u8], output: &mut [u8]) {
    assert_eq!(rgb_data.len() % 3, 0);
    assert_eq!(output.len(), rgb_data.len() / 3);

    for (i, pixel) in rgb_data.chunks_exact(3).enumerate() {
        let r = pixel[0] as u32;
        let g = pixel[1] as u32;
        let b = pixel[2] as u32;

        // Weighted average for luminance
        let gray = ((r * 77 + g * 150 + b * 29) >> 8) as u8;
        output[i] = gray;
    }
}

```

Example: SIMD-Friendly Data Layouts

Structure-of-arrays layout is more SIMD-friendly than array-of-structures for vector operations.

```
#[repr(C)]
struct Vec3 {
    x: f32,
    y: f32,
    z: f32,
}

fn normalize_vectors(vectors: &mut [Vec3]) {
    for v in vectors {
        let len = (v.x * v.x + v.y * v.y + v.z * v.z).sqrt();
        if len > 0.0 {
            v.x /= len;
            v.y /= len;
            v.z /= len;
        }
    }
}
```

SIMD principles: 1. **Use as_chunks for aligned access:** Processes fixed-size chunks efficiently 2. **Handle remainders explicitly:** Don't forget the last few elements 3. **Align data structures:** Use `#[repr(align(N))]` for better SIMD performance 4. **Let the compiler auto-vectorize:** Simple loops often vectorize automatically 5. **Profile before optimizing:** SIMD isn't always faster for small datasets

Pattern 6: Advanced Slice Patterns

Problem: Removing elements during iteration requires complex index tracking. Transforming vectors in-place while filtering requires multiple passes.

Solution: Use `drain()` for removing ranges while iterating. Apply `retain_mut()` for in-place filtering with mutation.

Why It Matters: These patterns enable complex transformations without temporary allocations. In-place compaction with swap avoids $O(N^2)$ repeated deletions.

Use Cases: In-place vector compaction, gap buffer implementations, sorting implementations with pivot splitting, parallel processing with `split_at_mut`, memory-efficient filtering, zero-copy type conversions (e.g., `&[u8]` to `&[u32]`), efficient element removal patterns.

Example: In-Place Vector Compaction

Compacting a vector by removing unwanted elements without allocation uses a two-pointer technique with swap operations.

```
fn compact_vector(vec: &mut Vec<Item>) {
    let mut write_index = 0;
```

```

    for read_index in 0..vec.len() {
        if vec[read_index].should_keep() {
            if read_index != write_index {
                vec.swap(read_index, write_index);
            }
            write_index += 1;
        }
    }

    vec.truncate(write_index);
}

```

Example: Extracting Elements with Drain

Draining elements that match a predicate into a separate vector while preserving the original's capacity.

```

fn extract_matching(vec: &mut Vec<Item>, predicate: impl Fn(&Item) -> bool) -> Vec<Item> {
    let mut extracted = Vec::new();
    let mut i = 0;

    while i < vec.len() {
        if predicate(&vec[i]) {
            extracted.push(vec.remove(i));
        } else {
            i += 1;
        }
    }

    extracted
}

```

Example: Splice for Range Replacement

`splice` removes a range and replaces it with new elements in a single operation, more efficient than separate remove and insert.

```

fn replace_range(vec: &mut Vec<i32>, start: usize, end: usize, replacement: &[i32]) {
    vec.splice(start..end, replacement.iter().copied());
}

```

Example: Efficient Mid-Vector Insertion

`split_off` divides a vector at an index, enabling efficient insertion without shifting all elements twice.

```

fn insert_slice_at(vec: &mut Vec<u8>, index: usize, data: &[u8]) {
    let tail = vec.split_off(index);
    vec.extend_from_slice(data);
    vec.extend_from_slice(&tail);
}

```

Example: Sliding Window Maximum with Deque

VecDeque enables efficient double-ended operations crucial for algorithms like sliding window maximum.

```
use std::collections::VecDeque;

fn sliding_window_max(values: &[i32], window_size: usize) -> Vec<i32> {
    let mut result = Vec::new();
    let mut deque = VecDeque::new();

    for (i, &value) in values.iter().enumerate() {
        // Remove elements outside window
        while deque.front().map_or(false, |&idx| idx <= i.saturating_sub(window_size)) {
            deque.pop_front();
        }

        // Remove smaller elements from back
        while deque.back().map_or(false, |&idx| values[idx] < value) {
            deque.pop_back();
        }

        deque.push_back(i);

        if i >= window_size - 1 {
            result.push(values[*deque.front().unwrap()]);
        }
    }

    result
}
```

Example: Circular Buffer Implementation

Circular buffers use modular arithmetic to wrap indices, providing efficient fixed-size queues without shifting elements.

```
struct CircularBuffer<T> {
    data: Vec<T>,
    head: usize,
    tail: usize,
    size: usize,
}

impl<T: Default + Clone> CircularBuffer<T> {
    fn new(capacity: usize) -> Self {
        CircularBuffer {
            data: vec![T::default(); capacity],
            head: 0,
            tail: 0,
            size: 0,
        }
    }
}
```

```

    }

    fn push(&mut self, item: T) {
        self.data[self.tail] = item;
        self.tail = (self.tail + 1) % self.data.len();

        if self.size < self.data.len() {
            self.size += 1;
        } else {
            self.head = (self.head + 1) % self.data.len();
        }
    }

    fn as_slices(&self) -> (&[T], &[T]) {
        if self.head <= self.tail {
            (&self.data[self.head..self.tail], &[])
        } else {
            (&self.data[self.head..], &self.data[..self.tail])
        }
    }
}

```

Example: Copy-Free Slice Swapping

`std::mem::swap` enables efficient element-wise swapping between two mutable slices without temporary storage.

```

fn interleave_slices(a: &mut [u8], b: &mut [u8]) {
    assert_eq!(a.len(), b.len());

    for (a_val, b_val) in a.iter_mut().zip(b.iter_mut()) {
        std::mem::swap(a_val, b_val);
    }
}

```

Example: Self-Referential Duplication

`extend_from_within` copies a range from within the vector and appends it, useful for repeating patterns.

```

fn duplicate_segment(vec: &mut Vec<u8>, start: usize, end: usize) {
    vec.extend_from_within(start..end);
}

```

Example: Binary Partitioning

Lomuto partition scheme efficiently separates elements based on a predicate, foundational for quicksort and selection algorithms.

```

fn partition_by_sign(values: &mut [i32]) -> usize {
    let mut left = 0;
    let mut right = values.len();

    while left < right {
        if values[left] >= 0 {
            left += 1;
        } else {
            right -= 1;
            values.swap(left, right);
        }
    }

    left
}

```

Example: Three-Way Partitioning

Dutch National Flag algorithm partitions into three regions (less than, equal to, greater than) in a single pass.

```

fn partition_three_way(values: &mut [i32], pivot: i32) -> (usize, usize) {
    let mut low = 0;
    let mut mid = 0;
    let mut high = values.len();

    while mid < high {
        if values[mid] < pivot {
            values.swap(low, mid);
            low += 1;
            mid += 1;
        } else if values[mid] > pivot {
            high -= 1;
            values.swap(mid, high);
        } else {
            mid += 1;
        }
    }

    (low, high)
}

```

Advanced patterns guidelines: 1. **Use drain for selective removal:** More efficient than repeated remove() 2. **Prefer retain over drain+filter:** Built-in and optimized 3. **Use VecDeque for double-ended operations:** Better than Vec for queue operations 4. **Implement circular buffers for fixed-size windows:** Avoid shifting elements 5. **Partition in-place when possible:** Avoids allocation

Summary

Vectors and slices are fundamental to Rust data processing. By mastering capacity management, algorithmic operations, chunking patterns, zero-copy techniques, and SIMD optimizations, you can write high-performance code that competes with C/C++ while maintaining Rust's safety guarantees.

Key takeaways: 1. Pre-allocate with `with_capacity` to eliminate reallocations 2. Use slice algorithms (`binary_search`, `partition_point`, `sort_unstable`) for common operations 3. Leverage chunks and windows for batch processing and signal processing 4. Return slices instead of cloning to achieve zero-copy parsing 5. Use `as_chunks` and SIMD-friendly layouts for data-parallel operations 6. Profile your code—premature optimization often leads to complexity without gains

Performance dos and don'ts:

✓ **Do:** - Pre-allocate when size is known or estimable - Use `sort_unstable` for primitives - Process data in chunks for better cache locality - Return slices instead of cloning - Let the compiler auto-vectorize simple loops

✗ **Don't:** - Use `push` in loops without pre-allocating - Call `shrink_to_fit` on frequently modified vectors - Clone large slices when borrowing suffices - Implement manual SIMD without profiling first - Forget to handle remainders in chunked operations

String Processing

Strings in Rust are more complex than in many languages due to UTF-8 encoding guarantees, ownership semantics, and platform interoperability requirements. `String` is a heap-allocated, growable UTF-8 string, while `&str` is an immutable view into UTF-8 data. Understanding the type system, encoding rules, and zero-copy techniques is essential for writing efficient text processing code.

The key insight is that string type selection, allocation strategy, and understanding UTF-8's variable-length encoding dramatically impact both correctness and performance.

Pattern 1: String Type Selection

Problem: Rust offers multiple string types (`String`, `&str`, `Cow`, `OsString`, `Path`), and choosing the wrong one leads to unnecessary allocations, API inflexibility, or platform compatibility issues. Developers face the ownership vs borrowing decision at every string boundary.

Solution: Use `String` for owned, growable UTF-8 strings when you need to build or modify text. Use `&str` for borrowed string slices—the most flexible function parameter type.

Why It Matters: Type choice determines allocation patterns and API ergonomics. Functions taking `&str` work with both `String` and `&str` due to deref coercion.

Use Cases: `String` for building strings dynamically, returning from functions, storing in collections. `&str` for function parameters, parsing, zero-copy operations. `Cow<str>` for normalization, escaping, sanitization where input is often already valid. `OsString` for file paths, environment variables, FFI with OS APIs. `Path` for cross-platform path manipulation with extension extraction, parent directory operations.

Example: String - Owned and Growable

`String` is a heap-allocated, growable UTF-8 string that owns its data. Use it when you need to build strings dynamically or return strings from functions.

```
fn string_example() {
    let mut s = String::from("Hello");
    s.push_str(", World!");
    println!("{}", s);

    // Use when:
    // - Need to own the string
    // - Building strings dynamically
    // - Returning strings from functions
}
```

Example: &str - Borrowed String Slice

`&str` is an immutable view into UTF-8 data without ownership. It's the most flexible type for function parameters since `String` automatically derefs to `&str`.

```
fn str_slice_example(s: &str) {
    println!("Length: {}", s.len());

    // Use when:
    // - Read-only access needed
    // - Function parameters (most flexible)
    // - String literals
}
```

Example: Cow - Clone on Write

`Cow<str>` enables conditional allocation: borrow when possible, allocate only when modification is needed. This eliminates unnecessary allocations in common cases.

```
use std::borrow::Cow;

fn cow_example<'a>(data: &'a str, uppercase: bool) -> Cow<'a, str> {
    if uppercase {
        Cow::Owned(data.to_uppercase()) // Allocates
    } else {
        Cow::Borrowed(data) // No allocation
    }
}
```

Example: OsString/OsStr - Platform-Native Strings

Operating systems don't guarantee UTF-8. `OsString` and `OsStr` handle platform-specific encodings (UTF-16 on Windows, bytes on Unix) safely.

```
use std::ffi::OsStr, OsString;

fn os_string_example() {
    use std::env;

    for (key, value) in env::vars_os() {
        println!("{} = {}", key, value);
    }

    // Use when:
    // - Dealing with file system
    // - Environment variables
    // - FFI with OS APIs
}
```

Example: Path/PathBuf - Cross-Platform File Paths

Path and PathBuf wrap OsStr/OsString with path-specific operations and handle platform differences in path separators automatically.

```
use std::path::{Path, PathBuf};

fn path_example() {
    let path = Path::new("/tmp/foo.txt");

    println!("Extension: {:?}", path.extension());
    println!("Parent: {:?}", path.parent());
    println!("File name: {:?}", path.file_name());

    // Building paths
    let mut path_buf = PathBuf::from("/tmp");
    path_buf.push("subdir");
    path_buf.push("file.txt");

    // Use when:
    // - Working with file paths
    // - Cross-platform path manipulation
}
```

Example: Type Conversions

Understanding how to convert between string types is essential for working effectively with Rust's string ecosystem.

```
use std::borrow::Cow;
use std::ffi::OsStr;
use std::path::Path;

fn main() {
    // Demonstrate type conversions
```

```

let string = String::from("Hello");
let str_slice: &str = &string; // String -> &str (deref coercion)
let cow: Cow<str> = Cow::Borrowed(str_slice);

// String from &str
let owned: String = str_slice.to_string();

// Path conversions
let path = Path::new("file.txt");
let os_str: &OsStr = path.as_os_str();

println!("Type conversions demonstrated");
}

```

Algorithm & Design Rationale:

The string type hierarchy reflects a fundamental trade-off in systems programming:

- String vs &str:** This is the owned/borrowed dichotomy. `String` is heap-allocated and growable (similar to `Vec<u8>` but with UTF-8 guarantees). `&str` is a “view” into UTF-8 data stored elsewhere. Functions taking `&str` are maximally flexible since `String` automatically derefs to `&str`, but the reverse isn’t true.
- Cow (Clone on Write):** This is a performance optimization that delays allocation. The type `Cow<'a, str>` is an enum: either `Borrowed(&'a str)` or `Owned(String)`. When a function might or might not need to modify its input, `Cow` allows returning the borrowed input when no changes are needed, avoiding allocation entirely.
- OsString/OsStr:** Operating systems don’t guarantee UTF-8 encoding. Windows uses UTF-16, Unix systems use arbitrary byte sequences. `OsString` handles these platform differences while preserving the owned/borrowed distinction. Use these at OS boundaries: file paths, environment variables, command-line arguments.
- Path/PathBuf:** These wrap `OsStr/OsString` with path-specific operations (extension extraction, joining, parent directory). They understand path separators are platform-dependent (`/` vs `\`).

Memory Layout: - `String`: 3 words (pointer, length, capacity) - `&str`: 2 words (pointer, length) - `Cow<str>`: 4 words (discriminant + either 2-word `&str` or 3-word `String`)

Key Concepts: - `String` owns data, `&str` borrows - `Cow` optimizes by borrowing when possible, allocating only when necessary - `OsString` handles platform-specific encodings (UTF-16 on Windows, bytes on Unix) - `Path` provides platform-independent path operations with correct separator handling

Pattern 2: String Builder Pattern

Problem: Concatenating strings in loops creates $O(N^2)$ complexity—each `s = s + "text"` allocates a new string and copies all previous content. Building HTML, SQL queries, or templates with repeated `push_str()` causes multiple reallocations when capacity is exceeded.

Solution: Pre-allocate capacity with `String::with_capacity(n)` when approximate size is known. Implement builder pattern with `&mut self` returns for method chaining.

Why This Matters: Pre-allocation eliminates reallocations— $O(N)$ total time vs $O(N)$ amortized with up to $\log(N)$ reallocations. A 10KB HTML document built with 100 appends: pre-allocated uses one 10KB buffer, non-pre-allocated copies $\sim 20\text{KB}$ total (due to exponential growth). Method chaining creates fluent APIs that are both ergonomic and efficient. Builder pattern separates construction complexity from final immutable result.

Use Cases: HTML/XML generation (builders with tag methods, indentation), SQL query construction (type-safe builder preventing injection), log formatting (structured message building), template rendering (placeholder substitution), configuration file generation, protocol message assembly.

Example: Basic StringBuilder with Capacity Pre-allocation

A simple string builder that pre-allocates capacity and provides method chaining for fluent APIs. The `build()` method consumes the builder to transfer ownership without copying.

```
struct StringBuilder {
    buffer: String,
}

impl StringBuilder {
    fn new() -> Self {
        StringBuilder {
            buffer: String::new(),
        }
    }

    fn with_capacity(capacity: usize) -> Self {
        StringBuilder {
            buffer: String::with_capacity(capacity),
        }
    }

    fn append(&mut self, s: &str) -> &mut Self {
        self.buffer.push_str(s);
        self
    }

    fn append_line(&mut self, s: &str) -> &mut Self {
        self.buffer.push_str(s);
        self.buffer.push('\n');
        self
    }

    fn append_fmt(&mut self, args: std::fmt::Arguments) -> &mut Self {
        use std::fmt::Write;
        let _ = write!(&mut self.buffer, "{}", args);
        self
    }
}
```

```

fn build(self) -> String {
    self.buffer
}

fn as_str(&self) -> &str {
    &self.buffer
}

}

```

Example: Domain-Specific HTML Builder

An HTML builder wraps `StringBuilder` with domain-specific methods for tag handling and automatic indentation, making HTML generation both safe and ergonomic.

```

struct HtmlBuilder {
    builder: StringBuilder,
    indent: usize,
}

impl HtmlBuilder {
    fn new() -> Self {
        HtmlBuilder {
            builder: String::with_capacity(1024),
            indent: 0,
        }
    }

    fn open_tag(&mut self, tag: &str) -> &mut Self {
        self.write_indent();
        self.builder.append("<").append(tag).append(">\n");
        self.indent += 2;
        self
    }

    fn close_tag(&mut self, tag: &str) -> &mut Self {
        self.indent -= 2;
        self.write_indent();
        self.builder.append("</").append(tag).append(">\n");
        self
    }

    fn content(&mut self, text: &str) -> &mut Self {
        self.write_indent();
        self.builder.append(text).append("\n");
        self
    }

    fn write_indent(&mut self) {
        for _ in 0..self.indent {
            self.builder.append(" ");
        }
    }
}

```

```

fn build(self) -> String {
    self.builder.build()
}

}

```

Example: Using the Builders

Demonstrating the fluent API style enabled by method chaining with `&mut self` returns.

```

fn main() {
    // Simple string building
    let mut sb = StringBuilder::with_capacity(100);
    sb.append("Hello")
        .append(", ")
        .append("World")
        .append("!");
    println!("{}", sb.as_str());

    // HTML building
    let mut html = HtmlBuilder::new();
    html.open_tag("html")
        .open_tag("body")
        .open_tag("h1")
        .content("Welcome")
        .close_tag("h1")
        .open_tag("p")
        .content("This is a paragraph")
        .close_tag("p")
        .close_tag("body")
        .close_tag("html");

    println!("{}", html.build());
}

```

Algorithm Analysis:

The builder pattern's efficiency comes from capacity pre-allocation. When you call `String::with_capacity(n)`, Rust allocates a buffer of size `n` immediately. Subsequent `push_str()` calls are $O(1)$ operations that copy bytes into the pre-allocated buffer—no reallocation needed until capacity is exhausted.

Without pre-allocation, `String` uses an exponential growth strategy (doubling capacity when full). While this gives amortized $O(1)$ insertion, it still wastes memory and CPU on copying during growth. If you know the approximate final size, pre-allocation eliminates this overhead.

Performance Characteristics: - **With capacity:** $O(N)$ total time for N bytes inserted - **Without capacity:** $O(N)$ amortized, but with up to $\log(N)$ reallocations - **Memory:** Pre-allocated buffer may have unused capacity; this is acceptable for temporary builders

Key Patterns: - Pre-allocate capacity when size is known (or can be estimated) - Method chaining (`&mut self` returns) for fluent API - Consume builder with `build(self)` to transfer ownership without copying

Pattern 3: Zero-Copy String Operations

Problem: Parsing structured text (CSV, logs, configs) by creating owned `String` for each field wastes memory and CPU. A 10MB log file with 100K lines allocating strings for each line = 100K allocations + copying 10MB of data.

Solution: Use iterator methods that return `&str` slices borrowing from original data: `split()`, `lines()`, `split_whitespace()`. Pass slices to processing functions instead of collecting into vectors.

Why It Matters: Zero-copy parsing eliminates allocations entirely—10MB file processing uses 10MB (the file buffer) instead of 20MB (file + owned strings). String slicing is O(1)—just creating a fat pointer (2 words: pointer + length).

Use Cases: CSV/TSV parsing (split by delimiter, process fields), log analysis (pattern matching on lines), configuration file parsing (key=value splitting), protocol parsing (header extraction), text search (finding substrings without copying), streaming data processing (process-then-discard pattern).

Example: Zero-Copy Line Parser

A parser that returns iterators over string slices without allocating. Each operation borrows from the original data, making parsing extremely efficient.

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

impl<'a> LineParser<'a> {
    fn new(data: &'a str) -> Self {
        LineParser { data }
    }

    // Returns iterator over lines without allocation
    fn lines(&self) -> impl Iterator<Item = &'a str> {
        self.data.lines()
    }

    // Split by delimiter without allocation
    fn split(&self, delimiter: &str) -> impl Iterator<Item = &'a str> {
        self.data.split(delimiter)
    }

    // Extract field by index
    fn field(&self, line: &'a str, index: usize) -> Option<&'a str> {
        line.split(',').nth(index)
    }
}
```

```
    }
}
```

Example: Zero-Allocation CSV Parser

A CSV parser that returns slices into the original data rather than allocating strings for each field. Ideal for streaming or one-pass processing.

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

impl<'a> CsvParser<'a> {
    fn new(data: &'a str) -> Self {
        CsvParser { data }
    }

    fn parse(&self) -> Vec<Vec<&'a str>> {
        self.data
            .lines()
            .map(|line| line.split(',').map(|field| field.trim()).collect())
            .collect()
    }

    // Process without intermediate allocations
    fn process<F>(&self, mut f: F)
    where
        F: FnMut(&[&str]),
    {
        for line in self.data.lines() {
            let fields: Vec<&str> = line.split(',').map(|f| f.trim()).collect();
            f(&fields);
        }
    }
}
```

Example: String View with UTF-8 Boundary Checking

A safe string view that validates UTF-8 character boundaries before slicing, preventing panics from invalid slices.

```
struct StringView<'a> {
    data: &'a str,
    start: usize,
    len: usize,
}

impl<'a> StringView<'a> {
    fn new(data: &'a str, start: usize, len: usize) -> Option<Self> {
        if start + len <= data.len() && data.is_char_boundary(start) {
            if start + len == data.len() || data.is_char_boundary(start + len) {
                Some(StringView { data, start, len })
            } else {
                None
            }
        } else {
            None
        }
    }
}
```

```

        return Some(StringView { data, start, len });
    }
}
None
}

fn as_str(&self) -> &'a str {
    &self.data[self.start..self.start + self.len]
}

fn slice(&self, start: usize, len: usize) -> Option<StringView<'a>> {
    if start + len <= self.len {
        StringView::new(self.data, self.start + start, len)
    } else {
        None
    }
}
}

```

Example: Using Zero-Copy Parsers

Demonstrating how to use zero-copy parsing techniques for efficient text processing without unnecessary allocations.

```

fn main() {
    let data = "name,age,city\nAlice,30,NYC\nBob,25,LA";

    let parser = CsvParser::new(data);
    let rows = parser.parse();

    for row in &rows {
        println!("{}: {:?}", row);
    }

    // Zero-copy processing
    parser.process(|fields| {
        if fields.len() >= 2 {
            println!("Name: {}, Age: {}", fields[0], fields[1]);
        }
    });
}

// String view example
let text = "Hello, World!";
if let Some(view) = StringView::new(text, 0, 5) {
    println!("View: {}", view.as_str());
}
}

```

Algorithm Insights:

String slicing in Rust is implemented as a fat pointer: a pointer to the start byte plus a length. Creating a slice is O(1)—just two integer values. No copying occurs.

The `split()` iterator maintains state (current position) and advances through the string on each `next()` call, returning slices between delimiters. This is lazy: if you only take the first 3 items from `split()`, it never scans past the 3rd delimiter.

Critical Safety Issue: UTF-8 characters can be 1-4 bytes. Slicing in the middle of a multi-byte character would create invalid UTF-8. Rust prevents this at compile-time by making slicing operations panic if the index isn't a character boundary. Use `is_char_boundary()` to check before slicing, or use `char_indices()` which returns valid indices.

Performance: - Creating a slice: O(1) - Iterating with `split()`: O(N) where N is string length - Memory: Zero allocations (slices borrow from original)

Key Techniques: - Return iterators instead of vectors to avoid collecting until necessary - Use string slices (`&str`) as return types for maximum flexibility - Leverage `split()`, `lines()`, `split_whitespace()` for zero-copy splitting - Always check `is_char_boundary()` before manual slicing with byte indices

Pattern 4: Cow for Conditional Allocation

Problem: Functions that sometimes modify input (escape HTML entities, normalize whitespace, strip prefixes) and sometimes don't face an allocation dilemma. Always allocating wastes memory when input is already valid.

Solution: Return `Cow<str>` which is either `Borrowed(&str)` or `Owned(String)`. Implement two-phase algorithm: fast-path check if modification needed (scan for special characters, extra whitespace, etc), then return `Cow::Borrowed(input)` for no-op case.

Why This Matters: The fast-path check (O(N) scan) is faster than allocation + copy. For already-normalized input (common in production), `Cow` returns immediately with zero allocation. Web server escaping HTML: if 90% of inputs have no special characters, `Cow` eliminates 90% of allocations. URL normalization processing millions of requests: `Cow` saves GB of allocations. Even worst case (modification needed) matches always-allocating performance while best case is 10-100x faster.

Use Cases: HTML escaping (only allocate if <>&" present), whitespace normalization (only if multiple spaces found), case conversion (only if mixed case), prefix/suffix stripping (only if present), path canonicalization, validation with optional sanitization.

Example: Normalize Whitespace with Cow

A two-phase algorithm: first check if normalization is needed, then only allocate if multiple consecutive spaces are found.

```
use std::borrow::Cow;

fn normalize_whitespace(s: &str) -> Cow<str> {
    let mut prev_was_space = false;
    let mut needs_normalization = false;
```

```

for c in s.chars() {
    if c.is_whitespace() {
        if prev_was_space {
            needs_normalization = true;
            break;
        }
        prev_was_space = true;
    } else {
        prev_was_space = false;
    }
}

if !needs_normalization {
    return Cow::Borrowed(s);
}

// Build normalized string
let mut result = String::with_capacity(s.len());
let mut prev_was_space = false;

for c in s.chars() {
    if c.is_whitespace() {
        if !prev_was_space {
            result.push(' ');
            prev_was_space = true;
        }
    } else {
        result.push(c);
        prev_was_space = false;
    }
}

Cow::Owned(result)
}

```

Example: Conditional HTML Escaping

Check if any special characters exist before allocating. Most strings don't need escaping, so this fast-path check saves allocations.

```

use std::borrow::Cow;

fn escape_html(s: &str) -> Cow<str> {
    if !s.contains(&['<', '>', '&', '"', '\''][..]) {
        return Cow::Borrowed(s);
    }

    let mut escaped = String::with_capacity(s.len() + 20);

    for c in s.chars() {
        match c {

```

```

        '<' => escaped.push_str("&lt;"),
        '>' => escaped.push_str("&gt;"),
        '&' => escaped.push_str("&amp;"),
        '\"' => escaped.push_str("&quot;"),
        '\'' => escaped.push_str("&#39;"),
        _ => escaped.push(c),
    }
}

Cow::Owned(escaped)
}

```

Example: Strip Prefix/Suffix Without Allocation

Only allocate a new string if prefix or suffix actually exists. Otherwise return the original slice.

```

use std::borrow::Cow;

fn strip_affixes<'a>(s: &'a str, prefix: &str, suffix: &str) -> Cow<'a, str> {
    let mut start = 0;
    let mut end = s.len();

    if s.starts_with(prefix) {
        start = prefix.len();
    }

    if s.ends_with(suffix) {
        end = end.saturating_sub(suffix.len());
    }

    if start == 0 && end == s.len() {
        Cow::Borrowed(s)
    } else {
        Cow::Owned(s[start..end].to_string())
    }
}

```

Example: Conditional Case Normalization

Check if the string is already lowercase before allocating for conversion. Avoids unnecessary work for already-normalized input.

```

use std::borrow::Cow;

fn to_lowercase_if_needed(s: &str) -> Cow<str> {
    if s.chars().all(|c| !c.is_uppercase()) {
        Cow::Borrowed(s)
    } else {
        Cow::Owned(s.to_lowercase())
    }
}

```

```
}
```

Example: Using Cow Functions

Demonstrating how Cow eliminates allocations when input is already in the desired format.

```
fn main() {
    // Whitespace normalization
    let s1 = "hello world";
    let s2 = "hello  world"; // Extra space

    println!("s1: {:?}", normalize_whitespace(s1)); // Borrowed
    println!("s2: {:?}", normalize_whitespace(s2)); // Owned

    // HTML escaping
    let safe = "Hello World";
    let unsafe_text = "Hello <b>World</b>";

    println!("Safe: {:?}", escape_html(safe)); // Borrowed
    println!("Unsafe: {:?}", escape_html(unsafe_text)); // Owned

    // Prefix/suffix stripping
    println!("{:?}", strip_affixes("hello", "", "")); // Borrowed
    println!("{:?}", strip_affixes("[hello]", "[", "]")); // Owned

    // Case normalization
    println!("{:?}", to_lowercase_if_needed("hello")); // Borrowed
    println!("{:?}", to_lowercase_if_needed("Hello")); // Owned
}
```

Algorithm Strategy:

The **Cow** pattern implements a two-phase algorithm:

1. **Scan Phase:** Iterate through the input to detect if modification is needed. This is a read-only $O(N)$ pass. If no modification is required, return `Cow::Borrowed(input)` immediately—zero allocation.
2. **Build Phase:** If modification is needed, allocate a new `String`, perform transformations, and return `Cow::Owned(result)`.

The key insight: the scan phase cost ($O(N)$) is dominated by the allocation and copy cost we're trying to avoid. Even though we potentially traverse the string twice (once to check, once to modify), this is faster than always allocating.

Trade-off Analysis: - **Best case** (no modification needed): $O(N)$ scan, zero allocation - **Worst case** (modification needed): $O(N)$ scan + $O(N)$ build = $O(N)$ total, one allocation - **Always allocating:** $O(N)$ build, one allocation (even when unnecessary)

For high-frequency operations where the input is often already in the desired form, `Cow` provides significant savings.

Key Patterns: - Implement fast-path check before allocating (e.g., `contains()` check for escaping) - Return `Cow::Borrowed` for no-op transformations - Return `Cow::Owned` only when changes made - Pre-allocate capacity for owned variants when size is known (`s.len() + overhead`)

Pattern 5: UTF-8 Validation and Repair

Problem: UTF-8 is variable-length (1-4 bytes per character), but external data sources don't guarantee validity. Files may have encoding mismatches (Latin-1 labeled as UTF-8), network data can be corrupted, FFI receives arbitrary bytes.

Solution: Validate external data with `str::from_utf8()` for strict errors or `String::from_utf8_lossy()` to replace invalid sequences with `◆`. Implement custom validators understanding UTF-8 structure (first byte determines length, continuation bytes match `10xxxxxx`, detect overlong encodings, reject surrogates).

Why It Matters: Invalid UTF-8 crashes programs or creates security vulnerabilities (overlong encodings bypass filters). A byte index mid-character panics: "hello"[1..4] is fine, "h llo"[1..4] panics (  is 2 bytes).

Use Cases: File I/O from unknown encodings, network protocol parsing, FFI with C strings, data recovery tools, text editors (proper cursor movement), terminal emulators (width calculation), web servers (sanitizing input), internationalization (proper truncation/reversal).

Example: Lossy UTF-8 Validation

For handling potentially invalid data, `from_utf8_llossy` replaces invalid byte sequences with the replacement character (◆). This is useful for recovery and logging.

```
fn validate_utf8_lossy(data: &[u8]) -> String {
    String::from_utf8_lossy(data).into_owned()
}
```

Example: Strict UTF-8 Validation

When you need to know if data is valid UTF-8, use `from_utf8` which returns an error for invalid sequences.

```
fn validate_utf8_strict(data: &[u8]) -> Result<&str, std::str::Utf8Error> {
    std::str::from_utf8(data)
}
```

Example: Custom UTF-8 Validator

A comprehensive UTF-8 validator that detects overlong encodings and provides detailed error positions. This demonstrates the complete UTF-8 validation algorithm.

```
struct Utf8Validator<'a> {
    data: &'a [u8],
}

impl<'a> Utf8Validator<'a> {
    fn new(data: &'a [u8]) -> Self {
        Utf8Validator { data }
    }

    fn validate(&self) -> Result<&'a str, Utf8Error> {
        let mut pos = 0;

        while pos < self.data.len() {
            match self.decode_char(pos) {
                Ok((_, next_pos)) => pos = next_pos,
                Err(error_pos) => {
                    return Err(Utf8Error {
                        valid_up_to: error_pos,
                        error_len: self.error_length(error_pos),
                    });
                }
            }
        }
    }

    unsafe { Ok(std::str::from_utf8_unchecked(self.data)) }
}

fn decode_char(&self, pos: usize) -> Result<(char, usize), usize> {
    if pos >= self.data.len() {
        return Err(pos);
    }

    let first = self.data[pos];

    // 1-byte sequence (ASCII)
    if first < 0x80 {
        return Ok((first as char, pos + 1));
    }

    // 2-byte sequence
    if first & 0xE0 == 0xC0 {
        if pos + 1 >= self.data.len() {
            return Err(pos);
        }
        let second = self.data[pos + 1];
        if second & 0xC0 != 0x80 {
            return Err(pos);
        }
        let ch = ((first as u32 & 0x1F) << 6) | (second as u32 & 0x3F);
        if ch < 0x80 {
            return Err(pos); // Overlong encoding
        }
    }
}
```

```

        return Ok((char::from_u32(ch).ok_or(pos)?, pos + 2));
    }

    // 3-byte sequence
    if first & 0xF0 == 0xE0 {
        if pos + 2 >= self.data.len() {
            return Err(pos);
        }
        let second = self.data[pos + 1];
        let third = self.data[pos + 2];
        if second & 0xC0 != 0x80 || third & 0xC0 != 0x80 {
            return Err(pos);
        }
        let ch = ((first as u32 & 0x0F) << 12)
            | ((second as u32 & 0x3F) << 6)
            | (third as u32 & 0x3F);
        if ch < 0x800 {
            return Err(pos); // Overlong encoding
        }
        return Ok((char::from_u32(ch).ok_or(pos)?, pos + 3));
    }

    // 4-byte sequence
    if first & 0xF8 == 0xF0 {
        if pos + 3 >= self.data.len() {
            return Err(pos);
        }
        let bytes = &self.data[pos..pos + 4];
        if bytes[1] & 0xC0 != 0x80
            || bytes[2] & 0xC0 != 0x80
            || bytes[3] & 0xC0 != 0x80
        {
            return Err(pos);
        }
        let ch = ((first as u32 & 0x07) << 18)
            | ((bytes[1] as u32 & 0x3F) << 12)
            | ((bytes[2] as u32 & 0x3F) << 6)
            | (bytes[3] as u32 & 0x3F);
        if ch < 0x10000 || ch > 0x10FFFF {
            return Err(pos); // Overlong or out of range
        }
        return Ok((char::from_u32(ch).ok_or(pos)?, pos + 4));
    }

    Err(pos)
}

fn error_length(&self, pos: usize) -> Option<usize> {
    if pos >= self.data.len() {
        return None;
    }

    let first = self.data[pos];

```

```

    if first < 0x80 {
        Some(1)
    } else if first & 0xE0 == 0xC0 {
        Some(2)
    } else if first & 0xF0 == 0xE0 {
        Some(3)
    } else if first & 0xF8 == 0xF0 {
        Some(4)
    } else {
        Some(1)
    }
}
}

#[derive(Debug)]
struct Utf8Error {
    valid_up_to: usize,
    error_len: Option<usize>,
}

```

Example: Using UTF-8 Validators

Demonstrating validation strategies for different scenarios: strict validation, lossy conversion, and detailed error reporting.

```

fn main() {
    // Valid UTF-8
    let valid = "Hello, 世界!".as_bytes();
    let validator = Utf8Validator::new(valid);
    assert!(validator.validate().is_ok());

    // Invalid UTF-8
    let invalid = &[0xFF, 0xFE, 0xFD];
    let validator = Utf8Validator::new(invalid);
    match validator.validate() {
        Ok(_) => println!("Valid"),
        Err(e) => println!("Invalid UTF-8 at position {}", e.valid_up_to),
    }

    // Lossy conversion
    let lossy = validate_utf8_lossy(invalid);
    println!("Lossy: {}", lossy);
}

```

Algorithm Walkthrough:

The validator implements a state machine that processes bytes sequentially:

- 1. First Byte Analysis:** Examine the bit pattern to determine character length:

- **0xxxxxxx**: 1-byte ASCII (U+0000-U+007F)
- **110xxxxx**: 2-byte sequence expected

- **1110xxxx**: 3-byte sequence expected
- **11110xxx**: 4-byte sequence expected
- Any other pattern is invalid

2. Continuation Byte Validation: Each subsequent byte must match **10xxxxxxxx**. The validator checks this with `byte & 0xC0 == 0x80`.

3. Code Point Reconstruction: Extract bits from each byte and combine:

- For 2-byte: `((first & 0x1F) << 6) | (second & 0x3F)`
- For 3-byte: `((first & 0x0F) << 12) | ((second & 0x3F) << 6) | (third & 0x3F)`
- For 4-byte: `((first & 0x07) << 18) | ...`

4. Overlong Detection: Each range has a minimum code point:

- 2-byte sequences must encode U+0080 or higher (if lower, use 1-byte)
- 3-byte sequences must encode U+0800 or higher
- 4-byte sequences must encode U+10000 or higher This prevents security attacks that use overlong encodings to bypass filters.

5. Range Validation: Reject surrogate pairs (U+D800-U+DFFF) and values > U+10FFFF.

Performance: - Time: O(N) single pass through bytes - Space: O(1) constant memory for state - Early termination on first error

Key Concepts: - UTF-8 is self-synchronizing: you can find character boundaries by scanning for non-**10xxxxxxxx** bytes - Overlong encoding detection prevents security exploits - Surrogate detection (U+D800-U+DFFF) rejects UTF-16 artifacts - Lossy conversion (`from_utf8_lossy`) replaces invalid sequences with U+FFFD (❖) - Strict validation (`from_utf8`) returns `Err` with the error position

Pattern 6: Character and Grapheme Iteration

Problem: What users perceive as “one character” isn’t what Rust’s `.chars()` sees. Emoji like “👨‍👩‍👧‍👦” (family) are multiple code points joined with Zero-Width Joiners.

Solution: Understand three iteration levels: bytes (`.bytes()`), characters (`.chars()`), graphemes (`unicode-segmentation::graphemes()`). Use bytes only for serialization/protocols.

Why It Matters: Byte iteration sees 4 bytes for “👋”, char iteration sees 1 code point, grapheme iteration sees 1 user-perceived character. But “👨‍👩‍👧‍👦” is 7 code points (4 emoji + 3 ZWJ), 1 grapheme.

Use Cases: Text editors (cursor movement across graphemes), terminal output (width calculation for alignment), string truncation (safe at grapheme boundaries), text reversal (preserve composed characters), search/replace (whole grapheme), length display (user-perceived count not bytes), internationalization.

Example: Three Levels of String Iteration

Demonstrating the difference between bytes, characters (code points), and grapheme clusters—each serves different purposes.

```

use unicode_segmentation::UnicodeSegmentation;

fn analyze_string(s: &str) {
    println!("String: {:?}", s);
    println!("Byte length: {}", s.len());

    // Byte iteration
    println!("\nBytes:");
    for (i, byte) in s.bytes().enumerate() {
        print!("{:02X} ", byte);
        if (i + 1) % 8 == 0 {
            println!();
        }
    }
    println!();

    // Character (code point) iteration
    println!("\nCharacters (code points):");
    for (i, ch) in s.chars().enumerate() {
        println!("{}: '{}' (U+{:04X})", i, ch, ch as u32);
    }

    // Grapheme cluster iteration (requires unicode-segmentation crate)
    println!("\nGrapheme clusters:");
    for (i, grapheme) in s.graphemes(true).enumerate() {
        println!("{}: '{}'", i, grapheme);
    }

    println!("\nChar count: {}", s.chars().count());
    println!("Grapheme count: {}", s.graphemes(true).count());
}

```

Example: Display Width Calculation

Implements East Asian Width rules for terminal alignment. CJK characters and fullwidth forms occupy two columns.

```

fn display_width(s: &str) -> usize {
    s.chars().map(|c| {
        let cp = c as u32;
        // Simplified: full-width chars count as 2
        if (0x1100..=0x115F).contains(&cp)      // Hangul Jamo
            || (0xE80..=0xFFFF).contains(&cp)    // CJK
            || (0xAC00..=0xD7AF).contains(&cp)   // Hangul Syllables
            || (0xFF00..=0xFF60).contains(&cp)   // Fullwidth Forms
        {
            2
        } else {
            1
        }
    })
}

```

```
    }).sum()
}
```

Example: Safe String Truncation

Truncating at character boundaries prevents corrupting multi-byte UTF-8 sequences. Uses `char_indices()` to find safe cut points.

```
fn truncate_at_char(s: &str, max_chars: usize) -> &str {
    match s.char_indices().nth(max_chars) {
        Some((idx, _)) => &s[..idx],
        None => s,
    }
}
```

Example: Grapheme-Aware Truncation

For user-facing text, truncate at grapheme cluster boundaries to avoid breaking composed characters or emoji sequences.

```
use unicode_segmentation::UnicodeSegmentation;

fn truncate_at_grapheme(s: &str, max_graphemes: usize) -> &str {
    s.graphemes(true)
        .take(max_graphemes)
        .collect::<String>()
        .as_str()
}
```

Example: Reversing While Preserving Graphemes

String reversal that keeps grapheme clusters intact—essential for text editors and international text processing.

```
use unicode_segmentation::UnicodeSegmentation;

fn reverse_graphemes(s: &str) -> String {
    s.graphemes(true).rev().collect()
}
```

Example: Using Character Iteration Techniques

Demonstrating different iteration levels on ASCII, multi-byte characters, complex emoji, and CJK text.

```
fn main() {
    // Simple ASCII
    analyze_string("Hello");
    println!("{}\n{}", "=".repeat(50));
```

```

// Multi-byte UTF-8
analyze_string("Hello");
println!("\n{}", "=" .repeat(50));

// Emoji with modifier
analyze_string("👨‍👩‍👧‍👦"); // Family emoji
println!("\n{}", "=" .repeat(50));

// Korean text
let korean = "안녕하세요";
println!("Korean: {}", korean);
println!("Display width: {}", display_width(korean));

// Truncation
let text = "Hello, 世界! 🙌";
println!("\nOriginal: {}", text);
println!("Truncated (5 chars): {}", truncate_at_char(text, 5));

// Reversal
let text = "café";
println!("\nOriginal: {}", text);
println!("Reversed: {}", reverse_graphemes(text));
}

```

Understanding the Three Levels:

1. **Bytes** (`s.bytes()`): Raw UTF-8 bytes. “é” encoded as U+00E9 is two bytes: `0xC3 0xA9`. This is the lowest level—useful for serialization and binary protocols, but meaningless for text processing.
2. **Characters** (`s.chars()`): Unicode code points. “é” can be:
 - Single code point: U+00E9 (precomposed)
 - Two code points: U+0065 (e) + U+0301 (combining acute accent) Both representations are valid and display identically. `chars()` yields one or two items depending on the representation.
3. **Graphemes** (`s.graphemes(true)`): User-perceived characters following Unicode segmentation rules. “é” is always one grapheme, regardless of internal representation. Complex emoji like “👨‍👩‍👧‍👦” are composed of base emoji joined with Zero Width Joiners (ZWJ), but form a single grapheme.

East Asian Width: Characters have display widths in terminal applications. ASCII is width 1, but CJK (Chinese/Japanese/Korean) characters and emoji are width 2. The `display_width()` function estimates this using Unicode ranges.

Truncation Safety: Never truncate using byte indices unless you verify boundaries. Use `char_indices()` which returns `(byte_index, char)` pairs where `byte_index` is guaranteed to be a valid UTF-8 boundary.

Key Concepts: - Bytes < Characters < Graphemes (increasingly high-level abstractions) - Grapheme clusters preserve user-perceived text units - East Asian Width (UAX #11) affects terminal display

calculations - Always truncate at grapheme or character boundaries, never bytes - Use `char_indices()` for safe byte-index iteration

Pattern 7: String Parsing State Machines

Problem: Parsing structured text (source code, protocols, markup) with simple string methods leads to complex nested loops and fragile conditional logic. Tracking whether you're "inside a string" or "in a comment" requires multiple boolean flags.

Solution: Model the parser as a finite state machine with explicit states (Start, InString, InComment, InNumber, etc.). Each state handles specific characters and transitions to new states.

Why It Matters: State machines make parsing logic explicit and declarative—you can visualize the automaton on paper. Adding a new token type means adding a state and transitions, not hunting through conditionals.

Use Cases: Lexers for programming languages (keywords, operators, literals, comments), protocol parsers (HTTP headers, binary formats), markup languages (Markdown, HTML fragments), configuration file parsers (TOML, INI), CSV/TSV with escaping, syntax highlighting (real-time tokenization), code completion (incomplete token recovery), template languages (text + interpolation), log parsing (structured formats).

Examples

```
//=====
// Pattern Lexer (complete)
//=====

#[derive(Debug, PartialEq, Clone)]
enum Token {
    Identifier(String),
    Number(f64),
    String(String),
    Operator(String),
    Keyword(String),
    Whitespace,
    Comment(String),
    Invalid(char),
}

#[derive(Debug, PartialEq)]
enum LexerState {
    Start,
    InIdentifier,
    InNumber,
    InString,
    InComment,
    InOperator,
}

struct Lexer {
    input: Vec<char>,
```

```
    pos: usize,
    state: LexerState,
    current_token: String,
}

impl Lexer {
    fn new(input: &str) -> Self {
        Self {
            input: input.chars().collect(),
            pos: 0,
            state: LexerState::Start,
            current_token: String::new(),
        }
    }

    fn tokenize(&mut self) -> Vec<Token> {
        let mut tokens = Vec::new();

        while self.pos < self.input.len() {
            if let Some(token) = self.next_token() {
                if !matches!(token, Token::Whitespace) {
                    tokens.push(token);
                }
            }
        }

        tokens
    }

    fn next_token(&mut self) -> Option<Token> {
        let ch = self.current_char?;

        match self.state {
            LexerState::Start => self.handle_start(ch),
            LexerState::InIdentifier => self.handle_identifier(ch),
            LexerState::InNumber => self.handle_number(ch),
            LexerState::InString => self.handle_string(ch),
            LexerState::InComment => self.handle_comment(ch),
            LexerState::InOperator => self.handle_operator(ch),
        }
    }

    fn handle_start(&mut self, ch: char) -> Option<Token> {
        match ch {
            c if c.is_whitespace() => {
                self.pos += 1;
                Some(Token::Whitespace)
            }
            c if c.is_alphanumeric() || c == '_' => {
                self.state = LexerState::InIdentifier;
                self.current_token.push(c);
                self.pos += 1;
                None
            }
        }
    }
}
```

```

    }

    c if c.is_numeric() => {
        self.state = LexerState::InNumber;
        self.current_token.push(c);
        self.pos += 1;
        None
    }
    "" => {
        self.state = LexerState::InString;
        self.pos += 1;
        None
    }
    /* if self.peek() == Some('/') => {
        self.state = LexerState::InComment;
        self.pos += 2; // Skip //
        None
    }
    c if "+-*<>=!=|".contains(c) => {
        self.state = LexerState::InOperator;
        self.current_token.push(c);
        self.pos += 1;
        None
    }
    c => {
        self.pos += 1;
        Some(Token::Invalid(c))
    }
}

fn handle_identifier(&mut self, ch: char) -> Option<Token> {
    if ch.is_alphanumeric() || ch == '_' {
        self.current_token.push(ch);
        self.pos += 1;
        None
    } else {
        let token = self.finish_identifier();
        self.state = LexerState::Start;
        Some(token)
    }
}

fn handle_number(&mut self, ch: char) -> Option<Token> {
    if ch.is_numeric() || ch == '.' {
        self.current_token.push(ch);
        self.pos += 1;
        None
    } else {
        let token = Token::Number(
            self.current_token.parse().unwrap_or(0.0)
        );
        self.current_token.clear();
        self.state = LexerState::Start;
    }
}

```

```

        Some(token)
    }

fn handle_string(&mut self, ch: char) -> Option<Token> {
    if ch == '"' {
        let token = Token::String(self.current_token.clone());
        self.current_token.clear();
        self.state = LexerState::Start;
        self.pos += 1;
        Some(token)
    } else {
        self.current_token.push(ch);
        self.pos += 1;
        None
    }
}

fn handle_comment(&mut self, ch: char) -> Option<Token> {
    if ch == '\n' {
        let token = Token::Comment(self.current_token.clone());
        self.current_token.clear();
        self.state = LexerState::Start;
        Some(token)
    } else {
        self.current_token.push(ch);
        self.pos += 1;
        None
    }
}

fn handle_operator(&mut self, ch: char) -> Option<Token> {
    // Multi-char operators: ==, !=, <=, >=, &&, ||
    let two_char = format!("{}{}", self.current_token, ch);
    if matches!(two_char.as_str(), "==" | "!=" | "<=" | ">=" | "&&" | "||") {
        self.current_token = two_char;
        self.pos += 1;
        let token = Token::Operator(self.current_token.clone());
        self.current_token.clear();
        self.state = LexerState::Start;
        Some(token)
    } else {
        let token = Token::Operator(self.current_token.clone());
        self.current_token.clear();
        self.state = LexerState::Start;
        Some(token)
    }
}

fn finish_identifier(&mut self) -> Token {
    let keywords = ["if", "else", "while", "for", "return", "fn", "let"];
    let token = if keywords.contains(&self.current_token.as_str()) {

```

```

        Token::Keyword(self.current_token.clone())
    } else {
        Token::Identifier(self.current_token.clone())
    };

    self.current_token.clear();
    token
}

fn current_char(&self) -> Option<char> {
    self.input.get(self.pos).copied()
}

fn peek(&self) -> Option<char> {
    self.input.get(self.pos + 1).copied()
}
}

fn main() {
    let code = r#"
        fn main() {
            let x = 42;
            if x == 42 {
                return x + 10;
            }
        }
        // This is a comment
    "#;

    let mut lexer = Lexer::new(code);
    let tokens = lexer.tokenize();

    for token in tokens {
        println!("{}:?", token);
    }
}

```

Key Patterns: - State machine with explicit states - Lookahead with `peek()` - Multi-character token recognition - Keyword vs identifier discrimination

Pattern 8: URL Parser State Machine

Problem: Parsing URLs with regex or simple string splits is fragile and error-prone. URLs have complex structure (scheme, authority, path, query, fragment) with context-dependent delimiters—“:” after scheme vs in authority, “//” for authority vs “/” for path, “?” for query, “#” for fragment.

Solution: Use a state machine with explicit states (Scheme, AfterScheme, Authority, Path, Query, Fragment) that transitions based on delimiter characters. Parse in one pass, accumulating characters into buffers.

Why It Matters: State machine parsing is faster (single pass), more maintainable (explicit transitions), and RFC-compliant (handles all edge cases). Adding support for new URL schemes or components means adding states/transitions, not rewriting regex.

Use Cases: Web frameworks (route parsing, request URL decomposition), HTTP clients (validating and normalizing URLs), web scraping (extracting links, resolving relative URLs), URL shorteners (parsing and validating input), API gateways (routing based on URL structure), browser implementations (address bar parsing), link checkers (validating URL format), sitemap generators, OAuth redirect URI validation.

Examples

```
//=====
// Pattern: URL Parser (complete)
//=====

#[derive(Debug, PartialEq)]
struct Url {
    scheme: String,
    authority: Option<String>,
    path: String,
    query: Option<String>,
    fragment: Option<String>,
}

#[derive(Debug)]
enum ParseState {
    Scheme,
    AfterScheme,
    Authority,
    Path,
    Query,
    Fragment,
}

struct UrlParser {
    input: Vec<char>,
    pos: usize,
    state: ParseState,
}

impl UrlParser {
    fn new(url: &str) -> Self {
        UrlParser {
            input: url.chars().collect(),
            pos: 0,
            state: ParseState::Scheme,
        }
    }

    fn parse(&mut self) -> Result<Url, String> {
        let mut scheme = String::new();
        let mut authority = None;
        let mut path = String::new();
```

```

let mut query = None;
let mut fragment = None;

while self.pos < self.input.len() {
    let ch = self.input[self.pos];

    match self.state {
        ParseState::Scheme => {
            if ch == ':' {
                if scheme.is_empty() {
                    return Err("Empty scheme".to_string());
                }
                self.state = ParseState::AfterScheme;
                self.pos += 1;
            } else if ch.is_alphanumeric() || ch == '+' || ch == '-' || ch == '.' {
                scheme.push(ch);
                self.pos += 1;
            } else {
                return Err(format!("Invalid scheme character: {}", ch));
            }
        }

        ParseState::AfterScheme => {
            if self.pos + 1 < self.input.len()
                && self.input[self.pos] == '/'
                && self.input[self.pos + 1] == '/'
            {
                self.state = ParseState::Authority;
                self.pos += 2;
            } else {
                self.state = ParseState::Path;
            }
        }

        ParseState::Authority => {
            if ch == '/' {
                self.state = ParseState::Path;
            } else if ch == '?' {
                self.state = ParseState::Query;
                self.pos += 1;
            } else if ch == '#' {
                self.state = ParseState::Fragment;
                self.pos += 1;
            } else {
                if authority.is_none() {
                    authority = Some(String::new());
                }
                authority.as_mut().unwrap().push(ch);
                self.pos += 1;
            }
        }

        ParseState::Path => {
    }
}

```

```

        if ch == '?' {
            self.state = ParseState::Query;
            self.pos += 1;
        } else if ch == '#' {
            self.state = ParseState::Fragment;
            self.pos += 1;
        } else {
            path.push(ch);
            self.pos += 1;
        }
    }

ParseState::Query => {
    if ch == '#' {
        self.state = ParseState::Fragment;
        self.pos += 1;
    } else {
        if query.is_none() {
            query = Some(String::new());
        }
        query.as_mut().unwrap().push(ch);
        self.pos += 1;
    }
}

ParseState::Fragment => {
    if fragment.is_none() {
        fragment = Some(String::new());
    }
    fragment.as_mut().unwrap().push(ch);
    self.pos += 1;
}

Ok(Url {
    scheme,
    authority,
    path,
    query,
    fragment,
})
}

fn main() {
    let urls = [
        "https://example.com/path/to/page?key=value#section",
        "http://user:pass@host:8080/path",
        "file:///home/user/file.txt",
        "mailto:user@example.com",
    ];
}

```

```

for url_str in &urls {
    let mut parser = UrlParser::new(url_str);
    match parser.parse() {
        Ok(url) => {
            println!("URL: {}", url_str);
            println!(" Scheme: {}", url.scheme);
            println!(" Authority: {:?}", url.authority);
            println!(" Path: {}", url.path);
            println!(" Query: {:?}", url.query);
            println!(" Fragment: {:?}", url.fragment);
        }
        Err(e) => println!("Parse error: {}", e),
    }
}

```

Key Patterns: - State transitions on delimiter characters - Lookahead for multi-character delimiters - Optional components with `Option<String>`

Pattern 9: Gap Buffer Implementation

Problem: Text editors need fast insertion/deletion at cursor. `Vec<char>` requires $O(N)$ time for insert—all characters after cursor shift.

Solution: Use gap buffer for single-cursor editors: maintain “gap” at cursor position, insertion fills gap $O(1)$, cursor movement slides gap $O(\text{distance})$. When gap full, grow buffer (double size).

Why It Matters: Gap buffer: typing is $O(1)$ amortized, but moving cursor across document is $O(N)$ worst case. Rope: all operations $O(\log N)$, handles GB files, multiple cursors efficiently, undo/redo via structural sharing (no copying).

Use Cases: Gap buffer for simple single-user editors with localized editing, command-line text input, undo buffers for small documents. Rope for modern editors (VS Code, Sublime), large log file viewers, collaborative editing (multiple users), version control diffs, syntax highlighting (parsing unchanged after local edit), mobile text editors (memory constrained).

Examples

```

//=====
// Gap Buffer (complete)
//=====

struct GapBuffer {
    buffer: Vec<char>,
    gap_start: usize,
    gap_end: usize,
}

impl GapBuffer {
    fn new() -> Self {
        GapBuffer::with_capacity(64)
    }
}

```

```
fn with_capacity(capacity: usize) -> Self {
    GapBuffer {
        buffer: vec!['\0'; capacity],
        gap_start: 0,
        gap_end: capacity,
    }
}

fn from_str(s: &str) -> Self {
    let chars: Vec<char> = s.chars().collect();
    let len = chars.len();
    let capacity = (len * 2).max(64);

    let mut buffer = vec!['\0'; capacity];
    buffer[..len].copy_from_slice(&chars);

    GapBuffer {
        buffer,
        gap_start: len,
        gap_end: capacity,
    }
}

// Insert character at cursor (gap_start)
fn insert(&mut self, ch: char) {
    if self.gap_start == self.gap_end {
        self.grow();
    }

    self.buffer[self.gap_start] = ch;
    self.gap_start += 1;
}

// Delete character before cursor
fn delete_backward(&mut self) -> Option<char> {
    if self.gap_start == 0 {
        return None;
    }

    self.gap_start -= 1;
    Some(self.buffer[self.gap_start])
}

// Delete character after cursor
fn delete_forward(&mut self) -> Option<char> {
    if self.gap_end == self.buffer.len() {
        return None;
    }

    let ch = self.buffer[self.gap_end];
    self.gap_end += 1;
    Some(ch)
```

```

}

// Move cursor left
fn move_left(&mut self) {
    if self.gap_start > 0 {
        self.gap_start -= 1;
        self.gap_end -= 1;
        self.buffer[self.gap_end] = self.buffer[self.gap_start];
    }
}

// Move cursor right
fn move_right(&mut self) {
    if self.gap_end < self.buffer.len() {
        self.buffer[self.gap_start] = self.buffer[self.gap_end];
        self.gap_start += 1;
        self.gap_end += 1;
    }
}

// Move cursor to position
fn move_to(&mut self, pos: usize) {
    let current_pos = self.gap_start;

    if pos < current_pos {
        for _ in 0..(current_pos - pos) {
            self.move_left();
        }
    } else if pos > current_pos {
        for _ in 0..(pos - current_pos) {
            self.move_right();
        }
    }
}

fn grow(&mut self) {
    let new_capacity = self.buffer.len() * 2;
    let additional = new_capacity - self.buffer.len();

    // Extend buffer
    self.buffer.resize(new_capacity, '\0');

    // Move content after gap to end
    let content_after_gap = self.buffer.len() - self.gap_end - additional;
    for i in (0..content_after_gap).rev() {
        self.buffer[new_capacity - 1 - i] = self.buffer[self.gap_end + i];
    }

    self.gap_end = new_capacity - content_after_gap;
}

fn to_string(&self) -> String {
    let mut result = String::new();

```

```

        for i in 0..self.gap_start {
            result.push(self.buffer[i]);
        }

        for i in self.gap_end..self.buffer.len() {
            result.push(self.buffer[i]);
        }

        result
    }

    fn len(&self) -> usize {
        self.gap_start + (self.buffer.len() - self.gap_end)
    }

    fn cursor_position(&self) -> usize {
        self.gap_start
    }
}

fn main() {
    let mut gb = GapBuffer::from_str("Hello World");

    println!("Initial: {}", gb.to_string());
    println!("Cursor at: {}", gb.cursor_position());

    // Move to position 5 (before "World")
    gb.move_to(6);
    gb.delete_backward(); // Delete space
    gb.insert(',');
    gb.insert(' ');

    println!("After edit: {}", gb.to_string());

    // Insert at beginning
    gb.move_to(0);
    gb.insert('>');
    gb.insert(' ');

    println!("Final: {}", gb.to_string());
}

```

Algorithm Explanation:

A gap buffer is a vector with three regions:

| |
|--------------------------------------|
| [prefix gap suffix] |
| ↑ ↑ ↑ ↑ |
| 0 gap_start gap_end buffer.len() |

- **Prefix:** Characters before cursor (`buffer[0..gap_start]`)

- **Gap:** Empty space (`buffer[gap_start..gap_end]`)
- **Suffix:** Characters after cursor (`buffer[gap_end..]`)

The cursor is conceptually at `gap_start`. Operations:

1. Insert at Cursor: O(1)

```
buffer[gap_start] = ch;
gap_start += 1; // Gap shrinks from left
```

2. Delete Backward: O(1)

```
gap_start -= 1; // Gap grows to the left
// Character at buffer[gap_start] is now in the gap (deleted)
```

3. Delete Forward: O(1)

```
gap_end += 1; // Gap grows to the right
// Character at buffer[gap_end-1] is now in the gap (deleted)
```

4. Move Cursor Left: O(1)

```
gap_start -= 1;
gap_end -= 1;
buffer[gap_end] = buffer[gap_start]; // Move char across gap
```

The gap “slides” left by moving one character from prefix to suffix.

5. Move Cursor Right: O(1)

```
buffer[gap_start] = buffer[gap_end]; // Move char across gap
gap_start += 1;
gap_end += 1;
```

6. Move to Position: O(distance) Repeatedly move left or right. For large jumps, this can be optimized by bulk copying.

7. Gap Full: When `gap_start == gap_end`, grow the buffer: - Allocate new buffer (typically double size) - Copy prefix to start - Copy suffix to end - Gap is now in the middle

Memory Layout Example:

Initial: “Hello” with gap size 4 at position 5:

```
[H] [e] [l] [l] [o] [_] [_] [_]
      ↑gap_start=5   ↑gap_end=9
```

After inserting “World”:

```
[H] [e] [l] [l] [o] [ ] [W] [o] [r]  
    ↑gap_start=9, gap_end=9 (gap full!)
```

Performance Characteristics: - **Insert/delete at cursor:** O(1) amortized - **Cursor movement:** O(distance moved) - **Random access:** O(N) worst case (if need to move gap) - **Memory:** O(N + gap_size)

Trade-offs: - **Pros:** Extremely fast for sequential editing (typing, deleting) - **Cons:** Multiple cursor positions require multiple gaps (use Rope instead) - **Cons:** Moving cursor far requires O(N) operations

Gap buffers excel when edits are localized around a single cursor—exactly the pattern of human typing!

Key Concepts: - Gap buffer provides O(1) insertion/deletion at cursor position - Gap grows when full (exponential reallocation) - Cursor movement slides gap by copying characters across it: O(distance) - Efficient for localized, sequential edits (the common case in text editing) - Simple implementation compared to more complex structures like ropes

Rope Data Structure

Problem: Gap buffers struggle with: - Large documents (multi-megabyte files cause O(N) cursor movement) - Multiple cursors (each needs its own gap) - Undo/redo (copying entire buffer is expensive)

Solution: Rope—a binary tree where leaves contain string fragments. Operations split/concatenate tree nodes, achieving O(log N) insertion/deletion anywhere in the document.

Why This Matters: For documents > 1MB or collaborative editing with multiple cursors, ropes outperform gap buffers dramatically. The tree structure enables structural sharing for efficient undo/redo without copying the entire document.

Use Case: Modern text editors (Xi, Visual Studio Code internals), large log file viewers, collaborative editing systems, version control systems.

```
//=====  
// Rope (complete)  
//=====  
#[derive(Clone)]  
enum Rope {  
    Leaf(String),  
    Branch {  
        left: Box<Rope>,  
        right: Box<Rope>,  
        length: usize, // Total length of left subtree  
    },  
}  
  
impl Rope {  
    fn from_str(s: &str) -> Self {  
        Rope::Leaf(s.to_string())  
    }  
  
    fn concat(left: Rope, right: Rope) -> Self {
```

```

let length = left.len();
Rope::Branch {
    left: Box::new(left),
    right: Box::new(right),
    length,
}
}

fn len(&self) -> usize {
    match self {
        Rope::Leaf(s) => s.len(),
        Rope::Branch { length, right, .. } => length + right.len(),
    }
}

// Insert string at position
fn insert(&mut self, pos: usize, text: &str) {
    let (left, right) = self.split(pos);
    *self = Rope::concat(Rope::concat(left, Rope::from_str(text)), right);
}

// Delete range
fn delete(&mut self, start: usize, end: usize) {
    let (left, rest) = self.split(start);
    let (_, right) = rest.split(end - start);
    *self = Rope::concat(left, right);
}

// Split rope at position
fn split(self, pos: usize) -> (Rope, Rope) {
    match self {
        Rope::Leaf(s) => {
            if pos >= s.len() {
                (Rope::Leaf(s), Rope::Leaf(String::new()))
            } else if pos == 0 {
                (Rope::Leaf(String::new()), Rope::Leaf(s))
            } else {
                let (left, right) = s.split_at(pos);
                (Rope::Leaf(left.to_string()), Rope::Leaf(right.to_string()))
            }
        }
        Rope::Branch { left, right, length } => {
            if pos < length {
                let (ll, lr) = left.split(pos);
                (ll, Rope::concat(lr, *right))
            } else if pos == length {
                (*left, *right)
            } else {
                let (rl, rr) = right.split(pos - length);
                (Rope::concat(*left, rl), rr)
            }
        }
    }
}

```

```

}

// Get character at position
fn char_at(&self, pos: usize) -> Option<char> {
    match self {
        Rope::Leaf(s) => s.chars().nth(pos),
        Rope::Branch { left, right, length } => {
            if pos < *length {
                left.char_at(pos)
            } else {
                right.char_at(pos - length)
            }
        }
    }
}

// Convert to string
fn to_string(&self) -> String {
    match self {
        Rope::Leaf(s) => s.clone(),
        Rope::Branch { left, right, .. } => {
            format!("{}{}", left.to_string(), right.to_string())
        }
    }
}

// Rebalance tree if needed
fn rebalance(self) -> Self {
    // Simplified rebalancing: collect all leaves and rebuild
    let text = self.to_string();
    if text.len() < 100 {
        return Rope::Leaf(text);
    }

    let mid = text.len() / 2;
    let (left, right) = text.split_at(mid);
    Rope::concat(Rope::Leaf(left.to_string()), Rope::Leaf(right.to_string()))
}
}

fn main() {
    let mut rope = Rope::from_str("Hello World");
    println!("Initial: {}", rope.to_string());

    // Insert at position 5
    rope.insert(5, ", Beautiful");
    println!("After insert: {}", rope.to_string());

    // Delete range
    rope.delete(5, 16); // Remove ", Beautiful"
    println!("After delete: {}", rope.to_string());

    // Character access
}

```

```

if let Some(ch) = rope.char_at(0) {
    println!("First char: {}", ch);
}

println!("Length: {}", rope.len());
}

```

Algorithm Explanation:

A rope is a binary tree where: - **Leaf nodes**: Contain actual string data - **Branch nodes**: Have left/right children and store the total length of the left subtree

Example rope for "Hello, World!":

```

Branch(6)
/
Leaf("Hello") Leaf(", World!")

```

The number in **Branch(6)** is the length of the left subtree (6 characters in "Hello").

Core Operations:

1. Indexing (`char_at(pos)`): O(log N)

```

if pos < left.len() {
    search left subtree
} else {
    search right subtree at (pos - left.len())
}

```

Navigate down the tree by comparing position with left subtree length. This is O(log N) with balanced trees (O(height) in general).

2. Concatenation (`concat(rope1, rope2)`): O(1)

```

Branch {
    left: rope1,
    right: rope2,
    length: rope1.len()
}

```

Simply create a new branch node—no copying of string data! This is why ropes excel: combining two documents is instant.

3. Splitting (`split(pos)`): O(log N)

Splitting at position `pos` creates two ropes. Navigate to the leaf containing `pos`: - Split that leaf into two strings - Reassemble ancestors, putting left parts in one rope and right parts in another

Example: Splitting "Hello, World!" at position 7:

```
Original:  
    Branch(6)  
    /      \  
Leaf("Hello") Leaf(", World!")
```

Split at 7 (after "Hello, "):

```
Left rope:  
    Branch(6)  
    /      \  
Leaf("Hello") Leaf(", ")
```

Right rope:

```
Leaf("World!")
```

4. Insertion (`insert(pos, text)`): O(log N)

```
let (left, right) = rope.split(pos);  
rope = concat(concat(left, from_str(text)), right);
```

This is the key insight: insertion is just split + concatenate! We create new tree nodes but reuse existing string leaves. The old tree still exists (structural sharing for undo).

5. Deletion (`delete(start, end)`): O(log N)

```
let (left, rest) = rope.split(start);  
let (_, right) = rest.split(end - start);  
rope = concat(left, right);
```

Tree Rebalancing:

Without rebalancing, a rope can degenerate into a linked list (height O(N)). Rebalancing maintains O(log N) height. Simple strategy: - If leaf gets too large (>1KB), split it - If tree gets too deep, rebuild by collecting all text and recreating a balanced tree - Production implementations use more sophisticated techniques (red-black trees, weight-balanced trees)

Structural Sharing for Undo:

Old rope versions can coexist with new versions:

```
let v1 = rope.clone();           // Cheap: just Arc/Rc clone  
rope.insert(100, "text");       // Creates new nodes, reuses old leaves  
// v1 still valid, shares leaves with new rope
```

Undo is just reverting to the old rope reference—no need to “unapply” operations.

Performance Characteristics: - **Insert/Delete:** O(log N) anywhere in document - **Index access:** O(log N) - **Concatenation:** O(1) - **Split:** O(log N) - **Iteration:** O(N) but with good cache locality if leaves are large - **Memory:** O(N) for text + O(N/leaf_size) for tree nodes

Trade-offs:

| Operation | Gap Buffer | Rope |
|---------------------------|-------------|-------------------------|
| Insert at cursor | O(1) | O(log N) |
| Insert random position | O(N) | O(log N) |
| Cursor movement | O(distance) | N/A (no cursor) |
| Large files | Poor | Excellent |
| Multiple cursors | Poor | Excellent |
| Undo/redo | O(N) copy | O(1) structural sharing |
| Memory overhead | Low | Moderate |
| Implementation complexity | Simple | Complex |

When to Use: - **Gap Buffer:** Small documents, single cursor, simple implementation - **Rope:** Large documents, multiple cursors, undo/redo, collaborative editing

Key Concepts: - Rope is a binary tree of string fragments (leaves) and metadata (branches) - All operations are O(log N) by navigating the tree - Concatenation is O(1)—just create a branch node - Structural sharing enables O(1) undo/redo without copying - Rebalancing maintains O(log N) height for guaranteed performance - Trades constant-time cursor insertion for logarithmic-time random access

Pattern 10: Knuth-Morris-Pratt (KMP) String Search

Problem: Naive string search is O(NM) where N = text length, M = pattern length. Searching 1GB log file for 100-byte pattern = $10^9 * 100$ comparisons worst case.

Solution: Use KMP (Knuth-Morris-Pratt) for guaranteed O(N+M) with no text backtracking via preprocessed “failure function”. Use Boyer-Moore for practical O(N/M) best case by scanning pattern right-to-left and skipping sections.

Why It Matters: KMP guarantees linear time—1GB file with 1KB pattern: naive = 10^{12} ops worst case, KMP = 10^9 ops always. Boyer-Moore often 3-5x faster than KMP in practice (especially long patterns, large alphabets)—grep and text editors use it.

Use Cases: Text search in editors (Boyer-Moore for interactive search), genomic analysis (KMP for ATCG sequences), log filtering (pattern matching millions of lines), compiler lexical analysis (token recognition), intrusion detection (packet payload scanning), plagiarism detection (document comparison), virus scanning.

Examples

```
//=====
// Pattern: Knuth-Morris-Pratt (KMP) String Search (complete)
//=====

struct KmpMatcher {
    pattern: Vec<char>,
    failure: Vec<usize>,
}

impl KmpMatcher {
```

```
fn new(pattern: &str) -> Self {
    let pattern: Vec<char> = pattern.chars().collect();
    let failure = Self::compute_failure(&pattern);

    KmpMatcher { pattern, failure }
}

fn compute_failure(pattern: &[char]) -> Vec<usize> {
    let mut failure = vec![0; pattern.len()];
    let mut j = 0;

    for i in 1..pattern.len() {
        while j > 0 && pattern[i] != pattern[j] {
            j = failure[j - 1];
        }

        if pattern[i] == pattern[j] {
            j += 1;
        }

        failure[i] = j;
    }

    failure
}

fn find_all(&self, text: &str) -> Vec<usize> {
    let text: Vec<char> = text.chars().collect();
    let mut matches = Vec::new();
    let mut j = 0;

    for (i, &ch) in text.iter().enumerate() {
        while j > 0 && ch != self.pattern[j] {
            j = self.failure[j - 1];
        }

        if ch == self.pattern[j] {
            j += 1;
        }

        if j == self.pattern.len() {
            matches.push(i + 1 - j);
            j = self.failure[j - 1];
        }
    }

    matches
}

fn contains(&self, text: &str) -> bool {
    !self.find_all(text).is_empty()
}
}
```

```

fn main() {
    let matcher = KmpMatcher::new("ABABC");
    let text = "ABABDABACDABABCABAB";

    let matches = matcher.find_all(text);
    println!("Pattern found at positions: {:?}", matches);

    for pos in matches {
        println!(" Position {}: {}", pos, &text[pos..pos + 5]);
    }
}

```

Algorithm Explanation:

KMP's brilliance lies in its **failure function** (also called “partial match table” or “prefix function”). This preprocessed array tells us: “If we’ve matched j characters and then mismatch, how many characters from the pattern’s start also appear at the current position?”

Failure Function Construction (`compute_failure`):

For pattern “ABABC”:

| | |
|----------|-----------|
| Pattern: | A B A B C |
| Index: | 0 1 2 3 4 |
| Failure: | 0 0 1 2 0 |

Explanation: - `failure[0] = 0`: No proper prefix/suffix for single character - `failure[1] = 0`: “AB” has no matching prefix/suffix - `failure[2] = 1`: “ABA” has “A” as both prefix and suffix (length 1) - `failure[3] = 2`: “ABAB” has “AB” as both prefix and suffix (length 2) - `failure[4] = 0`: “ABABC” has no matching prefix/suffix

The algorithm builds this by: 1. Comparing `pattern[i]` with `pattern[j]` where j represents the length of the current matching prefix 2. If they match, increment j (extending the match) 3. If they don’t match and $j > 0$, set $j = \text{failure}[j-1]$ (try a shorter prefix) 4. Store the result: `failure[i] = j`

Search Algorithm (`find_all`):

When searching, we maintain j = number of pattern characters matched. On a mismatch: - **Naive approach**: Reset j to 0 and continue - **KMP approach**: Set $j = \text{failure}[j-1]$, utilizing overlap information

Example: Searching for “ABABC” in “ABABDABABC”:

| | |
|----------|--------------------------|
| Text: | A B A B D A B A B C |
| Pattern: | A B A B C |
| | ↑ mismatch at position 4 |

Instead of restarting from position 0, we know:
- We matched “ABAB” (4 chars)
- `failure[3] = 2`, meaning “AB” at start matches “AB” at position 2

- Resume matching from pattern[2], not pattern[0]

This eliminates redundant comparisons!

Complexity Analysis: - **Preprocessing:** O(M) to build failure function - **Search:** O(N) for text scan (never backtracks!) - **Total:** O(N + M) - **Space:** O(M) for failure array

The key insight: each text character is examined at most once. The *j* variable may decrease (via failure function), but the text index *i* only advances forward.

Key Concepts: - Failure function encodes prefix/suffix overlap information - No backtracking in text—each character examined once - O(N + M) time complexity (optimal for string matching) - Works well when pattern has internal repetition

Pattern 12: Boyer-Moore String Search

Problem: KMP scans every text character left-to-right, no skipping possible. Can we do better?

Solution: Scan pattern right-to-left (most discriminating character first). Build “bad character table” mapping each character to its rightmost position in pattern.

Why This Matters: Best case O(N/M)—searching for “PATTERN” (7 chars) in text where “N” never appears skips 7 positions per comparison, examining only N/7 characters! Average case much faster than KMP for long patterns and large alphabets. English text search: Boyer-Moore 3-5x faster than KMP. DNA search (4-letter alphabet): KMP competitive. This is why grep, Vim, and text editors use Boyer-Moore variants.

Use Cases: Interactive text search in editors (fast visual feedback), grep/ripgrep tools (searching codebases), plagiarism detection (comparing documents), bioinformatics (protein sequence search has 20-letter alphabet), large document search (legal discovery, log analysis), web page search.

Examples

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

//=====
// Boyer-Moore string search algorithm
//=====

struct BoyerMoore {
    pattern: Vec<char>,
    bad_char: HashMap<char, usize>,
}

impl BoyerMoore {
    fn new(pattern: &str) -> Self {
        let pattern: Vec<char> = pattern.chars().collect();
        let bad_char = Self::build_bad_char_table(&pattern);

        BoyerMoore { pattern, bad_char }
    }

    fn search(text: &str) -> Option {
        let mut j = 0;
        let mut i = 0;
        let mut n = self.pattern.len();

        while i < text.len() {
            if self.pattern[j] == text[i] {
                j += 1;
                i += 1;
                if j == n {
                    return Some(i);
                }
            } else {
                j = self.bad_char.get(&text[i]).unwrap_or(&self.pattern.len());
                i += 1;
            }
        }

        None
    }
}
```

```

fn build_bad_char_table(pattern: &[char]) -> HashMap<char, usize> {
    let mut table = HashMap::new();

    for (i, &ch) in pattern.iter().enumerate() {
        table.insert(ch, i);
    }

    table
}

fn find_all(&self, text: &str) -> Vec<usize> {
    let text: Vec<char> = text.chars().collect();
    let mut matches = Vec::new();
    let m = self.pattern.len();
    let n = text.len();

    if m > n {
        return matches;
    }

    let mut s = 0; // Shift of pattern relative to text

    while s <= n - m {
        let mut j = m;

        // Scan from right to left
        while j > 0 && self.pattern[j - 1] == text[s + j - 1] {
            j -= 1;
        }

        if j == 0 {
            // Match found
            matches.push(s);

            // Shift pattern
            if s + m < n {
                let next_char = text[s + m];
                s += m - self.bad_char.get(&next_char).unwrap_or(&0);
            } else {
                s += 1;
            }
        } else {
            // Mismatch: use bad character rule
            let bad_char = text[s + j - 1];
            let shift = if let Some(&pos) = self.bad_char.get(&bad_char) {
                if pos < j - 1 {
                    j - 1 - pos
                } else {
                    1
                }
            } else {
                j
            }
        }
    }
}

```

```

        };
        s += shift;
    }
}

matches
}
}

fn main() {
    let bm = BoyerMoore::new("EXAMPLE");
    let text = "THIS IS A SIMPLE EXAMPLE FOR EXAMPLE MATCHING";

    let matches = bm.find_all(text);
    println!("Matches at: {:?}", matches);

    for pos in matches {
        println!("  {}", &text[pos..pos + 7]);
    }
}

```

Algorithm Explanation:

Boyer-Moore's key insight: **scan the pattern right-to-left**. When you find a mismatch, you often have information that allows skipping multiple positions in the text.

Bad Character Heuristic:

The “bad character rule” says: when we mismatch at text character **c**, we can shift the pattern to align the rightmost occurrence of **c** in the pattern with the mismatched text position.

Example: Searching for “EXAMPLE” in text “...SIMPLE...”:

```

Text: S I M P L E
Pattern: E X A M P L E
          ↑ mismatch

```

The mismatched character is 'S'. Looking at pattern "EXAMPLE":

- 'S' doesn't appear in the pattern at all
- We can shift the entire pattern past 'S'
- This skips 7 positions in one step!

If 'S' appeared in the pattern, we'd align it:

```

Text: S I M P L E
Pattern: E X A M P L E (shifted to align any 'S')

```

Bad Character Table Construction:

For each character in the pattern, store its rightmost position:

Pattern: EXAMPLE

Table: E→6, L→5, P→4, M→3, A→2, X→1

When we mismatch on character **c** at pattern position **j**: - If **c** is in the table at position **pos** and **pos < j**: shift **j - pos** positions - If **c** is not in the table: shift **j** positions (entire pattern) - If **pos ≥ j**: shift 1 position (to avoid negative shift)

Right-to-Left Scanning:

Scanning right-to-left is crucial. If the last character of the pattern doesn't match, we immediately know the pattern can't match at this position. The rightmost character is the most "discriminating"—if it's rare in the text, we skip many positions.

Complexity Analysis: - **Best case:** O(N/M) — when last pattern character never matches, we skip M positions each time - **Average case:** O(N) — significantly faster than KMP in practice - **Worst case:** O(NM) — pathological patterns like "AAA...A" in text "AAA...A" - **Preprocessing:** O(M + alphabet_size)

The full Boyer-Moore includes both "bad character" and "good suffix" heuristics. Our implementation uses bad character only, which is simpler but still very effective.

Why It's Fast in Practice: - Long patterns benefit more (larger skips) - Large alphabets (English: 26 letters) have better discrimination - Real text isn't pathological—mismatches are common and allow big skips

Key Concepts: - Right-to-left scanning maximizes information from mismatches - Bad character rule enables large forward jumps - Sublinear average-case performance (O(N/M) best case) - Optimal for long patterns and large alphabets - Trade-off: slightly more complex than KMP, but faster in practice

Pattern 12: String Interning

Problem: Programs with repeated strings waste memory—a web server with 10K sessions storing "logged_in" user state = 10K copies of same string. Compilers store identifiers thousands of times.

Solution: Implement string interning pool using **HashMap<Arc<str>, Arc<str>>** to deduplicate. When interning a string: check if already in pool, return existing **Arc** (cheap clone); if not in pool, insert new **Arc** and return it.

Why It Matters: Memory: 1M instances of "error" = 5MB as separate strings, 5 bytes + overhead as interned. String comparison: O(N) for string equality, O(1) for Arc pointer equality.

Use Cases: Compiler symbol tables (variable names, type names repeated in AST), configuration systems (keys like "database.host" repeated), logging (level strings "ERROR"/"INFO" everywhere), web frameworks (route paths, session keys), game engines (asset tags, entity names), network protocols (header names, status codes), i18n (translation keys).

Examples

```
use std::collections::HashMap;
use std::sync::Arc;
```

```
//=====
// String Interning (complete)
//=====

#[derive(Debug, Clone, PartialEq, Eq, Hash)]
struct InternedString(Arc<str>);

impl InternedString {
    fn as_str(&self) -> &str {
        &self.0
    }
}

struct StringInterner {
    map: HashMap<Arc<str>, InternedString>,
}

impl StringInterner {
    fn new() -> Self {
        StringInterner {
            map: HashMap::new(),
        }
    }

    fn intern(&mut self, s: &str) -> InternedString {
        if let Some(interned) = self.map.get(s) {
            return interned.clone();
        }

        let arc: Arc<str> = Arc::from(s);
        let interned = InternedString(arc.clone());
        self.map.insert(arc, interned.clone());
        interned
    }

    fn len(&self) -> usize {
        self.map.len()
    }

    fn memory_usage(&self) -> usize {
        self.map.iter()
            .map(|(k, _)| k.len())
            .sum()
    }
}

fn main() {
    let mut interner = StringInterner::new();

    let s1 = interner.intern("hello");
    let s2 = interner.intern("world");
    let s3 = interner.intern("hello"); // Returns same Arc
```

```

    println!("s1 == s3: {}", s1 == s3); // true
    println!("Unique strings: {}", interner.len());

    // Demonstrate memory savings
    let tags = vec!["rust", "programming", "rust", "tutorial", "rust"];
    let interned_tags: Vec<_> = tags.iter()
        .map(|&s| interner.intern(s))
        .collect();

    println!("Tags: {} unique", interner.len());
    println!("Memory: {} bytes", interner.memory_usage());
}


```

String Types

| Type | Owned | UTF-8 Use Case |
|-----------------------------|-----------|------------------------------|
| <code>String</code> | Yes | Dynamic, growable strings |
| <code>&str</code> | No | String slices, literals |
| <code>Cow<str></code> | Sometimes | Clone-on-write optimization |
| <code>OsString</code> | Yes | Platform strings, file paths |
| <code>Path</code> | No | File system paths |

Performance Tips

1. **Pre-allocate:** Use `String::with_capacity()` when size known
2. **Avoid cloning:** Use `&str` parameters instead of `String`
3. **Use Cow:** For conditional modifications
4. **Intern strings:** For repeated strings
5. **Chars not bytes:** Use `char_indices()` for UTF-8 safety

Summary

1. **String Type Selection:** String (owned), &str (borrowed), Cow (conditional), OsString (platform), Path (filesystem)
2. **Builder Patterns:** Pre-allocate capacity, method chaining, domain-specific builders for HTML/SQL
3. **Zero-Copy Operations:** Iterator methods returning &str slices, Cow for conditional allocation
4. **UTF-8 Handling:** Validation, character boundaries, grapheme clusters for display
5. **Text Editor Data Structures:** Gap buffers O(1) cursor insert, ropes O(log N) everywhere
6. **Pattern Matching:** KMP O(N+M) guaranteed, Boyer-Moore O(N/M) best case
7. **String Interning:** Arc-based deduplication, O(1) comparison, memory savings

Key Takeaways: - Type choice determines allocation: &str for parameters, String for owned, Cow for conditional - Pre-allocation is O(N) vs O(N) amortized with log(N) reallocations - Zero-copy parsing: 10MB file = 10MB memory (not 20MB with owned strings) - UTF-8 has three levels: bytes < characters < graphemes (use appropriate level) - Gap buffer for simple editors, rope for production features (multi-cursor, undo, large files) - KMP for guaranteed linear time, Boyer-Moore 3-5x faster in practice - Intern repeated strings: 1M copies of "error" = 5 bytes + overhead vs 5MB

Performance Guidelines: - String building: pre-allocate capacity when size known or estimable - Parsing: use zero-copy iterators (split, lines), collect only when necessary - UTF-8: use `char_indices()` for boundaries, graphemes for display, bytes for protocols - Text editing: gap buffer < 1MB, rope > 1MB or multiple cursors - Search: Boyer-Moore for interactive (large alphabet), KMP for guaranteed (small alphabet) - Comparison: intern if comparing same strings repeatedly

Safety Notes: - Never index strings with byte positions without checking char boundaries - Validate UTF-8 from external sources (`from_utf8`, `from_utf8_lossy`) - Use grapheme clusters for user-facing operations (truncation, reversal, width) - Cow scans input twice (check + build) but faster than always allocating - Arc clones are cheap (refcount increment) but not free—benchmark if critical

HashMap & HashSet Patterns

`HashMap<K, V>` is Rust's primary hash table implementation, offering excellent average-case performance for lookups, insertions, and deletions. This chapter explores key patterns for using `HashMap` and `HashSet` effectively, from basic operations to advanced techniques for performance, memory optimization, and concurrency.

Pattern 1: The Entry API

The Entry API is the most idiomatic and efficient way to handle complex conditional logic in `HashMap`, such as "insert if absent, update if present."

- **Problem:** - **Inefficiency:** A common but inefficient pattern is to first check if a key exists (`contains_key`), and then perform a separate operation to insert or update the value. This results in at least two separate hash lookups.
- **Solution:** - **The Entry API:** Use `map.entry(key)`, which performs a single lookup and returns an `Entry` enum. This enum represents the state of the key's slot in the map.
- **Why It Matters:** - **Performance:** The Entry API reduces hash lookups from two or more down to just one, which can double the performance of lookup-heavy operations like word counting or building aggregations. - **Readability:** It produces much cleaner, more idiomatic Rust code that clearly expresses the intended logic.
- **Use Cases:**
 - **Frequency Counting:** Counting occurrences of words, characters, or any other item.
 - **Grouping and Aggregation:** Grouping a list of items by a key and calculating sums, counts, or averages for each group.
 - **Cache Implementation:** Efficiently managing entries in a cache, like an LRU cache.
 - **Building Adjacency Lists:** Constructing graph data structures by adding edges between nodes.
 - **Default Value Initialization:** Ensuring a key has a default value before modifying it.

Example: Word Frequency Counter

The classic use case for the Entry API is counting word frequencies. The `.or_insert(0)` method gets the current count for a word or inserts `0` if the word is new. We can then increment the count in place.

```

use std::collections::HashMap;

fn word_frequency_counter() {
    let text = "the quick brown fox jumps over the lazy dog";
    let mut counts: HashMap<String, usize> = HashMap::new();

    for word in text.split_whitespace() {
        // Get the entry for the word, insert 0 if it's vacant,
        // and then get a mutable reference to the value to increment it.
        *counts.entry(word.to_string()).or_insert(0) += 1;
    }

    println!("Word counts: {:?}", counts);
    // Top word is "the" with a count of 2.
    assert_eq!(counts.get("the"), Some(&2));
}

```

Example: Grouping Items by Key

The Entry API is perfect for grouping items from a list into a `HashMap` where keys are a property of the item and values are a `Vec` of items sharing that property. `.or_insert_with(Vec::new)` is used here to lazily create a new vector only when a new key is encountered.

```

use std::collections::HashMap;

#[derive(Debug)]
struct Sale {
    category: String,
    amount: f64,
}

fn group_sales_by_category() {
    let sales = vec![
        Sale { category: "Electronics".to_string(), amount: 1200.0 },
        Sale { category: "Furniture".to_string(), amount: 450.0 },
        Sale { category: "Electronics".to_string(), amount: 25.0 },
    ];

    let mut sales_by_category: HashMap<String, Vec<Sale>> = HashMap::new();

    for sale in sales {
        // Find the vector for the category, creating a new one if it doesn't exist,
        // and then push the sale into it.
        sales_by_category
            .entry(sale.category.clone())
            .or_insert_with(Vec::new)
            .push(sale);
    }

    println!("Sales grouped by category: {:?}", sales_by_category);
}

```

```
    assert_eq!(sales_by_category.get("Electronics").unwrap().len(), 2);
}
```

Example: Implementing an LRU Cache

The Entry API can be used to implement more complex data structures like a Least Recently Used (LRU) cache. Here, we use `Entry::Occupied` and `Entry::Vacant` to handle the logic for existing and new cache entries separately.

```
use std::collections::{HashMap, VecDeque};
use std::hash::Hash;

struct LruCache<K, V> {
    capacity: usize,
    map: HashMap<K, V>,
    order: VecDeque<K>, // Tracks usage order, from least to most recent.
}

impl<K: Eq + Hash + Clone> LruCache<K, V> {
    fn new(capacity: usize) -> Self {
        Self { capacity, map: HashMap::new(), order: VecDeque::new() }
    }

    fn put(&mut self, key: K, value: V) {
        use std::collections::hash_map::Entry;

        match self.map.entry(key.clone()) {
            Entry::Occupied(mut entry) => {
                // Key already exists, update the value.
                entry.insert(value);
                // Move it to the front of the usage queue.
                self.order.retain(|k| k != &key);
                self.order.push_back(key);
            }
            Entry::Vacant(entry) => {
                // Key is new. First, check if we need to evict an old entry.
                if self.map.len() >= self.capacity {
                    if let Some(lru_key) = self.order.pop_front() {
                        self.map.remove(&lru_key);
                    }
                }
                // Insert the new value and add it to the front of the usage queue.
                entry.insert(value);
                self.order.push_back(key);
            }
        }
    }
}
```

Pattern 2: Custom Hashing and Equality

By default, `HashMap` uses a secure but slower hashing algorithm (SipHash) and relies on the standard `Eq` and `Hash` traits. For many use cases, providing a custom implementation is necessary for correctness or performance.

- **Problem:** - **Semantic Equality:** The default `Hash` and `PartialEq` derived for a type may not match the desired semantic equality. For example, you might want string keys to be case-insensitive, or floating-point keys to have a tolerance.
 - **Solution:** - **Implement Hash and PartialEq:** Manually implement the `Hash` and `PartialEq` traits for your key type. It is a critical invariant that if `a == b`, then `hash(a) == hash(b)`.
 - **Why It Matters:** - **Correctness:** Custom equality and hashing are essential for creating maps that behave correctly according to your application's domain logic. An incorrect `Hash` implementation can lead to keys being lost or not found in the map.
- **Use Cases:**
- **Case-Insensitive Keys:** For usernames, HTTP headers, or configuration keys.
 - **Performance-Critical Maps:** In compilers (symbol tables), game engines (entity IDs), or any hot path that heavily uses a map with trusted integer keys.
 - **Composite Keys:** Using a struct with multiple fields as a single key.
 - **Floating-Point Keys:** For spatial indexing or scientific computing, typically by wrapping floats to handle `Nan` and rounding for approximate equality.

Example: Case-Insensitive String Keys

To make a `HashMap` treat string keys as case-insensitive, we can create a newtype wrapper around `String`. We then implement `PartialEq` to compare strings case-insensitively and `Hash` to hash their lowercase versions.

```
use std::collections::HashMap;
use std::hash::{Hash, Hasher};

#[derive(Debug, Eq)]
struct CaseInsensitiveString(String);

impl Hash for CaseInsensitiveString {
    fn hash<H: Hasher>(&self, state: &mut H) {
        // Hash the lowercase version of the string to ensure "A" and "a" have the same hash.
        for byte in self.0.bytes().map(|b| b.to_ascii_lowercase()) {
            byte.hash(state);
        }
    }
}

impl PartialEq for CaseInsensitiveString {
    fn eq(&self, other: &Self) -> bool {
        // Compare the strings case-insensitively.
        self.0.eq_ignore_ascii_case(&other.0)
    }
}
```

```

    }

}

fn case_insensitive_headers() {
    let mut headers = HashMap::new();

    headers.insert(CaseInsensitiveString("Content-Type".to_string()), "application/json");
    headers.insert(CaseInsensitiveString("X-Request-ID".to_string()), "12345");

    // Lookup is case-insensitive.
    let key = CaseInsensitiveString("content-type".to_string());
    assert_eq!(headers.get(&key), Some(&"application/json"));
}

}

```

Example: Faster Hashing with FxHashMap

For performance-critical code paths where the keys are trusted (not controlled by a potential attacker), you can replace the standard `HashMap` with `FxHashMap` from the `rustc-hash` crate. It uses a much faster, non-cryptographic hash function.

```

// Add `rustc-hash = "1.1"` to Cargo.toml
use rustc_hash::FxHashMap;
use std::collections::HashMap;
use std::time::Instant;

fn benchmark_fxhashmap() {
    const SIZE: usize = 1_000_000;

    // Benchmark FxHashMap
    let start = Instant::now();
    let mut fx_map = FxHashMap::default();
    for i in 0..SIZE {
        fx_map.insert(i, i);
    }
    println!("FxHashMap insertion time: {:?}", start.elapsed());

    // Benchmark standard HashMap
    let start = Instant::now();
    let mut std_map = std::collections::HashMap::new();
    for i in 0..SIZE {
        std_map.insert(i, i);
    }
    println!("Standard HashMap insertion time: {:?}", start.elapsed());
}

```

Pattern 3: Capacity and Memory Management

Failing to manage `HashMap` capacity can lead to poor performance due to repeated resizing, or wasted memory from over-allocation.

- **Problem:** - **Latency Spikes:** When a `HashMap` reaches its capacity, it must resize, which involves allocating a new, larger backing array and re-hashing and moving every single element. This can cause significant latency spikes in performance-sensitive applications.
- **Solution:** - **Pre-allocation:** If you know the final size of the map, use `HashMap::with_capacity(n)` to pre-allocate the required memory upfront, avoiding all intermediate resizes. - **reserve:** Before inserting a batch of items, use `map.reserve(additional_items)` to ensure there is enough capacity, preventing a resize during the insertion loop.
- **Why It Matters:** - **Performance:** Pre-allocating capacity can make the construction of large `HashMaps` 3-10x faster. A map of 1 million entries could trigger ~20 resize operations if not pre-allocated.
- **Use Cases:**
 - **Batch Data Loading:** When loading a large dataset from a file or database, pre-allocate the map with the size of the dataset.
 - **Configuration Maps:** A global configuration map that is loaded at startup and then read frequently should be shrunk to fit after loading.
 - **High-Frequency Trading / Real-Time Systems:** Any system where latency spikes are unacceptable must carefully manage capacity to avoid resizes.
 - **Memory-Constrained Environments:** Any application where memory usage is a primary concern.

Example: Pre-allocating for Batch Processing

When you know roughly how many items you're going to insert, using `HashMap::with_capacity` can dramatically speed up insertion by avoiding repeated resizing and re-hashing.

```
use std::collections::HashMap;
use std::time::Instant;

fn batch_processing_with_capacity() {
    const BATCH_SIZE: usize = 500_000;

    // Without pre-allocation
    let start = Instant::now();
    let mut map1 = HashMap::new();
    for i in 0..BATCH_SIZE {
        map1.insert(i, i);
    }
    println!("Without pre-allocation: {:?}", start.elapsed(),
            map1.capacity());

    // With pre-allocation
    let start = Instant::now();
    let mut map2 = HashMap::with_capacity(BATCH_SIZE);
    for i in 0..BATCH_SIZE {
        map2.insert(i, i);
    }
    println!("With pre-allocation: {:?}", start.elapsed(),
            map2.capacity());
}
```

```

let start = Instant::now();
let mut map2 = HashMap::with_capacity(BATCH_SIZE);
for i in 0..BATCH_SIZE {
    map2.insert(i, i);
}
println!("With pre-allocation: {:?}", final capacity: {}, start.elapsed(),
        map2.capacity());
}

```

Example: Shrinking to Reclaim Memory

For a long-lived `HashMap` that is populated once and then mostly read from, you can call `shrink_to_fit()` after population to release any excess memory capacity.

```

use std::collections::HashMap;

fn shrinking_to_fit() {
    let mut map = HashMap::with_capacity(1000);
    println!("Initial capacity: {}", map.capacity());

    for i in 0..100 {
        map.insert(i, i);
    }
    println!("Capacity after 100 insertions: {}", map.capacity());

    // Shrink the map to reclaim the unused capacity.
    map.shrink_to_fit();
    println!("Capacity after shrinking: {}", map.capacity());
}

```

Pattern 4: Alternative Map Implementations

`HashMap` is a great default, but the Rust ecosystem offers several other map implementations that are better suited for specific use cases.

- **Problem:** - **Order:** `HashMap` does not preserve insertion order, and its iteration order is effectively random. This is problematic if you need deterministic output or want to iterate over items in the order they were added.
- **Solution:** - **BTreeMap:** A map based on a B-Tree. It keeps its keys sorted at all times.
- **Why It Matters:** - **Choosing the Right Tool:** Using the right map for the job can lead to simpler code and better performance. Using a `BTreeMap` for range queries can turn an O(N) scan into an efficient O(log N) operation.
- **Use Cases:**
 - **BTreeMap:**
 - **Leaderboards / Rankings:** Keeping scores sorted.
 - **Time-Series Data:** Querying data within a specific time range.

- **Deterministic Serialization:** Ensuring keys in a serialized format like JSON are always in the same order.
- **IndexMap:**
 - **Ordered Configuration:** Preserving the order of keys from a config file.
 - **LRU Caches:** **IndexMap** is often a great choice for LRU caches as it naturally tracks insertion order.
 - **Remembering User Choices:** Displaying items in the order a user added them.

Example: BTreeMap for Ordered Operations

BTreeMap keeps its keys sorted. This makes it ideal for use cases that require ordered iteration or range queries, like a leaderboard or time-series data.

```
use std::collections::BTreeMap;

fn leaderboard() {
    // Scores are the keys, so they are kept sorted.
    let mut scores = BTreeMap::new();
    scores.insert(1500, "Alice".to_string());
    scores.insert(2200, "David".to_string());
    scores.insert(1800, "Charlie".to_string());

    // `iter()` returns items in sorted key order. `.rev()` gets us descending order for a
    // top-down leaderboard.
    println!("Leaderboard (Top 3):");
    for (score, name) in scores.iter().rev().take(3) {
        println!("- {}: {}", name, score);
    }

    // BTreeMap also supports efficient range queries.
    println!("\nPlayers with scores between 1500 and 2000:");
    for (score, name) in scores.range(1500..=2000) {
        println!("- {}: {}", name, score);
    }
}
```

Example: IndexMap for Insertion Order Preservation

IndexMap is a drop-in replacement for **HashMap** that remembers the order in which keys were inserted. This is useful for creating ordered JSON objects or any other scenario where order matters.

```
// Add `indexmap = "2.0"` to Cargo.toml
use indexmap::IndexMap;

fn ordered_json() {
    let mut user_data = IndexMap::new();

    // The insertion order is preserved.
    user_data.insert("id", "123".to_string());
    user_data.insert("name", "Alice".to_string());
```

```

user_data.insert("email", "alice@example.com".to_string());

// When serialized (e.g., to JSON), the fields will appear in the order they were
// inserted.
// This is not guaranteed with a standard HashMap.
let as_vec: Vec<_> = user_data.iter().map(|(k,v)| (k.to_string(), v.clone())).collect();
println!("Fields in insertion order: {:?}", as_vec);
}

```

Pattern 5: Concurrent HashMaps

A standard `HashMap` cannot be safely shared across multiple threads. While `Arc<Mutex<HashMap>>` is a valid approach, it suffers from heavy lock contention.

- **Problem:** - **High Contention:** Wrapping a `HashMap` in a `Mutex` or `RwLock` means that only one thread (for `Mutex`) or one writer (for `RwLock`) can access the map at a time, regardless of which key they are trying to access. On a multi-core machine, this becomes a major performance bottleneck.
- **Solution:** - **DashMap:** The `dashmap` crate provides a concurrent `HashMap` that is sharded internally. It stripes the map across many small, independent locks.
- **Why It Matters:** - **Scalability:** A concurrent map like `DashMap` can scale almost linearly with the number of CPU cores for many workloads, whereas a `Mutex`-wrapped `HashMap` does not scale at all. This is critical for building high-performance, multi-threaded applications like web servers, databases, and caches.
- **Use Cases:**
 - **High-Concurrency Caches:** A shared in-memory cache in a web server that is read from and written to by many request-handling threads.
 - **Session Stores:** Storing user session data that is accessed concurrently.
 - **Request Counters / Metrics:** Atomically incrementing counters for different endpoints or metrics from many threads.
 - **Any shared, mutable key-value store in a multi-threaded application.**

Example: Concurrent Request Counter with DashMap

`DashMap` provides an API similar to `HashMap` but is designed for high-concurrency scenarios. It allows multiple threads to read and write to the map at the same time with minimal blocking.

```

// Note: Add `dashmap = "5.5"` and `rayon = "1.8"` to Cargo.toml
use dashmap::DashMap;
use std::sync::Arc;
use rayon::prelude::*;

fn concurrent_request_counter() {
    let counters = Arc::new(DashMap::new());
    // Simulate 1000 concurrent requests to different endpoints.
}

```

```

(0..1000).into_par_iter().for_each(|i| {
    let endpoint = format!("/endpoint_{}", i % 10);
    // DashMap's entry API is similar to HashMap's and is thread-safe.
    *counters.entry(endpoint).or_insert(0) += 1;
});

println!("Request counts per endpoint:");
for entry in counters.iter() {
    println!("{}: {}", entry.key(), entry.value());
}
}
}

```

Key Takeaways**:

- Use Entry API to avoid double lookups
- Pre-allocate capacity when size is known
- Choose the right map type for your use case
- FxHashMap is 2-3x faster for integer keys
- DashMap is essential for high-concurrency scenarios
- BTreeMap when you need ordering or range queries

Performance Tips: - Reserve capacity before batch insertions - Use compact key types to reduce memory - Consider string interning for repeated strings - Profile before choosing hash function - DashMap for >4 concurrent writers

Advanced Collections

This chapter explores advanced collection types and data structures beyond the standard Vec and HashMap. We'll cover double-ended queues, priority queues, graph representations, prefix trees, and lock-free concurrent data structures through practical, real-world examples.

Pattern 1: VecDeque and Ring Buffers

Problem: Vec only supports O(1) operations at one end—`push_front()` requires shifting all elements making it O(N). Implementing queues (FIFO) with Vec is inefficient: either `pop(0)` is O(N) or reversing is needed.

Solution: Use `VecDeque<T>` which maintains a ring buffer internally with head/tail pointers. Operations `push_front()`, `push_back()`, `pop_front()`, `pop_back()` are all O(1).

Why It Matters: VecDeque enables efficient double-ended operations impossible with Vec. A task queue processing 1M items: Vec with `remove(0)` is O(N) per operation = O(N²) total.

Use Cases: FIFO queues (task processing, message queues), deques (double-ended queues), ring buffers (audio/video streaming, fixed-size logs), sliding windows (moving averages, pattern matching), BFS traversal, undo/redo stacks.

Example: Task Queue with Priority Lanes

A multi-lane task queue where tasks can be added and removed from both ends efficiently.

```
use std::collections::VecDeque;

#[derive(Debug, Clone, PartialEq, Eq)]
enum Priority {
    High,
    Normal,
    Low,
}

#[derive(Debug, Clone)]
struct Task {
    id: u64,
    description: String,
    priority: Priority,
}

struct TaskQueue {
    high_priority: VecDeque<Task>,
    normal_priority: VecDeque<Task>,
    low_priority: VecDeque<Task>,
    next_id: u64,
}

impl TaskQueue {
    fn new() -> Self {
        Self {
            high_priority: VecDeque::new(),
            normal_priority: VecDeque::new(),
            low_priority: VecDeque::new(),
            next_id: 1,
        }
    }

    fn enqueue(&mut self, description: String, priority: Priority) -> u64 {
        let id = self.next_id;
        self.next_id += 1;

        let task = Task {
            id,
            description,
            priority: priority.clone(),
        };

        match priority {
            Priority::High => self.high_priority.push_back(task),
            Priority::Normal => self.normal_priority.push_back(task),
            Priority::Low => self.low_priority.push_back(task),
        }
    }
}
```

```

        id
    }

fn enqueue_urgent(&mut self, description: String) -> u64 {
    let id = self.next_id;
    self.next_id += 1;

    let task = Task {
        id,
        description,
        priority: Priority::High,
    };

    // Add to front of high priority queue
    self.high_priority.push_front(task);
    id
}

fn dequeue(&mut self) -> Option<Task> {
    // Try high priority first
    if let Some(task) = self.high_priority.pop_front() {
        return Some(task);
    }

    // Then normal priority
    if let Some(task) = self.normal_priority.pop_front() {
        return Some(task);
    }

    // Finally low priority
    self.low_priority.pop_front()
}

fn peek(&self) -> Option<&Task> {
    self.high_priority
        .front()
        .or_else(|| self.normal_priority.front())
        .or_else(|| self.low_priority.front())
}

fn remove_by_id(&mut self, id: u64) -> Option<Task> {
    // Helper to remove from a specific queue
    fn remove_from_queue(queue: &mut VecDeque<Task>, id: u64) -> Option<Task> {
        let pos = queue.iter().position(|t| t.id == id)?;
        queue.remove(pos)
    }

    remove_from_queue(&mut self.high_priority, id)
        .or_else(|| remove_from_queue(&mut self.normal_priority, id))
        .or_else(|| remove_from_queue(&mut self.low_priority, id))
}

```

```

fn len(&self) -> usize {
    self.high_priority.len() + self.normal_priority.len() + self.low_priority.len()
}

fn is_empty(&self) -> bool {
    self.len() == 0
}

fn clear(&mut self) {
    self.high_priority.clear();
    self.normal_priority.clear();
    self.low_priority.clear();
}
}

//=====
// Example usage
//=====

fn main() {
    let mut queue = TaskQueue::new();

    queue.enqueue("Process data".to_string(), Priority::Normal);
    queue.enqueue("Backup database".to_string(), Priority::Low);
    queue.enqueue("Handle error".to_string(), Priority::High);
    queue.enqueue_urgent("Critical security patch".to_string());

    println!("Processing tasks:");
    while let Some(task) = queue.dequeue() {
        println!(" [{}] {}", task.priority, task.description);
    }
}

```

Key VecDeque Operations: - `push_front()` / `push_back()`: O(1) insertion at either end -
`pop_front()` / `pop_back()`: O(1) removal from either end - `front()` / `back()`: O(1) peek at either end
- Random access: O(1) with indexing

Example: Ring Buffer for Real-Time Data

A fixed-size circular buffer that overwrites oldest data when full, commonly used for sensor data, logging, and audio processing.

```

use std::collections::VecDeque;

struct RingBuffer<T> {
    buffer: VecDeque<T>,
    capacity: usize,
}

impl<T> RingBuffer<T> {
    fn new(capacity: usize) -> Self {
        Self {

```

```

        buffer: VecDeque::with_capacity(capacity),
        capacity,
    }
}

fn push(&mut self, item: T) {
    if self.buffer.len() >= self.capacity {
        self.buffer.pop_front(); // Remove oldest
    }
    self.buffer.push_back(item);
}

fn get(&self, index: usize) -> Option<&T> {
    self.buffer.get(index)
}

fn iter(&self) -> impl Iterator<Item = &T> {
    self.buffer.iter()
}

fn len(&self) -> usize {
    self.buffer.len()
}

fn is_full(&self) -> bool {
    self.buffer.len() >= self.capacity
}

fn clear(&mut self) {
    self.buffer.clear();
}

fn as_slice(&self) -> (&[T], &[T]) {
    self.buffer.as_slices()
}
}

//=====
// Specialized: Sliding window statistics
//=====

struct SlidingWindowStats {
    buffer: RingBuffer<f64>,
}

impl SlidingWindowStats {
    fn new(window_size: usize) -> Self {
        Self {
            buffer: RingBuffer::new(window_size),
        }
    }

    fn add(&mut self, value: f64) {
        self.buffer.push(value);
    }
}

```

```

}

fn mean(&self) -> Option<f64> {
    if self.buffer.len() == 0 {
        return None;
    }

    let sum: f64 = self.buffer.iter().sum();
    Some(sum / self.buffer.len() as f64)
}

fn min(&self) -> Option<f64> {
    self.buffer.iter().copied().min_by(|a, b| a.partial_cmp(b).unwrap())
}

fn max(&self) -> Option<f64> {
    self.buffer.iter().copied().max_by(|a, b| a.partial_cmp(b).unwrap())
}

fn variance(&self) -> Option<f64> {
    if self.buffer.len() < 2 {
        return None;
    }

    let mean = self.mean()?;
    let sum_squared_diff: f64 = self.buffer
        .iter()
        .map(|&x| (x - mean).powi(2))
        .sum();

    Some(sum_squared_diff / self.buffer.len() as f64)
}

fn std_dev(&self) -> Option<f64> {
    self.variance().map(|v| v.sqrt())
}
}

//=====
// Real-world example: Audio sample buffer
//=====

struct AudioBuffer {
    samples: RingBuffer<f32>,
    sample_rate: u32,
}

impl AudioBuffer {
    fn new(duration_seconds: f32, sample_rate: u32) -> Self {
        let capacity = (duration_seconds * sample_rate as f32) as usize;
        Self {
            samples: RingBuffer::new(capacity),
            sample_rate,
        }
    }
}

```

```
}

fn add_sample(&mut self, sample: f32) {
    self.samples.push(sample);
}

fn add_samples(&mut self, samples: &[f32]) {
    for &sample in samples {
        self.add_sample(sample);
    }
}

fn rms(&self) -> f32 {
    if self.samples.len() == 0 {
        return 0.0;
    }

    let sum_squares: f32 = self.samples.iter().map(|&s| s * s).sum();
    (sum_squares / self.samples.len() as f32).sqrt()
}

fn peak(&self) -> f32 {
    self.samples
        .iter()
        .map(|&s| s.abs())
        .max_by(|a, b| a.partial_cmp(b).unwrap())
        .unwrap_or(0.0)
}

fn zero_crossing_rate(&self) -> f32 {
    if self.samples.len() < 2 {
        return 0.0;
    }

    let mut crossings = 0;
    let samples: Vec<_> = self.samples.iter().copied().collect();

    for i in 0..samples.len() - 1 {
        if (samples[i] >= 0.0 && samples[i + 1] < 0.0)
            || (samples[i] < 0.0 && samples[i + 1] >= 0.0)
        {
            crossings += 1;
        }
    }

    crossings as f32 / (samples.len() - 1) as f32
}

//=====
// Example usage
//=====

fn main() {
```

```

println!("== Sliding Window Stats ==\n");

let mut stats = SlidingWindowStats::new(5);

for value in [10.0, 20.0, 15.0, 25.0, 30.0, 18.0, 22.0] {
    stats.add(value);
    println!("Added {}: mean={:.2}, std_dev={:.2}",
            value,
            stats.mean().unwrap_or(0.0),
            stats.std_dev().unwrap_or(0.0));
}

println!("\n== Audio Buffer ==\n");

let mut audio = AudioBuffer::new(0.1, 44100); // 100ms buffer at 44.1kHz

// Simulate sine wave
for i in 0..4410 {
    let t = i as f32 / 44100.0;
    let sample = (2.0 * std::f32::consts::PI * 440.0 * t).sin(); // 440 Hz
    audio.add_sample(sample * 0.5); // 50% amplitude
}

println!("RMS: {:.4}", audio.rms());
println!("Peak: {:.4}", audio.peak());
println!("Zero crossing rate: {:.4}", audio.zero_crossing_rate());
}

```

Ring Buffer Use Cases: - Sensor data buffering - Audio/video processing - Network packet buffering - Undo/redo history (fixed size) - Performance monitoring (sliding window)

Example: Deque-Based Sliding Window Maximum

Find the maximum value in every sliding window of size k in an array efficiently ($O(n)$ time).

```

use std::collections::VecDeque;

struct SlidingWindowMax {
    deque: VecDeque<(usize, i32)>, // (index, value)
    window_size: usize,
}

impl SlidingWindowMax {
    fn new(window_size: usize) -> Self {
        Self {
            deque: VecDeque::new(),
            window_size,
        }
    }

    fn add(&mut self, index: usize, value: i32) -> Option<i32> {
        if index >= self.window_size {
            self.deque.pop_front();
        }
        self.deque.push_back((index, value));
        Some(self.deque[0].1)
    }
}

```

```

// Remove elements outside window
while let Some(&(idx, _)) = self.deque.front() {
    if idx + self.window_size <= index {
        self.deque.pop_front();
    } else {
        break;
    }
}

// Remove elements smaller than current
while let Some(&(_, val)) = self.deque.back() {
    if val <= value {
        self.deque.pop_back();
    } else {
        break;
    }
}

self.deque.push_back((index, value));

// Return max if window is full
if index >= self.window_size - 1 {
    self.deque.front().map(|(_, val)| *val)
} else {
    None
}
}

fn max_in_windows(arr: &[i32], k: usize) -> Vec<i32> {
    let mut solver = Self::new(k);
    let mut result = Vec::new();

    for (i, &val) in arr.iter().enumerate() {
        if let Some(max) = solver.add(i, val) {
            result.push(max);
        }
    }

    result
}
}

//=====
// Real-world application: Stock price analysis
//=====

struct StockAnalyzer {
    prices: Vec<f64>,
}

impl StockAnalyzer {
    fn new(prices: Vec<f64>) -> Self {
        Self { prices }
    }
}

```

```
fn resistance_levels(&self, window_size: usize) -> Vec<f64> {
    self.sliding_max(window_size)
}

fn support_levels(&self, window_size: usize) -> Vec<f64> {
    self.sliding_min(window_size)
}

fn sliding_max(&self, window_size: usize) -> Vec<f64> {
    let mut deque = VecDeque::new();
    let mut result = Vec::new();

    for (i, &price) in self.prices.iter().enumerate() {
        // Remove old elements
        while let Some(&idx) = deque.front() {
            if idx + window_size <= i {
                deque.pop_front();
            } else {
                break;
            }
        }

        // Maintain decreasing order
        while let Some(&idx) = deque.back() {
            if self.prices[idx] <= price {
                deque.pop_back();
            } else {
                break;
            }
        }

        deque.push_back(i);

        if i >= window_size - 1 {
            result.push(self.prices[*deque.front().unwrap()]);
        }
    }

    result
}

fn sliding_min(&self, window_size: usize) -> Vec<f64> {
    let mut deque = VecDeque::new();
    let mut result = Vec::new();

    for (i, &price) in self.prices.iter().enumerate() {
        while let Some(&idx) = deque.front() {
            if idx + window_size <= i {
                deque.pop_front();
            } else {
                break;
            }
        }
    }
}
```

```

    }

    // Maintain increasing order (opposite of max)
    while let Some(&idx) = deque.back() {
        if self.prices[idx] >= price {
            deque.pop_back();
        } else {
            break;
        }
    }

    deque.push_back(i);

    if i >= window_size - 1 {
        result.push(self.prices[*deque.front().unwrap()]);
    }
}

result
}

fn volatility(&self, window_size: usize) -> Vec<f64> {
    let max_values = self.sliding_max(window_size);
    let min_values = self.sliding_min(window_size);

    max_values
        .iter()
        .zip(min_values.iter())
        .map(|(max, min)| max - min)
        .collect()
}
}

fn main() {
    println!("==== Sliding Window Maximum ====\n");

    let arr = vec![1, 3, -1, -3, 5, 3, 6, 7];
    let k = 3;

    let result = SlidingWindowMax::max_in_windows(&arr, k);
    println!("Array: {:?}", arr);
    println!("Window size: {}", k);
    println!("Maximums: {:?}", result);

    println!("\n==== Stock Analysis ====\n");

    let prices = vec![
        100.0, 102.0, 101.0, 105.0, 103.0, 108.0, 107.0, 110.0, 109.0, 112.0,
    ];

    let analyzer = StockAnalyzer::new(prices.clone());

    println!("Prices: {:?}", prices);
}

```

```

        println!("Resistance (5-day high): {:?}", analyzer.resistance_levels(5));
        println!("Support (5-day low): {:?}", analyzer.support_levels(5));
        println!("Volatility (5-day range): {:?}", analyzer.volatility(5));
    }
}

```

Algorithm Complexity: - Time: $O(n)$ - each element added/removed at most once - Space: $O(k)$ - deque size bounded by window size - Better than naive $O(n*k)$ approach

Pattern 2: BinaryHeap and Priority Queues

Problem: Maintaining a sorted collection with frequent insertions is expensive—sorting after each insert is $O(N \log N)$. Finding the min/max element in unsorted Vec is $O(N)$.

Solution: Use `BinaryHeap<T>` which implements a max-heap: $O(\log N)$ insertion, $O(\log N)$ pop of maximum, $O(1)$ peek at maximum. Wrap values in `Reverse<T>` for min-heap behavior.

Why It Matters: `BinaryHeap` provides optimal performance for priority operations. Task scheduler with 10K tasks: sorting after each insert = $O(N \log N)$ per insert.

Use Cases: Priority task scheduling, event simulation (process by timestamp), Dijkstra/A* pathfinding, top-K element finding (median, percentiles), merge K sorted lists, deadline scheduling, rate limiting.

Example: Task Scheduler with Deadlines

Schedule tasks based on priority and deadlines, ensuring high-priority tasks are executed first.

```

use std::collections::BinaryHeap;
use std::cmp::Ordering;

#[derive(Debug, Clone, Eq, PartialEq)]
struct Task {
    id: u64,
    priority: u32,
    deadline: u64,
    duration: u32,
    description: String,
}

impl Ord for Task {
    fn cmp(&self, other: &Self) -> Ordering {
        // First compare by priority (higher is better)
        match self.priority.cmp(&other.priority) {
            Ordering::Equal => {
                // Then by deadline (earlier is better, so reverse)
                other.deadline.cmp(&self.deadline)
            }
            other => other,
        }
    }
}

```

```

impl PartialOrd for Task {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        Some(self.cmp(other))
    }
}

struct TaskScheduler {
    heap: BinaryHeap<Task>,
    current_time: u64,
    next_id: u64,
}
impl TaskScheduler {
    fn new() -> Self {
        Self {
            heap: BinaryHeap::new(),
            current_time: 0,
            next_id: 1,
        }
    }

    fn schedule(&mut self, description: String, priority: u32, deadline: u64, duration: u32) {
        let task = Task {
            id: self.next_id,
            priority,
            deadline,
            duration,
            description,
        };
        self.next_id += 1;
        self.heap.push(task);
    }

    fn execute_next(&mut self) -> Option<Task> {
        let task = self.heap.pop()?;
        // Check if deadline missed
        if self.current_time > task.deadline {
            println!(
                "Warning: Task {} missed deadline (current={}, deadline={})",
                task.id, self.current_time, task.deadline
            );
        }

        self.current_time += task.duration as u64;
        Some(task)
    }

    fn peek(&self) -> Option<&Task> {
        self.heap.peek()
    }
}

```

```

fn pending_count(&self) -> usize {
    self.heap.len()
}

fn execute_all(&mut self) -> Vec<Task> {
    let mut executed = Vec::new();
    while let Some(task) = self.execute_next() {
        executed.push(task);
    }
    executed
}

fn get_current_time(&self) -> u64 {
    self.current_time
}
}

//=====
// Real-world example: CPU process scheduler
//=====

#[derive(Debug, Clone, Eq, PartialEq)]
struct Process {
    pid: u32,
    priority: i32,      // Higher is more important
    arrival_time: u64,
    burst_time: u32,
    remaining_time: u32,
}

impl Ord for Process {
    fn cmp(&self, other: &Self) -> Ordering {
        // Highest priority first
        match self.priority.cmp(&other.priority) {
            Ordering::Equal => {
                // Shortest remaining time first (SRT scheduling)
                other.remaining_time.cmp(&self.remaining_time)
            }
            other => other,
        }
    }
}

impl PartialOrd for Process {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        Some(self.cmp(other))
    }
}

struct CpuScheduler {
    ready_queue: BinaryHeap<Process>,
    current_time: u64,
}

```

```
impl CpuScheduler {
    fn new() -> Self {
        Self {
            ready_queue: BinaryHeap::new(),
            current_time: 0,
        }
    }

    fn add_process(&mut self, process: Process) {
        self.ready_queue.push(process);
    }

    fn run_time_slice(&mut self, time_slice: u32) -> Option<ProcessResult> {
        let mut process = self.ready_queue.pop()?;

        let executed = time_slice.min(process.remaining_time);
        process.remaining_time -= executed;
        self.current_time += executed as u64;

        let result = ProcessResult {
            pid: process.pid,
            time_executed: executed,
            completed: process.remaining_time == 0,
        };

        // Re-queue if not finished
        if process.remaining_time > 0 {
            self.ready_queue.push(process);
        }

        Some(result)
    }

    fn simulate(&mut self, time_slice: u32) {
        println!("Starting CPU scheduler simulation...\n");

        while let Some(result) = self.run_time_slice(time_slice) {
            println!(
                "Time {}: Process {} executed for {}ms {}",
                self.current_time,
                result.pid,
                result.time_executed,
                if result.completed { "(completed)" } else { "" }
            );
        }
    }
}

#[derive(Debug)]
struct ProcessResult {
    pid: u32,
    time_executed: u32,
    completed: bool,
```

```
}

fn main() {
    println!("==== Task Scheduler ====\n");

    let mut scheduler = TaskScheduler::new();

    scheduler.schedule("Write report".to_string(), 5, 100, 20);
    scheduler.schedule("Fix bug".to_string(), 10, 50, 15);
    scheduler.schedule("Code review".to_string(), 7, 80, 10);
    scheduler.schedule("Meeting".to_string(), 8, 60, 30);

    println!("Executing tasks in priority order:\n");
    let executed = scheduler.execute_all();

    for task in executed {
        println!(
            "Task {}: {} (priority={}, deadline={})",
            task.id, task.description, task.priority, task.deadline
        );
    }
}

println!("\n==== CPU Scheduler ====\n");

let mut cpu = CpuScheduler::new();

cpu.add_process(Process {
    pid: 1,
    priority: 5,
    arrival_time: 0,
    burst_time: 30,
    remaining_time: 30,
});

cpu.add_process(Process {
    pid: 2,
    priority: 10,
    arrival_time: 0,
    burst_time: 20,
    remaining_time: 20,
});

cpu.add_process(Process {
    pid: 3,
    priority: 7,
    arrival_time: 0,
    burst_time: 15,
    remaining_time: 15,
});

cpu.simulate(10); // 10ms time slices
}
```

BinaryHeap Characteristics: - Max-heap by default (largest element at top) - O(log n) push and pop
- O(1) peek - Good for: priority queues, event scheduling, top-k problems

Example: K-way Merge and Median Tracking

Merge k sorted lists efficiently, and track the median of a stream of numbers

```
use std::collections::BinaryHeap;
use std::cmp::{Ordering, Reverse};

//=====
// K-way merge: merge k sorted iterators
//=====

struct KWayMerge<T> {
    heap: BinaryHeap<MergeItem<T>>,
}

struct MergeItem<T> {
    value: T,
    source_id: usize,
}

impl<T: Ord> Ord for MergeItem<T> {
    fn cmp(&self, other: &Self) -> Ordering {
        // Reverse for min-heap behavior
        other.value.cmp(&self.value)
    }
}

impl<T: Ord> PartialOrd for MergeItem<T> {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        Some(self.cmp(other))
    }
}

impl<T: Ord> Eq for MergeItem<T> {}

impl<T: Ord> PartialEq for MergeItem<T> {
    fn eq(&self, other: &Self) -> bool {
        self.value == other.value
    }
}

impl<T: Ord + Clone> KWayMerge<T> {
    fn merge(lists: Vec<Vec<T>>) -> Vec<T> {
        let mut heap = BinaryHeap::new();
        let mut iter: Vec<_> = lists.into_iter().map(|v| v.into_iter()).collect();

        // Initialize heap with first element from each list
        for (id, iter) in iter.iter_mut().enumerate() {
            if let Some(value) = iter.next() {
```

```

        heap.push(MergeItem {
            value,
            source_id: id,
        });
    }
}

let mut result = Vec::new();

while let Some(item) = heap.pop() {
    result.push(item.value);

    // Get next element from same source
    if let Some(value) = iters[item.source_id].next() {
        heap.push(MergeItem {
            value,
            source_id: item.source_id,
        });
    }
}

result
}
}

//=====
// Running median tracker using two heaps
//=====

struct MedianTracker {
    lower_half: BinaryHeap<i32>,           // max-heap
    upper_half: BinaryHeap<Reverse<i32>>, // min-heap
}

impl MedianTracker {
    fn new() -> Self {
        Self {
            lower_half: BinaryHeap::new(),
            upper_half: BinaryHeap::new(),
        }
    }

    fn add(&mut self, num: i32) {
        // Add to appropriate heap
        if self.lower_half.is_empty() || num <= *self.lower_half.peek().unwrap() {
            self.lower_half.push(num);
        } else {
            self.upper_half.push(Reverse(num));
        }

        // Rebalance: ensure size difference <= 1
        if self.lower_half.len() > self.upper_half.len() + 1 {
            if let Some(val) = self.lower_half.pop() {
                self.upper_half.push(Reverse(val));
            }
        }
    }
}

```

```

        }
    } else if self.upper_half.len() > self.lower_half.len() {
        if let Some(Reverse(val)) = self.upper_half.pop() {
            self.lower_half.push(val);
        }
    }
}

fn median(&self) -> Option<f64> {
    if self.lower_half.is_empty() && self.upper_half.is_empty() {
        return None;
    }

    if self.lower_half.len() > self.upper_half.len() {
        Some(*self.lower_half.peek().unwrap() as f64)
    } else if self.upper_half.len() > self.lower_half.len() {
        Some(self.upper_half.peek().unwrap().0 as f64)
    } else {
        let lower = *self.lower_half.peek().unwrap() as f64;
        let upper = self.upper_half.peek().unwrap().0 as f64;
        Some((lower + upper) / 2.0)
    }
}

fn count(&self) -> usize {
    self.lower_half.len() + self.upper_half.len()
}

//=====
// Real-world: External sort for large files
//=====

struct ExternalSorter {
    chunk_size: usize,
}

impl ExternalSorter {
    fn new(chunk_size: usize) -> Self {
        Self { chunk_size }
    }

    fn sort(&self, data: Vec<i32>) -> Vec<i32> {
        // Phase 1: Sort chunks
        let mut chunks: Vec<Vec<i32>> = data
            .chunks(self.chunk_size)
            .map(|chunk| {
                let mut sorted = chunk.to_vec();
                sorted.sort();
                sorted
            })
            .collect();

        // Phase 2: K-way merge
    }
}

```

```

        KWayMerge::merge(chunks)
    }

fn main() {
    println!("==== K-Way Merge ====\n");

    let lists = vec![
        vec![1, 4, 7, 10],
        vec![2, 5, 8, 11],
        vec![3, 6, 9, 12],
    ];

    let merged = KWayMerge::merge(lists.clone());
    println!("Input lists: {:?}", lists);
    println!("Merged: {:?}", merged);

    println!("\n==== Running Median ====\n");

    let mut tracker = MedianTracker::new();

    for num in [5, 15, 1, 3, 8, 7, 9, 2] {
        tracker.add(num);
        println!("Added {}: median = {:.1}", num, tracker.median().unwrap());
    }

    println!("\n==== External Sort ====\n");

    let data: Vec<i32> = (0..20).rev().collect();
    println!("Unsorted: {:?}", data);

    let sorter = ExternalSorter::new(5);
    let sorted = sorter.sort(data);
    println!("Sorted: {:?}", sorted);
}

```

Median Tracker Analysis: - Time: $O(\log n)$ per insertion - Space: $O(n)$ - Works by maintaining two heaps: max-heap (lower half) and min-heap (upper half) - Median is either top of one heap or average of both tops

Example: Top-K Frequent Elements

Find the k most frequent elements in a stream efficiently.

```

use std::collections::{HashMap, BinaryHeap};
use std::cmp::{Ordering, Reverse};

#[derive(Eq, PartialEq)]
struct FreqItem<T> {
    item: T,
    count: usize,
}

```

```

}

impl<T: Eq> Ord for FreqItem<T> {
    fn cmp(&self, other: &Self) -> Ordering {
        self.count.cmp(&other.count)
    }
}

impl<T: Eq> PartialOrd for FreqItem<T> {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        Some(self.cmp(other))
    }
}

struct TopKFrequent<T> {
    counts: HashMap<T, usize>,
    k: usize,
}

impl<T> TopKFrequent<T>
where
    T: Eq + std::hash::Hash + Clone,
{
    fn new(k: usize) -> Self {
        Self {
            counts: HashMap::new(),
            k,
        }
    }

    fn add(&mut self, item: T) {
        *self.counts.entry(item).or_insert(0) += 1;
    }

    fn add_batch(&mut self, items: Vec<T>) {
        for item in items {
            self.add(item);
        }
    }

    fn top_k(&self) -> Vec<(T, usize)> {
        // Use min-heap to keep only top k
        let mut heap: BinaryHeap<Reverse<FreqItem<&T>> = BinaryHeap::new();

        for (item, &count) in &self.counts {
            heap.push(Reverse(FreqItem { item, count }));

            if heap.len() > self.k {
                heap.pop();
            }
        }

        heap.into_iter()
    }
}

```

```

        .map(|Reverse(freq_item)| (freq_item.item.clone(), freq_item.count))
        .collect()
    }

    fn top_k_sorted(&self) -> Vec<(T, usize)> {
        let mut result = self.top_k();
        result.sort_by(|a, b| b.1.cmp(&a.1));
        result
    }
}

//=====
// Real-world: Log analysis
//=====

struct LogAnalyzer {
    error_tracker: TopKFrequent<String>,
    ip_tracker: TopKFrequent<String>,
    endpoint_tracker: TopKFrequent<String>,
}

impl LogAnalyzer {
    fn new(k: usize) -> Self {
        Self {
            error_tracker: TopKFrequent::new(k),
            ip_tracker: TopKFrequent::new(k),
            endpoint_tracker: TopKFrequent::new(k),
        }
    }

    fn process_log(&mut self, log_entry: LogEntry) {
        if let Some(error) = log_entry.error {
            self.error_tracker.add(error);
        }
        self.ip_tracker.add(log_entry.ip);
        self.endpoint_tracker.add(log_entry.endpoint);
    }

    fn report(&self) {
        println!("Top Errors:");
        for (error, count) in self.error_tracker.top_k_sorted() {
            println!("  {}: {}", error, count);
        }

        println!("\nTop IP Addresses:");
        for (ip, count) in self.ip_tracker.top_k_sorted() {
            println!("  {}: {}", ip, count);
        }

        println!("\nTop Endpoints:");
        for (endpoint, count) in self.endpoint_tracker.top_k_sorted() {
            println!("  {}: {}", endpoint, count);
        }
    }
}

```

```
}

#[derive(Debug, Clone)]
struct LogEntry {
    ip: String,
    endpoint: String,
    error: Option<String>,
}

fn main() {
    println!("==== Top-K Frequent Elements ====\n");

    let mut tracker = TopKFrequent::new(3);

    let words = vec![
        "apple", "banana", "apple", "cherry", "banana", "apple",
        "date", "banana", "apple", "cherry",
    ];
    tracker.add_batch(words.iter().map(|&s| s.to_string()).collect());

    println!("Top 3 words:");
    for (word, count) in tracker.top_k_sorted() {
        println!(" {}: {}", word, count);
    }

    println!("\n==== Log Analysis ====\n");

    let mut analyzer = LogAnalyzer::new(3);

    // Simulate logs
    let logs = vec![
        LogEntry {
            ip: "192.168.1.1".to_string(),
            endpoint: "/api/users".to_string(),
            error: None,
        },
        LogEntry {
            ip: "192.168.1.2".to_string(),
            endpoint: "/api/posts".to_string(),
            error: Some("404 Not Found".to_string()),
        },
        LogEntry {
            ip: "192.168.1.1".to_string(),
            endpoint: "/api/users".to_string(),
            error: None,
        },
        LogEntry {
            ip: "192.168.1.3".to_string(),
            endpoint: "/api/posts".to_string(),
            error: Some("500 Internal Error".to_string()),
        },
        LogEntry {
```

```

        ip: "192.168.1.1".to_string(),
        endpoint: "/api/comments".to_string(),
        error: Some("404 Not Found".to_string()),
    },
];

for log in logs {
    analyzer.process_log(log);
}

analyzer.report();
}

```

Top-K Pattern: - Maintain min-heap of size k - For each element, add to heap and remove smallest if size > k - Time: $O(n \log k)$ vs $O(n \log n)$ for full sort - Space: $O(k)$ for heap vs $O(n)$ for sorting all elements

Pattern 3: Graph Representations

Problem: Naive graph implementations with recursive structures hit Rust's ownership rules—nodes can't mutually reference each other without causing cycles. Using `Rc<RefCell<Node>>` everywhere is verbose and has runtime overhead.

Solution: Use adjacency list with `Vec<Vec<u8>>` (node IDs as indices) for most graphs. Use adjacency matrix `Vec<Vec<bool>>` for dense graphs or when edge checks must be $O(1)$.

Why It Matters: Graph representation determines algorithm performance. Dijkstra's with adjacency list: $O((V+E) \log V)$.

Use Cases: Adjacency list for social networks, dependency graphs, road networks (sparse). Adjacency matrix for complete graphs, grid-based pathfinding, dense weighted graphs. HashMap-based for dynamic graphs (adding/removing nodes), unknown node sets.

Example: Adjacency List with Weighted Edges

A graph with weighted edges for algorithms like Dijkstra's shortest path.

```

use std::collections::{HashMap, BinaryHeap, HashSet};
use std::cmp::Ordering;
use std::hash::Hash;

#[derive(Debug, Clone)]
struct Edge<T> {
    to: T,
    weight: u32,
}

struct WeightedGraph<T> {
    adjacency: HashMap<T, Vec<Edge<T>>>,
    directed: bool,
}

```

```
}

impl<T> WeightedGraph<T>
where
    T: Eq + Hash + Clone,
{
    fn new(directed: bool) -> Self {
        Self {
            adjacency: HashMap::new(),
            directed,
        }
    }

    fn add_vertex(&mut self, vertex: T) {
        self.adjacency.entry(vertex).or_insert_with(Vec::new);
    }

    fn add_edge(&mut self, from: T, to: T, weight: u32) {
        self.adjacency
            .entry(from.clone())
            .or_insert_with(Vec::new)
            .push(Edge {
                to: to.clone(),
                weight,
            });

        if !self.directed {
            self.adjacency
                .entry(to)
                .or_insert_with(Vec::new)
                .push(Edge { to: from, weight });
        }
    }

    fn neighbors(&self, vertex: &T) -> Option<&Vec<Edge<T>>> {
        self.adjacency.get(vertex)
    }

    fn vertices(&self) -> Vec<&T> {
        self.adjacency.keys().collect()
    }

    fn edge_count(&self) -> usize {
        let total: usize = self.adjacency.values().map(|edges| edges.len()).sum();
        if self.directed {
            total
        } else {
            total / 2
        }
    }
}

//=====
```

```

// Dijkstra's shortest path
//=====
#[derive(Eq, PartialEq)]
struct State<T> {
    cost: u32,
    node: T,
}

impl<T: Eq> Ord for State<T> {
    fn cmp(&self, other: &Self) -> Ordering {
        other.cost.cmp(&self.cost) // Min-heap
    }
}

impl<T: Eq> PartialOrd for State<T> {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        Some(self.cmp(other))
    }
}

impl<T> WeightedGraph<T>
where
    T: Eq + Hash + Clone,
{
    fn dijkstra(&self, start: &T) -> HashMap<T, u32> {
        let mut distances: HashMap<T, u32> = HashMap::new();
        let mut heap = BinaryHeap::new();

        distances.insert(start.clone(), 0);
        heap.push(State {
            cost: 0,
            node: start.clone(),
        });

        while let Some(State { cost, node }) = heap.pop() {
            // Skip if we found a better path
            if let Some(&best) = distances.get(&node) {
                if cost > best {
                    continue;
                }
            }

            // Check neighbors
            if let Some(edges) = self.neighbors(&node) {
                for edge in edges {
                    let next_cost = cost + edge.weight;

                    let is_better = distances
                        .get(&edge.to)
                        .map_or(true, |&current| next_cost < current);

                    if is_better {
                        distances.insert(edge.to.clone(), next_cost);
                    }
                }
            }
        }

        distances
    }
}

```

```

                heap.push(State {
                    cost: next_cost,
                    node: edge.to.clone(),
                });
            }
        }
    }
}

distances
}

fn shortest_path(&self, start: &T, end: &T) -> Option<(Vec<T>, u32)> {
    let mut distances: HashMap<T, u32> = HashMap::new();
    let mut previous: HashMap<T, T> = HashMap::new();
    let mut heap = BinaryHeap::new();

    distances.insert(start.clone(), 0);
    heap.push(State {
        cost: 0,
        node: start.clone(),
    });

    while let Some(State { cost, node }) = heap.pop() {
        if node == *end {
            // Reconstruct path
            let mut path = vec![end.clone()];
            let mut current = end;

            while let Some(prev) = previous.get(current) {
                path.push(prev.clone());
                current = prev;
            }

            path.reverse();
            return Some((path, cost));
        }

        if let Some(&best) = distances.get(&node) {
            if cost > best {
                continue;
            }
        }

        if let Some(edges) = self.neighbors(&node) {
            for edge in edges {
                let next_cost = cost + edge.weight;

                let is_better = distances
                    .get(&edge.to)
                    .map_or(true, |&current| next_cost < current);

                if is_better {

```

```

            distances.insert(edge.to.clone(), next_cost);
            previous.insert(edge.to.clone(), node.clone());
            heap.push(State {
                cost: next_cost,
                node: edge.to.clone(),
            });
        }
    }
}

None
}

//=====
// Real-world: Route planning
//=====

fn main() {
    println!("==> Weighted Graph - Route Planning ==\n");

    let mut map = WeightedGraph::new(false);

    // Cities and distances (km)
    map.add_edge("SF", "LA", 383);
    map.add_edge("SF", "Portland", 635);
    map.add_edge("LA", "Phoenix", 373);
    map.add_edge("Portland", "Seattle", 173);
    map.add_edge("Phoenix", "Denver", 868);
    map.add_edge("Seattle", "Denver", 1316);
    map.add_edge("LA", "Denver", 1016);

    println!("Finding shortest paths from SF:\n");
    let distances = map.dijkstra(&"SF");

    for (city, distance) in &distances {
        println!("  SF -> {}: {}km", city, distance);
    }

    println!("\n Shortest path SF -> Denver:");
    if let Some((path, distance)) = map.shortest_path(&"SF", &"Denver") {
        println!("  Path: {:?}", path);
        println!("  Distance: {}km", distance);
    }
}

```

Graph Representation Trade-offs: - **Adjacency List:** Space $O(V + E)$, good for sparse graphs - **Adjacency Matrix:** Space $O(V^2)$, good for dense graphs, $O(1)$ edge lookup - **Edge List:** Simple, good for iterating all edges

Example: Topological Sort and Dependency Resolution

Order tasks respecting dependencies, detect cycles in dependency graphs.

```
use std::collections::{HashMap, HashSet, VecDeque};
use std::hash::Hash;

struct DirectedGraph<T> {
    adjacency: HashMap<T, Vec<T>>,
}

impl<T> DirectedGraph<T>
where
    T: Eq + Hash + Clone,
{
    fn new() -> Self {
        Self {
            adjacency: HashMap::new(),
        }
    }

    fn add_edge(&mut self, from: T, to: T) {
        self.adjacency
            .entry(from.clone())
            .or_insert_with(Vec::new)
            .push(to.clone());

        // Ensure 'to' vertex exists
        self.adjacency.entry(to).or_insert_with(Vec::new);
    }

    fn vertices(&self) -> Vec<&T> {
        self.adjacency.keys().collect()
    }

    // Kahn's algorithm for topological sort
    fn topological_sort(&self) -> Result<Vec<T>, String> {
        let mut in_degree: HashMap<T, usize> = HashMap::new();

        // Calculate in-degrees
        for vertex in self.vertices() {
            in_degree.entry(vertex.clone()).or_insert(0);
        }

        for edges in self.adjacency.values() {
            for to in edges {
                *in_degree.entry(to.clone()).or_insert(0) += 1;
            }
        }

        // Queue vertices with no incoming edges
        let mut queue: VecDeque<T> = in_degree
```

```

    .iter()
    .filter(|(v, &degree)| degree == 0)
    .map(|(v, _)| v.clone())
    .collect();

let mut result = Vec::new();

while let Some(vertex) = queue.pop_front() {
    result.push(vertex.clone());

    // Reduce in-degree for neighbors
    if let Some(edges) = self.adjacency.get(&vertex) {
        for to in edges {
            if let Some(degree) = in_degree.get_mut(to) {
                *degree -= 1;
                if *degree == 0 {
                    queue.push_back(to.clone());
                }
            }
        }
    }
}

// Check for cycles
if result.len() != self.adjacency.len() {
    Err("Graph contains a cycle".to_string())
} else {
    Ok(result)
}
}

// DFS-based topological sort
fn topological_sort_dfs(&self) -> Result<Vec<T>, String> {
    let mut visited = HashSet::new();
    let mut rec_stack = HashSet::new();
    let mut result = Vec::new();

    for vertex in self.vertices() {
        if !visited.contains(vertex) {
            self.dfs_topo(
                vertex,
                &mut visited,
                &mut rec_stack,
                &mut result,
            )?;
        }
    }

    result.reverse();
    Ok(result)
}

fn dfs_topo(

```

```

    &self,
    vertex: &T,
    visited: &mut HashSet<T>,
    rec_stack: &mut HashSet<T>,
    result: &mut Vec<T>,
) -> Result<(), String> {
    visited.insert(vertex.clone());
    rec_stack.insert(vertex.clone());

    if let Some(edges) = self.adjacency.get(vertex) {
        for neighbor in edges {
            if !visited.contains(neighbor) {
                self.dfs_topo(neighbor, visited, rec_stack, result)?;
            } else if rec_stack.contains(neighbor) {
                return Err(format!("Cycle detected involving {:?}", neighbor));
            }
        }
    }

    rec_stack.remove(vertex);
    result.push(vertex.clone());
    Ok(())
}

fn has_cycle(&self) -> bool {
    self.topological_sort().is_err()
}
}

//=====
// Real-world: Build system dependency resolution
//=====

struct BuildSystem {
    dependencies: DirectedGraph<String>,
}

impl BuildSystem {
    fn new() -> Self {
        Self {
            dependencies: DirectedGraph::new(),
        }
    }

    fn add_target(&mut self, target: String, depends_on: Vec<String>) {
        for dep in depends_on {
            self.dependencies.add_edge(dep, target.clone());
        }
    }

    fn build_order(&self) -> Result<Vec<String>, String> {
        self.dependencies.topological_sort()
    }
}

```

```

fn check_cycles(&self) -> bool {
    self.dependencies.has_cycle()
}

}

//=====
// Real-world: Course prerequisite planning
//=====

struct CoursePlanner {
    prerequisites: DirectedGraph<String>,
}

impl CoursePlanner {
    fn new() -> Self {
        Self {
            prerequisites: DirectedGraph::new(),
        }
    }

    fn add_course(&mut self, course: String, prerequisites: Vec<String>) {
        for prereq in prerequisites {
            self.prerequisites.add_edge(prereq, course.clone());
        }
    }

    fn course_order(&self) -> Result<Vec<String>, String> {
        self.prerequisites.topological_sort()
    }

    fn can_complete(&self) -> bool {
        !self.prerequisites.has_cycle()
    }
}

fn main() {
    println!("==== Build System ===\n");

    let mut build = BuildSystem::new();

    build.add_target("main.o".to_string(), vec!["main.c".to_string(), "util.h".to_string()]);
    build.add_target("util.o".to_string(), vec!["util.c".to_string(), "util.h".to_string()]);
    build.add_target("program".to_string(), vec![ "main.o".to_string(), "util.o".to_string()]);

    match build.build_order() {
        Ok(order) => {
            println!("Build order:");
            for (i, target) in order.iter().enumerate() {
                println!(" {}.", "{}", i + 1, target);
            }
        }
        Err(e) => println!("Error: {}", e),
    }
}

```

```

println!("==> Course Planning ==>");

let mut planner = CoursePlanner::new();

planner.add_course("Data Structures".to_string(), vec!["Programming 101".to_string()]);
planner.add_course("Algorithms".to_string(), vec![ "Data Structures".to_string()]);
planner.add_course("AI".to_string(), vec![ "Algorithms".to_string()], "Linear
    Algebra".to_string());
planner.add_course("Machine Learning".to_string(), vec![ "AI".to_string(),
    "Statistics".to_string()]);

if planner.can_complete() {
    match planner.course_order() {
        Ok(order) => {
            println!("Suggested course order:");
            for (i, course) in order.iter().enumerate() {
                println!(" Semester {}: {}", (i / 2) + 1, course);
            }
        }
        Err(e) => println!("Error: {}", e),
    }
} else {
    println!("Cannot complete - circular prerequisites!");
}
}

```

Topological Sort Algorithms: 1. **Kahn's Algorithm** (BFS-based): - Time: $O(V + E)$ - Easier to detect cycles - Produces one valid ordering

1. DFS-based:

- Time: $O(V + E)$
 - Can find strongly connected components
 - Multiple valid orderings possible
-

Pattern 4: Trie and Radix Tree Structures

Problem: Finding all strings with a given prefix in HashMap requires checking every key— $O(N)$ with N strings. Autocomplete for 1M words checks all 1M.

Solution: Use Trie (prefix tree) where each node represents a character, paths from root spell strings. Prefix search is $O(M)$ where M is prefix length, not number of strings.

Why It Matters: Tries enable efficient prefix operations impossible with hash tables. Autocomplete in 1M-word dictionary: HashMap $O(N)$ scan per query.

Use Cases: Autocomplete (search engines, IDEs, command completion), spell checkers (dictionary lookup, suggestions), IP routing (longest prefix match), phonebook search by prefix, DNA sequence matching, text compression (shared prefix storage).

Example: Trie for Autocomplete and Prefix Search

Autocomplete with fast prefix matching.

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

#[derive(Default, Debug)]
struct TrieNode {
    children: HashMap<char, TrieNode>,
    is_end: bool,
    count: usize, // Frequency/popularity
}

struct Trie {
    root: TrieNode,
}

impl Trie {
    fn new() -> Self {
        Self {
            root: TrieNode::default(),
        }
    }

    fn insert(&mut self, word: &str) {
        self.insert_with_count(word, 1);
    }

    fn insert_with_count(&mut self, word: &str, count: usize) {
        let mut node = &mut self.root;

        for ch in word.chars() {
            node = node.children.entry(ch).or_default();
        }

        node.is_end = true;
        node.count += count;
    }

    fn search(&self, word: &str) -> bool {
        self.find_node(word).map_or(false, |node| node.is_end)
    }

    fn starts_with(&self, prefix: &str) -> bool {
        self.find_node(prefix).is_some()
    }

    fn find_node(&self, prefix: &str) -> Option<&TrieNode> {
        let mut node = &self.root;

        for ch in prefix.chars() {
            node = node.children.get(&ch)?;
        }

        if !node.is_end {
            return None;
        }

        return Some(node);
    }
}
```

```
    }

    Some(node)
}

fn autocomplete(&self, prefix: &str) -> Vec<String> {
    let mut results = Vec::new();

    if let Some(node) = self.find_node(prefix) {
        self.collect_words(node, prefix.to_string(), &mut results);
    }

    results
}

fn collect_words(&self, node: &TrieNode, current: String, results: &mut Vec<String>) {
    if node.is_end {
        results.push(current.clone());
    }

    for (&ch, child) in &node.children {
        let mut next = current.clone();
        next.push(ch);
        self.collect_words(child, next, results);
    }
}

fn top_k_autocomplete(&self, prefix: &str, k: usize) -> Vec<(String, usize)> {
    let mut results = Vec::new();

    if let Some(node) = self.find_node(prefix) {
        self.collect_words_with_count(node, prefix.to_string(), &mut results);
    }

    // Sort by count (descending) and take top k
    results.sort_by(|a, b| b.1.cmp(&a.1));
    results.truncate(k);
    results
}

fn collect_words_with_count(
    &self,
    node: &TrieNode,
    current: String,
    results: &mut Vec<(String, usize)>,
) {
    if node.is_end {
        results.push((current.clone(), node.count));
    }

    for (&ch, child) in &node.children {
        let mut next = current.clone();
        next.push(ch);
```

```

        self.collect_words_with_count(child, next, results);
    }

fn delete(&mut self, word: &str) -> bool {
    self.delete_helper(&mut self.root, word, 0)
}

fn delete_helper(&mut self, node: &mut TrieNode, word: &str, index: usize) -> bool {
    if index == word.len() {
        if !node.is_end {
            return false;
        }
        node.is_end = false;
        return node.children.is_empty();
    }

    let ch = word.chars().nth(index).unwrap();

    if let Some(child) = node.children.get_mut(&ch) {
        let should_delete = self.delete_helper(child, word, index + 1);

        if should_delete {
            node.children.remove(&ch);
            return !node.is_end && node.children.is_empty();
        }
    }
}

false
}

//=====
// Real-world: Search engine autocomplete
//=====

struct SearchAutocomplete {
    trie: Trie,
}

impl SearchAutocomplete {
    fn new() -> Self {
        Self {
            trie: Trie::new(),
        }
    }

    fn add_search_query(&mut self, query: &str) {
        // Normalize: lowercase
        let normalized = query.to_lowercase();
        self.trie.insert_with_count(&normalized, 1);
    }

    fn suggest(&self, prefix: &str, limit: usize) -> Vec<(String, usize)> {

```

```

        let normalized = prefix.to_lowercase();
        self.trie.top_k_autocomplete(&normalized, limit)
    }
}

//=====
// Real-world: Dictionary with spell check
//=====

struct Dictionary {
    trie: Trie,
}

impl Dictionary {
    fn new() -> Self {
        Self {
            trie: Trie::new(),
        }
    }

    fn add_word(&mut self, word: &str) {
        self.trie.insert(&word.to_lowercase());
    }

    fn contains(&self, word: &str) -> bool {
        self.trie.search(&word.to_lowercase())
    }

    fn suggest_corrections(&self, word: &str, maxSuggestions: usize) -> Vec<String> {
        let word = word.to_lowercase();

        // Try prefixes of increasing length
        for len in (1..=word.len()).rev() {
            let prefix = &word[..len];
            let suggestions = self.trie.autocomplete(prefix);

            if !suggestions.is_empty() {
                let mut results: Vec<_> = suggestions
                    .into_iter()
                    .filter(|s| self.edit_distance(s, &word) <= 2)
                    .collect();

                results.truncate(maxSuggestions);

                if !results.is_empty() {
                    return results;
                }
            }
        }
        vec![]
    }

    fn edit_distance(&self, s1: &str, s2: &str) -> usize {

```

```

let len1 = s1.chars().count();
let len2 = s2.chars().count();

let mut dp = vec![vec![0; len2 + 1]; len1 + 1];

for i in 0..=len1 {
    dp[i][0] = i;
}
for j in 0..=len2 {
    dp[0][j] = j;
}

let s1_chars: Vec<char> = s1.chars().collect();
let s2_chars: Vec<char> = s2.chars().collect();

for i in 1..=len1 {
    for j in 1..=len2 {
        let cost = if s1_chars[i - 1] == s2_chars[j - 1] {
            0
        } else {
            1
        };

        dp[i][j] = (dp[i - 1][j] + 1)
                    .min(dp[i][j - 1] + 1)
                    .min(dp[i - 1][j - 1] + cost);
    }
}

dp[len1][len2]
}

fn main() {
    println!("==== Autocomplete ====\n");

    let mut autocomplete = SearchAutocomplete::new();

    // Simulate search queries
    autocomplete.add_search_query("rust programming");
    autocomplete.add_search_query("rust tutorial");
    autocomplete.add_search_query("rust tutorial");
    autocomplete.add_search_query("rust book");
    autocomplete.add_search_query("python programming");

    println!("Suggestions for 'rust':");
    for (query, count) in autocomplete.suggest("rust", 5) {
        println!(" {} (searched {} times)", query, count);
    }

    println!("\n==== Dictionary ====\n");

    let mut dict = Dictionary::new();
}

```

```

for word in ["hello", "help", "helper", "world", "word", "work"] {
    dict.add_word(word);
}

let test_word = "heloo";
println!("Is '{}' in dictionary? {}", test_word, dict.contains(test_word));

println!("Suggestions for '{}':", test_word);
for suggestion in dict.suggest_corrections(test_word, 3) {
    println!("  {}", suggestion);
}
}

```

Trie Complexity: - Insert: O(m) where m = word length - Search: O(m) - Space: O(ALPHABET_SIZE * N * M) worst case - Ideal for: autocomplete, spell check, IP routing

Example: Radix Tree for Compressed Trie

A space-efficient radix tree (compressed trie) for storing strings with common prefixes.

```

use std::collections::HashMap;

#[derive(Debug)]
struct RadixNode {
    children: HashMap<char, Box<RadixNode>>,
    edge_label: String,
    is_end: bool,
    value: Option<String>,
}

impl RadixNode {
    fn new(label: String) -> Self {
        Self {
            children: HashMap::new(),
            edge_label: label,
            is_end: false,
            value: None,
        }
    }
}

struct RadixTree {
    root: RadixNode,
    size: usize,
}

impl RadixTree {
    fn new() -> Self {
        Self {
            root: RadixNode::new(String::new()),
        }
    }
}

```

```

        size: 0,
    }

}

fn insert(&mut self, key: &str, value: String) {
    if key.is_empty() {
        return;
    }

    self.insert_helper(&mut self.root, key, value);
    self.size += 1;
}

fn insert_helper(&mut self, node: &mut RadixNode, key: &str, value: String) {
    if key.is_empty() {
        node.is_end = true;
        node.value = Some(value);
        return;
    }

    let first_char = key.chars().next().unwrap();

    // Find matching child
    if let Some(child) = node.children.get_mut(&first_char) {
        let label = &child.edge_label;
        let common_prefix_len = common_prefix_length(key, label);

        if common_prefix_len == label.len() {
            // Full match: continue down
            let remaining = &key[common_prefix_len..];
            self.insert_helper(child, remaining, value);
        } else {
            // Partial match: split node
            let old_label = label.clone();
            let common = &old_label[..common_prefix_len];
            let old_suffix = &old_label[common_prefix_len..];
            let new_suffix = &key[common_prefix_len..];

            // Create new intermediate node
            let mut intermediate = Box::new(RadixNode::new(common.to_string()));

            // Move old child under intermediate
            let old_child = node.children.remove(&first_char).unwrap();
            let old_first = old_suffix.chars().next().unwrap();

            let mut relocated = old_child;
            relocated.edge_label = old_suffix.to_string();
            intermediate.children.insert(old_first, relocated);

            // Add new branch
            if !new_suffix.is_empty() {
                let new_first = new_suffix.chars().next().unwrap();
                let mut new_node = Box::new(RadixNode::new(new_suffix.to_string()));

```

```

        new_node.is_end = true;
        new_node.value = Some(value);
        intermediate.children.insert(new_first, new_node);
    } else {
        intermediate.is_end = true;
        intermediate.value = Some(value);
    }

    node.children.insert(first_char, intermediate);
}
} else {
    // No matching child: create new
    let mut new_node = Box::new(RadixNode::new(key.to_string()));
    new_node.is_end = true;
    new_node.value = Some(value);
    node.children.insert(first_char, new_node);
}
}

fn search(&self, key: &str) -> Option<&String> {
    self.search_helper(&self.root, key)
}

fn search_helper(&self, node: &RadixNode, key: &str) -> Option<&String> {
    if key.is_empty() {
        return if node.is_end {
            node.value.as_ref()
        } else {
            None
        };
    }

    let first_char = key.chars().next().unwrap();

    if let Some(child) = node.children.get(&first_char) {
        let label = &child.edge_label;
        let common_len = common_prefix_length(key, label);

        if common_len == label.len() {
            let remaining = &key[common_len..];
            self.search_helper(child, remaining)
        } else {
            None
        }
    } else {
        None
    }
}

fn starts_with(&self, prefix: &str) -> Vec<String> {
    let mut results = Vec::new();
    self.collect_with_prefix(&self.root, prefix, String::new(), &mut results);
    results
}
```

```

}

fn collect_with_prefix(
    &self,
    node: &RadixNode,
    remaining_prefix: &str,
    current_key: String,
    results: &mut Vec<String>,
) {
    if remaining_prefix.is_empty() {
        // Collect all keys under this node
        self.collect_all(node, current_key, results);
        return;
    }

    let first_char = remaining_prefix.chars().next().unwrap();

    if let Some(child) = node.children.get(&first_char) {
        let label = &child.edge_label;
        let common_len = common_prefix_length(remaining_prefix, label);

        let mut new_key = current_key.clone();
        new_key.push_str(&label[..common_len]);

        if common_len == label.len() {
            let new_remaining = &remaining_prefix[common_len..];
            self.collect_with_prefix(child, new_remaining, new_key, results);
        } else if common_len == remaining_prefix.len() {
            // Prefix matches completely
            self.collect_all(child, new_key, results);
        }
    }
}

fn collect_all(&self, node: &RadixNode, current_key: String, results: &mut Vec<String>) {
    if node.is_end {
        results.push(current_key.clone());
    }

    for (_, child) in &node.children {
        let mut new_key = current_key.clone();
        new_key.push_str(&child.edge_label);
        self.collect_all(child, new_key, results);
    }
}

fn len(&self) -> usize {
    self.size
}

fn common_prefix_length(s1: &str, s2: &str) -> usize {
    s1.chars()
}

```

```

        .zip(s2.chars())
        .take_while(|(a, b)| a == b)
        .count()
    }

//=====
// Real-world: IP routing table
//=====

struct RoutingTable {
    tree: RadixTree,
}

impl RoutingTable {
    fn new() -> Self {
        Self {
            tree: RadixTree::new(),
        }
    }

    fn add_route(&mut self, cidr: &str, gateway: &str) {
        self.tree.insert(cidr, gateway.to_string());
    }

    fn lookup(&self, ip: &str) -> Option<&String> {
        self.tree.search(ip)
    }

    fn routes_for_prefix(&self, prefix: &str) -> Vec<String> {
        self.tree.starts_with(prefix)
    }
}

fn main() {
    println!("== Radix Tree ==\n");

    let mut tree = RadixTree::new();

    tree.insert("test", "value1".to_string());
    tree.insert("testing", "value2".to_string());
    tree.insert("team", "value3".to_string());
    tree.insert("toast", "value4".to_string());

    println!("Search 'test': {:?}", tree.search("test"));
    println!("Search 'testing': {:?}", tree.search("testing"));
    println!("Search 'team': {:?}", tree.search("team"));

    println!("\nKeys starting with 'te':");
    for key in tree.starts_with("te") {
        println!("  {}", key);
    }

    println!("\n== IP Routing Table ==");
}

```

```

let mut routing = RoutingTable::new();

routing.add_route("192.168.1.0", "gateway1");
routing.add_route("192.168.2.0", "gateway2");
routing.add_route("192.168.1.100", "gateway3");

println!("Lookup 192.168.1.0: {:?}", routing.lookup("192.168.1.0"));
println!("Lookup 192.168.1.100: {:?}", routing.lookup("192.168.1.100"));

println!("\nRoutes for '192.168.1':");
for route in routing.routes_for_prefix("192.168.1") {
    println!("  {}", route);
}
}

```

Radix Tree Benefits: - More space-efficient than trie (compressed edges) - Fewer nodes for strings with long common prefixes - Used in: routing tables, memory allocators, file systems - Trade-off: more complex implementation

Pattern 5: Lock-Free Data Structures

Problem: Mutex-based data structures serialize all access—threads wait even when operating on different elements. Lock contention causes 80% of multi-threaded time spent waiting.

Solution: Use atomic operations (`AtomicUsize`, `AtomicBool`, etc.) for lock-free primitives. Implement lock-free algorithms with compare-and-swap (CAS) loops.

Why It Matters: Lock-free structures enable true parallelism. Multi-threaded counter with Mutex: serialized updates = 1 core performance.

Use Cases: MPMC queues (work-stealing schedulers, actor systems), atomic counters (metrics, rate limiting), lock-free stacks (memory allocators), concurrent hash maps (caches, indexes), real-time systems (audio, trading), high-throughput servers.

Example: Lock-Free Stack

A thread-safe stack without using mutexes, allowing multiple threads to push/pop concurrently.

```

use std::sync::atomic::{AtomicPtr, Ordering};
use std::ptr;
use std::sync::Arc;
use std::thread;

struct Node<T> {
    data: T,
    next: *mut Node<T>,
}

struct LockFreeStack<T> {
    head: AtomicPtr<Node<T>>,
}

```

```
}
```

```
impl<T> LockFreeStack<T> {
    fn new() -> Self {
        Self {
            head: AtomicPtr::new(ptr::null_mut()),
        }
    }

    fn push(&self, data: T) {
        let new_node = Box::into_raw(Box::new(Node {
            data,
            next: ptr::null_mut(),
        }));
        loop {
            let head = self.head.load(Ordering::Acquire);
            unsafe {
                (*new_node).next = head;
            }

            // Try to swap: if head unchanged, install new_node
            if self
                .head
                .compare_exchange(head, new_node, Ordering::Release, Ordering::Acquire)
                .is_ok()
            {
                break;
            }
        }
    }

    fn pop(&self) -> Option<T> {
        loop {
            let head = self.head.load(Ordering::Acquire);

            if head.is_null() {
                return None;
            }

            unsafe {
                let next = (*head).next;

                // Try to swap head with next
                if self
                    .head
                    .compare_exchange(head, next, Ordering::Release, Ordering::Acquire)
                    .is_ok()
                {
                    let data = ptr::read(&(*head).data);
                    // Note: In production, use proper memory reclamation (epoch-based)
                    // Deallocating here can cause use-after-free in concurrent scenarios
                    // drop(Box::from_raw(head)); // Commented out for safety
                }
            }
        }
    }
}
```

```

        return Some(data);
    }
}
}

fn is_empty(&self) -> bool {
    self.head.load(Ordering::Acquire).is_null()
}

unsafe impl<T: Send> Send for LockFreeStack<T> {}
unsafe impl<T: Send> Sync for LockFreeStack<T> {}

//=====
// Real-world: Thread-safe work stealing
//=====

struct WorkStealingQueue<T> {
    stack: Arc<LockFreeStack<T>>,
}

impl<T: Send + 'static> WorkStealingQueue<T> {
    fn new() -> Self {
        Self {
            stack: Arc::new(LockFreeStack::new()),
        }
    }

    fn push(&self, item: T) {
        self.stack.push(item);
    }

    fn steal(&self) -> Option<T> {
        self.stack.pop()
    }

    fn clone_handle(&self) -> Self {
        Self {
            stack: Arc::clone(&self.stack),
        }
    }
}

fn main() {
    println!("==== Lock-Free Stack ===\n");

    let stack = Arc::new(LockFreeStack::new());

    // Spawn multiple threads pushing concurrently
    let mut handles = vec![];

    for thread_id in 0..4 {
        let stack_clone = Arc::clone(&stack);

```

```

handles.push(thread::spawn(move || {
    for i in 0..100 {
        stack_clone.push(thread_id * 1000 + i);
    }
}));


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

// Pop all elements
let mut count = 0;
while stack.pop().is_some() {
    count += 1;
}

println!("Total items pushed and popped: {}", count);

println!("\n==== Work Stealing ===\n");

let queue = WorkStealingQueue::new();

// Producer thread
let producer_queue = queue.clone_handle();
let producer = thread::spawn(move || {
    for i in 0..1000 {
        producer_queue.push(i);
    }
});

// Consumer threads
let mut consumers = vec![];
for _ in 0..3 {
    let consumer_queue = queue.clone_handle();
    consumers.push(thread::spawn(move || {
        let mut stolen = 0;
        while let Some(_) = consumer_queue.steal() {
            stolen += 1;
        }
        stolen
    }));
}

producer.join().unwrap();

let mut total_stolen = 0;
for consumer in consumers {
    total_stolen += consumer.join().unwrap();
}

```

```

    println!("Total items stolen: {}", total_stolen);
}

```

Lock-Free Principles: - **Compare-and-Swap (CAS)**: Atomic operation for lock-free algorithms - **ABA Problem**: Must be handled with epoch-based reclamation - **Memory Ordering**: Acquire/Release semantics for correct synchronization - **Progress Guarantee**: At least one thread makes progress

Example: Lock-Free Queue with Crossbeam

Production-ready lock-free MPMC (Multi-Producer Multi-Consumer) queue.

```

//=====
// Note: Add `crossbeam = "0.8"` to Cargo.toml
//=====

use crossbeam::queue::{ArrayQueue, SegQueue};
use std::sync::Arc;
use std::thread;
use std::time::{Duration, Instant};

//=====
// Bounded MPMC queue
//=====

struct BoundedWorkQueue<T> {
    queue: Arc<ArrayQueue<T>>,
}

impl<T> BoundedWorkQueue<T> {
    fn new(capacity: usize) -> Self {
        Self {
            queue: Arc::new(ArrayQueue::new(capacity)),
        }
    }

    fn push(&self, item: T) -> Result<(), T> {
        self.queue.push(item)
    }

    fn pop(&self) -> Option<T> {
        self.queue.pop()
    }

    fn len(&self) -> usize {
        self.queue.len()
    }

    fn is_full(&self) -> bool {
        self.queue.is_full()
    }

    fn clone_handle(&self) -> Self {
        Self {
            queue: Arc::clone(&self.queue),
        }
    }
}

```

```

        }
    }

//=====
// Unbounded MPMC queue
//=====

struct UnboundedWorkQueue<T> {
    queue: Arc<SegQueue<T>>,
}

impl<T> UnboundedWorkQueue<T> {
    fn new() -> Self {
        Self {
            queue: Arc::new(SegQueue::new()),
        }
    }

    fn push(&self, item: T) {
        self.queue.push(item);
    }

    fn pop(&self) -> Option<T> {
        self.queue.pop()
    }

    fn is_empty(&self) -> bool {
        self.queue.is_empty()
    }

    fn clone_handle(&self) -> Self {
        Self {
            queue: Arc::clone(&self.queue),
        }
    }
}

//=====
// Real-world: Thread pool with lock-free task queue
//=====

struct ThreadPool {
    task_queue: UnboundedWorkQueue<Box<dyn FnOnce() + Send + 'static>>,
    workers: Vec<thread::JoinHandle<()>>,
    shutdown: Arc<std::sync::atomic::AtomicBool>,
}

impl ThreadPool {
    fn new(num_threads: usize) -> Self {
        let task_queue = UnboundedWorkQueue::new();
        let shutdown = Arc::new(std::sync::atomic::AtomicBool::new(false));
        let mut workers = Vec::new();

        for id in 0..num_threads {
    
```

```

    let queue_clone = task_queue.clone_handle();
    let shutdown_clone = Arc::clone(&shutdown);

    workers.push(thread::spawn(move || {
        while !shutdown_clone.load(std::sync::atomic::Ordering::Acquire) {
            if let Some(task) = queue_clone.pop() {
                task();
            } else {
                thread::sleep(Duration::from_millis(100));
            }
        }
    }));
}

Self {
    task_queue,
    workers,
    shutdown,
}
}

fn execute<F>(&self, task: F)
where
    F: FnOnce() + Send + 'static,
{
    self.task_queue.push(Box::new(task));
}

fn shutdown(self) {
    self.shutdown
        .store(true, std::sync::atomic::Ordering::Release);

    for worker in self.workers {
        worker.join().unwrap();
    }
}
}

//=====
// Benchmark: Lock-free vs Mutex
//=====

use std::sync::Mutex;

fn benchmark_lockfree_vs_mutex() {
    const ITEMS: usize = 100_000;
    const THREADS: usize = 4;

    // Lock-free queue
    println!("Lock-free queue:");
    let start = Instant::now();
    let lockfree_queue = UnboundedWorkQueue::new();

    let mut producers = vec![];

```

```

for _ in 0..THREADS {
    let queue = lockfree_queue.clone_handle();
    producers.push(thread::spawn(move || {
        for i in 0..ITEMS {
            queue.push(i);
        }
    }));
}

let mut consumers = vec![];
for _ in 0..THREADS {
    let queue = lockfree_queue.clone_handle();
    consumers.push(thread::spawn(move || {
        let mut count = 0;
        loop {
            if queue.pop().is_some() {
                count += 1;
                if count >= ITEMS {
                    break;
                }
            }
        }
    }));
}

for p in producers {
    p.join().unwrap();
}
for c in consumers {
    c.join().unwrap();
}

let lockfree_time = start.elapsed();
println!(" Time: {:+?}", lockfree_time);

// Mutex-based queue
println!("\nMutex-based queue:");
let start = Instant::now();
let mutex_queue = Arc::new(Mutex::new(std::collections::VecDeque::new()));

let mut producers = vec![];
for _ in 0..THREADS {
    let queue = Arc::clone(&mutex_queue);
    producers.push(thread::spawn(move || {
        for i in 0..ITEMS {
            queue.lock().unwrap().push_back(i);
        }
    }));
}

let mut consumers = vec![];
for _ in 0..THREADS {
    let queue = Arc::clone(&mutex_queue);
}

```

```

consumers.push(thread::spawn(move || {
    let mut count = 0;
    loop {
        if queue.lock().unwrap().pop_front().is_some() {
            count += 1;
            if count >= ITEMS {
                break;
            }
        }
    }
}));

for p in producers {
    p.join().unwrap();
}
for c in consumers {
    c.join().unwrap();
}

let mutex_time = start.elapsed();
println!(" Time: {:+?}", mutex_time);

println!(
    "\nSpeedup: {:.2}x",
    mutex_time.as_secs_f64() / lockfree_time.as_secs_f64()
);
}

fn main() {
    println!("== Lock-Free Queue ==\n");

    let queue = UnboundedWorkQueue::new();

    // Producer thread
    let producer = queue.clone_handle();
    let p = thread::spawn(move || {
        for i in 0..1000 {
            producer.push(i);
        }
    });

    // Consumer threads
    let mut consumers = vec![];
    for _ in 0..3 {
        let consumer = queue.clone_handle();
        consumers.push(thread::spawn(move || {
            let mut sum = 0;
            while let Some(val) = consumer.pop() {
                sum += val;
            }
            sum
        }));
    }
}

```

```

}

p.join().unwrap();

let total: i32 = consumers.into_iter().map(|h| h.join().unwrap()).sum();
println!("Total consumed: {}", total);

println!("\n==== Thread Pool ===\n");

let pool = ThreadPool::new(4);

for i in 0..10 {
    pool.execute(move || {
        println!("Task {} executing", i);
        thread::sleep(Duration::from_millis(100));
    });
}

thread::sleep(Duration::from_secs(2));
pool.shutdown();

println!("\n==== Performance Benchmark ===\n");
benchmark_lockfree_vs_mutex();
}

```

Lock-Free Queue Benefits: - **No blocking:** Threads never wait for locks - **Better scalability:** Performance scales with cores - **Progress guarantee:** System-wide progress even if threads are paused - **2-10x faster** than mutex-based queues under contention

Crossbeam Features: - **ArrayQueue:** Bounded MPMC, faster for fixed capacity - **SegQueue:** Unbounded MPMC, grows dynamically - Epoch-based memory reclamation (solves ABA problem)

Summary

This chapter covered advanced collection types:

1. **VecDeque:** O(1) push/pop at both ends, ring buffers, sliding windows
2. **BinaryHeap:** Priority queues, task scheduling, top-k problems, median tracking
3. **Graphs:** Weighted edges, Dijkstra's algorithm, topological sort, dependency resolution
4. **Tries:** Autocomplete, prefix search, dictionary operations
5. **Radix Trees:** Compressed tries, IP routing, space-efficient string storage
6. **Lock-Free Structures:** CAS-based stack, MPMC queues, thread pools without locks

Key Takeaways: - Choose the right collection for your access pattern - VecDeque is ideal for queues and sliding windows - BinaryHeap provides efficient priority-based access - Graph representation affects algorithm performance - Tries excel at prefix operations - Lock-free structures enable high-concurrency scenarios

Performance Guidelines: - VecDeque: O(1) amortized for push/pop at ends - BinaryHeap: O(log n) insert/remove, O(1) peek - Trie: O(m) operations where m = key length - Lock-free: No blocking, better scalability under contention

Threading Patterns

This chapter explores concurrent programming patterns in Rust using threads. We'll cover thread lifecycle management, parallel work distribution, message passing, shared state synchronization, and coordination primitives through practical, production-ready examples.

Pattern 1: Thread Spawn and Join Patterns

This pattern is the foundation of multi-threaded programming in Rust. It covers how to create threads, transfer data to them, and get results back safely.

- **Problem:** - **Underutilized CPU Cores:** A single-threaded program can't take advantage of multi-core processors, leaving expensive hardware idle. - **Data Ownership:** Moving data into a thread is tricky due to Rust's ownership rules.
- **Solution:** - **thread::spawn with move Closures:** Use `thread::spawn` to create a new thread. The `move` keyword before the closure forces it to take ownership of the variables it captures, safely transferring them to the new thread.
- **Why It Matters:** - **True Parallelism:** Threads allow your program to execute multiple operations at the same time on different CPU cores, dramatically improving performance for CPU-bound tasks. - **Compile-Time Safety:** Rust's ownership model prevents data races at compile time, one of the most common and difficult types of concurrency bugs.
- **Use Cases:**
 - **Parallel Computation:** Performing expensive calculations like image processing, simulations, or scientific computing.
 - **Background Tasks:** Running tasks like logging, metrics collection, or periodic cleanup without blocking the main application.
 - **Concurrent I/O:** Handling multiple network requests or file operations simultaneously.
 - **Responsive UI:** Offloading long-running tasks from the main UI thread to keep an application responsive.

Example: Spawning a Thread with Owned Data

To move data into a thread, use a `move` closure. This transfers ownership of the `data` vector from the parent thread to the new thread. The parent can no longer access it, preventing data races.

`thread::spawn` returns a `JoinHandle`, which we use to wait for the thread to finish and get its result.

```
use std::thread;

fn spawn_with_owned_data() {
    let data = vec![1, 2, 3, 4, 5];

    // The 'move' keyword transfers ownership of 'data' to the new thread.
    let handle = thread::spawn(move || {
        let sum: i32 = data.iter().sum();
        println!("Sum calculated by thread: {}", sum);
    });
}
```

```

        sum // The thread returns the sum.
    });

// The join() method waits for the thread to finish and returns a Result.
let result = handle.join().unwrap();
println!("Result received from thread: {}", result);

// This would fail to compile, as 'data' has been moved:
// println!("Data in main thread: {:?}", data);
}

```

Example: Parallel Computations with Multiple Threads

You can spawn multiple threads to perform different computations in parallel. If they need to work on the same initial data, you must clone it to give each thread its own owned copy.

```

use std::thread;

fn parallel_computations() {
    let numbers = vec![1, 2, 3, 4, 5];

    // Clone data for the first thread.
    let numbers_clone1 = numbers.clone();
    let sum_handle = thread::spawn(move || {
        numbers_clone1.iter().sum::<i32>()
    });

    // Clone data for the second thread.
    let numbers_clone2 = numbers.clone();
    let product_handle = thread::spawn(move || {
        numbers_clone2.iter().product::<i32>()
    });

    // Wait for both threads to complete and collect their results.
    let sum = sum_handle.join().unwrap();
    let product = product_handle.join().unwrap();

    println!("Original data: {:?}", numbers);
    println!("Parallel Sum: {}, Parallel Product: {}", sum, product);
}

```

Example: Handling Errors and Panics in Threads

The `join()` method returns a `Result`. An `Err` indicates the thread panicked. If the thread completes successfully, its own return value might also be a `Result`, so you often need to handle a nested `Result`.

```

use std::thread;

fn thread_with_error_handling() {
    let handle = thread::spawn(|| {
        // Simulate a computation that might fail.
    })
}

```

```

    if rand::random::<bool>() {
        Ok(42)
    } else {
        Err("Computation failed in thread!")
    }
};

match handle.join() {
    Ok(Ok(value)) => println!("Thread completed successfully with value: {}", value),
    Ok(Err(e)) => println!("Thread returned a business logic error: {}", e),
    Err(_) => println!("Thread panicked!"),
}
}

```

Example: Naming Threads for Better Debugging

For easier debugging, especially when you have many threads, you can give them names using the `thread::Builder`. The thread name will appear in panic messages and profiling tools.

```

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

fn named_threads() {
    let handles: Vec<_> = (0..3)
        .map(|i| {
            thread::Builder::new()
                .name(format!("worker-{}", i))
                .spawn(move || {
                    println!("Thread '{}' starting",
                        thread::current().name().unwrap_or("unnamed"));
                    thread::sleep(Duration::from_millis(100));
                    println!("Thread '{}' finished", i);
                    i * 2
                })
                .unwrap()
        })
        .collect();

    let results: Vec<i32> = handles
        .into_iter()
        .map(|h| h.join().unwrap())
        .collect();

    println!("Results from named threads: {:?}", results);
}

```

Example: Parallel File Processing

A real-world example of using threads to process multiple files in parallel. Each thread receives a file path, processes the file, and returns a result structure.

```
use std::thread;
use std::fs;
use std::path::PathBuf;

#[derive(Debug)]
struct ProcessResult {
    path: PathBuf,
    line_count: usize,
    word_count: usize,
    byte_count: usize,
    error: Option<String>,
}

fn process_file(path: &PathBuf) -> ProcessResult {
    match fs::read_to_string(path) {
        Ok(content) => ProcessResult {
            path: path.clone(),
            line_count: content.lines().count(),
            word_count: content.split_whitespace().count(),
            byte_count: content.len(),
            error: None,
        },
        Err(e) => ProcessResult {
            path: path.clone(),
            line_count: 0,
            word_count: 0,
            byte_count: 0,
            error: Some(e.to_string()),
        },
    }
}

fn process_files_parallel(paths: Vec<PathBuf>) -> Vec<ProcessResult> {
    let handles: Vec<_> = paths
        .into_iter()
        .map(|path| {
            thread::spawn(move || {
                process_file(&path)
            })
        })
        .collect();

    handles
        .into_iter()
        .map(|h| h.join().unwrap())
        .collect()
}
```

Example: Borrowing Stack Data with Scoped Threads

A regular `thread::spawn` requires the closure to have a '`static`' lifetime, meaning it cannot borrow data from the parent thread's stack. `thread::scope` solves this by guaranteeing that all threads spawned within the scope will finish before the scope ends, making it safe to borrow.

```
use std::thread;

fn scoped_threads_for_borrowing() {
    let mut data = vec![1, 2, 3, 4, 5];

    // 'thread::scope' creates a scope for spawning threads.
    // The scope guarantees that all threads within it will join before it exits.
    thread::scope(|s| {
        // This thread borrows 'data' immutably.
        s.spawn(|| {
            println!("Scoped thread sees sum: {}", data.iter().sum::<i32>());
        });

        // This thread also borrows 'data' immutably.
        s.spawn(|| {
            println!("Scoped thread sees product: {}", data.iter().product::<i32>());
        });
    }); // The scope blocks here until all spawned threads complete.

    // After the scope, we can mutate 'data' again.
    data.push(6);
    println!("After scope, data is: {:?}", data);
}
```

Pattern 2: Thread Pools and Work Stealing

Spawning a new thread for every small task is inefficient. Thread pools and work-stealing are advanced patterns for managing a fixed set of worker threads to execute many tasks efficiently.

- **Problem:** - **High Overhead:** Spawning thousands of OS threads for thousands of small tasks is slow and wastes memory. - **Resource Exhaustion:** An unbounded number of threads can exhaust system resources, leading to thrashing as the OS constantly switches between them.
- **Solution:** - **Thread Pools:** Create a fixed number of worker threads (often equal to the number of CPU cores) and reuse them for multiple tasks. Tasks are submitted to a shared queue, and idle workers pull tasks from it.
- **Why It Matters:** - **Efficiency:** Thread pools eliminate the overhead of thread creation, making it feasible to parallelize even small tasks. - **Automatic Load Balancing:** Work-stealing schedulers automatically distribute work, ensuring that all CPU cores are kept busy, leading to near-linear performance scaling for many parallel algorithms.
- **Use Cases:**
 - **Data Parallelism:** Processing large collections of data (e.g., arrays, vectors) in parallel.

- **Web Servers:** Handling a large number of incoming requests with a fixed pool of worker threads.
- **Recursive Algorithms:** Parallelizing divide-and-conquer algorithms like quicksort or tree traversals.
- **Batch Processing:** Running a large number of independent jobs, such as image encoding or data analysis tasks.

Example: Data Parallelism with Rayon

Rayon is the de-facto standard for data parallelism in Rust. It provides a `par_iter()` method that turns a sequential iterator into a parallel one, automatically distributing the work across a work-stealing thread pool.

```
// Add `rayon = "1.8"` to Cargo.toml
use rayon::prelude::*;

fn parallel_map_reduce_with_rayon() {
    let numbers: Vec<i32> = (1..=1_000_000).collect();

    // Parallel sum
    let sum: i32 = numbers.par_iter().sum();
    println!("Parallel Sum (Rayon): {}", sum);

    // Parallel map
    let squares: Vec<i32> = numbers
        .par_iter()
        .map(|&x| x * x)
        .collect();
    println!("First 5 squares: {:?}", &squares[..5]);

    // Parallel filter and count
    let even_count = numbers
        .par_iter()
        .filter(|&x| x % 2 == 0)
        .count();
    println!("Number of even numbers: {}", even_count);
}
```

Example: Parallel Sorting with Rayon

Rayon also provides parallel implementations of common algorithms, like sorting. For large datasets, `par_sort()` can be significantly faster than the standard sequential sort.

```
// Add `rayon = "1.8"` to Cargo.toml
use rayon::prelude::*;

fn parallel_sorting_with_rayon() {
    let mut data: Vec<i32> = (0..1_000_000).rev().collect();
    println!("First 10 elements (before sort): {:?}", &data[..10]);
```

```

// Parallel sort is much faster for large collections.
data.par_sort();

println!("First 10 elements (after sort): {:?}", &data[..10]);
}

```

Example: Real-World - Parallel Image Processing

Rayon is perfect for tasks like image processing, where the same operation can be applied to millions of pixels independently. This example shows how to apply filters to an image in parallel.

```

// Add `rayon = "1.8"` to Cargo.toml
use rayon::prelude::*;

struct Image {
    pixels: Vec<u8>,
    width: usize,
    height: usize,
}

impl Image {
    fn apply_filter_parallel(&mut self, filter: impl Fn(u8) -> u8 + Sync) {
        self.pixels.par_iter_mut().for_each(|pixel| {
            *pixel = filter(*pixel);
        });
    }

    fn brighten(&mut self, amount: u8) {
        self.apply_filter_parallel(|p| p.saturating_add(amount));
    }
}

```

Pattern 3: Message Passing with Channels

This pattern focuses on communication between threads by sending messages through channels, avoiding the complexities of shared memory and locks.

- **Problem:** - **Complexity of Locks:** Using `Arc<Mutex<T>>` for every piece of shared data is verbose, and it's easy to make mistakes like holding a lock for too long (hurting performance) or causing deadlocks. - **Race Conditions:** Manually coordinating access to shared data is a major source of bugs that are hard to reproduce and debug.
- **Solution:** - **Channels:** A channel provides a safe way to send data from one or more “producer” threads to one or more “consumer” threads. The channel handles all the necessary synchronization.
- **Why It Matters:** - **Simplicity and Safety:** Channels transform a complex synchronization problem into a simple producer/consumer pattern. The type system ensures that you can't have data races.

- **Use Cases:**

- **Producer-Consumer Pipelines:** A series of threads organized into stages, where each stage processes data and passes it to the next via a channel.
- **Event-Driven Architectures:** A central thread or task that receives events from multiple sources and dispatches them for processing.
- **Actor Systems:** A concurrency model where independent “actors” communicate with each other exclusively by sending messages.
- **Background Workers:** Offloading tasks like sending emails, logging, or processing analytics to a pool of workers that receive jobs via a channel.

Example: Basic Producer-Consumer with MPSC Channel

`std::sync::mpsc` provides a “Multiple Producer, Single Consumer” channel. Here, one producer thread sends data to a single consumer thread.

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

fn basic_mpsc_channel() {
    let (tx, rx) = mpsc::channel(); // tx = transmitter, rx = receiver

    // Spawn a producer thread.
    thread::spawn(move || {
        for i in 0..5 {
            println!("Sending: {}", i);
            tx.send(i).unwrap();
            thread::sleep(Duration::from_millis(100));
        }
    });

    // The receiver can be used as an iterator that blocks until a message is received.
    for received in rx {
        println!("Received: {}", received);
    }
}
```

Example: Multiple Producers

The transmitter (`tx`) can be cloned to allow multiple threads to send messages to the same receiver.

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

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

    for thread_id in 0..3 {
        // Clone the transmitter for each new thread.
    }
}
```

```

let tx_clone = tx.clone();
thread::spawn(move || {
    for i in 0..3 {
        let message = format!("Thread {} sends message {}", thread_id, i);
        tx_clone.send(message).unwrap();
    }
});
}

// Drop the original transmitter so the receiver knows when to stop waiting.
drop(tx);

// The receiver will automatically close when all transmitters have been dropped.
for received in rx {
    println!("Received: {}", received);
}
}

```

Example: Crossbeam Bounded Channel for Backpressure

The `crossbeam` crate offers more powerful channels. A “bounded” channel has a fixed capacity. If a fast producer tries to send to a full channel, it will block until a slow consumer makes space. This is called backpressure.

```

// Add `crossbeam = "0.8"` to Cargo.toml
use crossbeam::channel::bounded;
use std::thread;
use std::time::Duration;

fn bounded_channel_backpressure() {
    // A channel with a capacity of 2.
    let (tx, rx) = bounded(2);

    // A fast producer.
    thread::spawn(move || {
        for i in 0..10 {
            println!("Producer: trying to send {}", i);
            tx.send(i).unwrap(); // This will block if the channel is full.
            println!("Producer: sent {}", i);
        }
    });

    // A slow consumer.
    thread::sleep(Duration::from_secs(1));
    for _ in 0..10 {
        let value = rx.recv().unwrap();
        println!("Consumer: received {}", value);
        thread::sleep(Duration::from_millis(500));
    }
}

```

Example: The Actor Pattern

The actor model is a concurrency pattern where “actors” are isolated entities that communicate exclusively through messages. This example implements a simple actor that maintains an internal state.

```
// Add `crossbeam = "0.8"` to Cargo.toml
use crossbeam::channel::{unbounded, Sender, Receiver, select};
use std::thread;

// Messages the actor can receive.
enum ActorMessage {
    Process(String),
    GetState(Sender<String>), // Message to request the actor's state.
    Shutdown,
}

// The actor itself.
struct Actor {
    inbox: Receiver<ActorMessage>,
    state: String,
}

impl Actor {
    fn new(inbox: Receiver<ActorMessage>) -> Self {
        Self { inbox, state: String::new() }
    }

    // The actor's main loop.
    fn run(&mut self) {
        while let Ok(msg) = self.inbox.recv() {
            match msg {
                ActorMessage::Process(data) => {
                    self.state.push_str(&data);
                    println!("Actor state updated.");
                }
                ActorMessage::GetState(reply_to) => {
                    reply_to.send(self.state.clone()).unwrap();
                }
                ActorMessage::Shutdown => {
                    println!("Actor shutting down.");
                    break;
                }
            }
        }
    }
}

fn actor_pattern_example() {
    let (tx, rx) = unbounded();
    let actor_handle = thread::spawn(move || Actor::new(rx).run());

    // Send messages to the actor.
}
```

```

tx.send(ActorMessage::Process("Hello, ".to_string())).unwrap();
tx.send(ActorMessage::Process("Actor!".to_string())).unwrap();

// Send a message to get the actor's state.
let (reply_tx, reply_rx) = unbounded();
tx.send(ActorMessage::GetState(reply_tx)).unwrap();
let state = reply_rx.recv().unwrap();
println!("Retrieved actor state: '{}'", state);

// Shut down the actor.
tx.send(ActorMessage::Shutdown).unwrap();
actor_handle.join().unwrap();
}

```

Pattern 4: Shared State with Locks (Mutex & RwLock)

While message passing is preferred, sometimes you need multiple threads to access the same piece of data. This pattern uses `Arc`, `Mutex`, and `RwLock` to share memory safely.

- **Problem:** - **Ownership Conflicts:** Rust’s ownership rules prevent you from having multiple mutable references to the same data, which is exactly what you need when threads share state. - **Data Races:** Unsynchronized access to shared data can lead to data races, where the final state depends on the non-deterministic order of thread execution, causing subtle and difficult-to-reproduce bugs.
- **Solution:** - **`Arc<T>` (Atomically Reference-Counted Pointer):** `Arc` is a smart pointer that lets multiple threads have shared ownership of the same data. It keeps a count of active references, and when the last reference is dropped, the data is cleaned up.
- **Why It Matters:** - **Compile-Time Safety:** Rust’s type system ensures you use locks correctly. You cannot access the data without first acquiring the lock, and you cannot forget to release it, preventing entire classes of concurrency bugs.
- **Use Cases:**
 - **Shared Caches:** An in-memory cache that is accessed by multiple worker threads.
 - **Connection Pools:** A pool of database or network connections that are shared among threads in a web server.
 - **Global State/Configuration:** Application-wide configuration that needs to be read by many threads and occasionally updated.
 - **Metrics and Counters:** Collecting metrics (like request counts) from multiple threads into a single shared data structure.

Example: Shared Counter with `Arc<Mutex<T>>`

This is the “Hello, World!” of shared state concurrency. Multiple threads increment a shared counter, using a `Mutex` to ensure that the increments don’t interfere with each other.

```

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

```

```

fn shared_counter() {
    // Arc<Mutex<T>> is the standard way to share mutable state.
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        // Clone the Arc to give each thread a reference to the Mutex.
        let counter_clone = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            // lock() acquires the mutex, blocking until it's available.
            // The returned "lock guard" provides access to the data.
            let mut num = counter_clone.lock().unwrap();
            *num += 1;
            // The lock is automatically released when 'num' goes out of scope.
        });
        handles.push(handle);
    }

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

    println!("Final count: {}", *counter.lock().unwrap());
}

```

Example: Read-Heavy Workloads with RwLock

An **RwLock** is ideal when data is read frequently but written to infrequently. It allows unlimited concurrent readers but only a single writer.

```

use std::sync::{Arc, RwLock};
use std::thread;
use std::time::Duration;

fn rwlock_for_read_heavy_data() {
    let config = Arc::new(RwLock::new("initial_config".to_string()));
    let mut handles = vec![];

    // Spawn multiple reader threads.
    for i in 0..5 {
        let config_clone = Arc::clone(&config);
        let handle = thread::spawn(move || {
            // read() acquires a read lock. Multiple threads can hold a read lock.
            let cfg = config_clone.read().unwrap();
            println!("Reader {}: Current config is '{}'", i, *cfg);
        });
        handles.push(handle);
    }

    // Wait a moment, then spawn a writer thread.
    thread::sleep(Duration::from_millis(10));
}

```

```

let config_clone = Arc::clone(&config);
let writer_handle = thread::spawn(move || {
    // write() acquires a write lock. This will wait until all read locks are released.
    // No new readers can acquire a lock while the writer is waiting.
    let mut cfg = config_clone.write().unwrap();
    *cfg = "updated_config".to_string();
    println!("Writer: Updated config.");
});
handles.push(writer_handle);

for handle in handles {
    handle.join().unwrap();
}
println!("Final config: {}", *config.read().unwrap());
}

```

Pattern 5: Synchronization Primitives (Barrier & Condvar)

Barriers and Condvars are lower-level primitives used to coordinate the timing of thread execution.

- **Problem:** - **Phased Execution:** Some parallel algorithms require all threads to complete a certain phase of work before any thread can move on to the next phase. - **Inefficient Waiting:** A thread might need to wait for a specific condition to become true (e.g., for a queue to become non-empty).
- **Solution:** - **Barrier:** A **Barrier** is created with a count. Threads that reach the barrier will call **.wait()** and block.
- **Why It Matters:** - **Efficiency:** **Condvar** avoids busy-waiting, leading to much better CPU utilization. Threads that are waiting for a condition consume no CPU resources.
- **Use Cases:**
 - **Parallel Simulations:** Using a **Barrier** to ensure all threads have completed a time step before starting the next one.
 - **Producer-Consumer Queues:** Using a **Condvar** to make a consumer wait when the queue is empty and a producer wait when it's full.
 - **Parallel Testing:** Using a **Barrier** to start multiple threads simultaneously to test for race conditions under high contention.
 - **Phased Algorithms:** Any algorithm where work is divided into distinct, sequential phases that can be executed in parallel within each phase.

Example: Barrier for Phased Computation

A **Barrier** is used to synchronize multiple threads at a specific point. All threads must reach the barrier before any of them can proceed.

```

use std::sync::{Arc, Barrier};
use std::thread;

```

```

fn barrier_for_phased_work() {
    let num_threads = 4;
    let barrier = Arc::new(Barrier::new(num_threads));
    let mut handles = vec![];

    for id in 0..num_threads {
        let barrier_clone = Arc::clone(&barrier);
        handles.push(thread::spawn(move || {
            println!("Thread {}: Performing phase 1", id);
            // ... do some work ...
            barrier_clone.wait(); // All threads wait here.

            println!("Thread {}: All threads finished phase 1. Starting phase 2.", id);
            // ... do some work ...
            barrier_clone.wait(); // Wait again.

            println!("Thread {}: All threads finished phase 2.", id);
        }));
    }

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

```

Example: Condvar for a Bounded Queue

This example shows how to use a `Condvar` and a `Mutex` to build a thread-safe bounded queue. Producers wait if the queue is full, and consumers wait if it's empty.

```

use std::sync::{Arc, Mutex, Condvar};
use std::collections::VecDeque;
use std::thread;

struct BoundedQueue<T> {
    queue: Mutex<VecDeque<T>>,
    condvar: Condvar,
    capacity: usize,
}

impl<T> BoundedQueue<T> {
    fn new(capacity: usize) -> Self {
        Self {
            queue: Mutex::new(VecDeque::with_capacity(capacity)),
            condvar: Condvar::new(),
            capacity,
        }
    }

    fn push(&self, item: T) {
        let mut queue = self.queue.lock().unwrap();
        // Wait while the queue is full.
    }
}

```

```

        while queue.len() >= self.capacity {
            queue = self.condvar.wait(queue).unwrap();
        }
        queue.push_back(item);
        // Notify one waiting consumer that there's new data.
        self.condvar.notify_one();
    }

fn pop(&self) -> T {
    let mut queue = self.queue.lock().unwrap();
    // Wait while the queue is empty.
    while queue.is_empty() {
        queue = self.condvar.wait(queue).unwrap();
    }
    let item = queue.pop_front().unwrap();
    // Notify one waiting producer that there's new space.
    self.condvar.notify_one();
    item
}

fn condvar_for_queue() {
    let queue = Arc::new(BoundedQueue::new(3));

    // Producer thread
    let queue_clone_p = Arc::clone(&queue);
    let producer = thread::spawn(move || {
        for i in 0..10 {
            println!("Producer: pushing {}", i);
            queue_clone_p.push(i);
        }
    });

    // Consumer thread
    let queue_clone_c = Arc::clone(&queue);
    let consumer = thread::spawn(move || {
        for _ in 0..10 {
            let item = queue_clone_c.pop();
            println!("Consumer: popped {}", item);
        }
    });

    producer.join().unwrap();
    consumer.join().unwrap();
}

```

Async Runtime Patterns

This chapter explores asynchronous programming patterns in Rust using `async/await` and `async` runtimes. We'll cover future composition, stream processing, concurrency patterns, timeout handling, and runtime comparisons through practical, production-ready examples.

Pattern 1: Future Composition

Problem: Chaining async operations with nested `.await` calls creates deeply nested code (callback hell). Running multiple async operations concurrently with manual spawning is verbose.

Solution: Use future combinators: `map()`, `and_then()`, `or_else()` for chaining transformations. Use `join!` and `try_join!` to run futures concurrently, waiting for all.

Why It Matters: Proper composition determines performance and readability. Sequential `.await` on 3 independent operations takes 300ms; concurrent `join!` takes 100ms—3x faster.

Use Cases: HTTP request batching (parallel API calls), database query composition (dependent queries), microservice orchestration, retry logic with fallbacks, concurrent file operations, fan-out/fan-in patterns.

Example: Future Combinators and Error Handling

Compose multiple async operations, handle errors gracefully, and transform results without nested callbacks.

```
// Note: Add to Cargo.toml:  
// tokio = { version = "1.35", features = ["full"] }  
// reqwest = "0.11"  
// serde = { version = "1.0", features = ["derive"] }  
// serde_json = "1.0"  
  
use tokio;  
use std::time::Duration;
```

Example: Basic future composition with map

This example walks through the basics of future composition with `map`, highlighting each step so you can reuse the pattern.

```
async fn fetch_user_name(user_id: u64) -> Result<String, String> {  
    // Simulate API call  
    tokio::time::sleep(Duration::from_millis(100)).await;  
  
    if user_id == 0 {  
        Err("Invalid user ID".to_string())  
    } else {  
        Ok(format!("User {}", user_id))  
    }  
}  
  
async fn get_user_name_uppercase(user_id: u64) -> Result<String, String> {  
    // Map over the result  
    fetch_user_name(user_id)  
        .await
```

```
.map(|name| name.to_uppercase()))
}
```

Example: Chaining async operations

This example shows chaining async operations to illustrate where the pattern fits best.

```
async fn fetch_user_posts(user_id: u64) -> Result<Vec<String>, String> {
    tokio::time::sleep(Duration::from_millis(100)).await;
    Ok(vec![
        format!("Post 1 by user {}", user_id),
        format!("Post 2 by user {}", user_id),
    ])
}

async fn get_user_with_posts(user_id: u64) -> Result<(String, Vec<String>), String> {
    let name = fetch_user_name(user_id).await?;
    let posts = fetch_user_posts(user_id).await?;
    Ok((name, posts))
}
```

Example: Error conversion and propagation

This example shows how to error conversion and propagation in practice, emphasizing why it works.

```
#[derive(Debug)]
enum AppError {
    Network(String),
    NotFound,
    InvalidData(String),
}

impl From<reqwest::Error> for AppError {
    fn from(err: reqwest::Error) -> Self {
        AppError::Network(err.to_string())
    }
}

async fn fetch_json_data(url: &str) -> Result<serde_json::Value, AppError> {
    let response = reqwest::get(url).await?;

    if !response.status().is_success() {
        return Err(AppError::NotFound);
    }

    let data = response.json().await?;
    Ok(data)
}
```

Example: Combining multiple futures with different error types

This example shows combining multiple futures with different error types to illustrate where the pattern fits best.

```
use futures::future::TryFutureExt;

async fn complex_operation() -> Result<String, AppError> {
    let data1 = fetch_json_data("https://api.example.com/data1")
        .await?;

    let data2 = fetch_json_data("https://api.example.com/data2")
        .await?;

    // Process both results
    Ok(format!("Combined: {:?} and {:?}", data1, data2))
}

// Real-world: HTTP client with retries
async fn fetch_with_retry<F, Fut, T, E>(
    mut f: F,
    max_retries: usize,
) -> Result<T, E>
where
    F: FnMut() -> Fut,
    Fut: std::future::Future<Output = Result<T, E>>,
    E: std::fmt::Display,
{
    let mut attempts = 0;

    loop {
        match f().await {
            Ok(result) => return Ok(result),
            Err(e) => {
                attempts += 1;
                if attempts >= max_retries {
                    return Err(e);
                }
                println!("Attempt {} failed: {}. Retrying...", attempts, e);
                tokio::time::sleep(Duration::from_secs(2u64.pow(attempts as u32))).await;
            }
        }
    }
}

async fn fetch_data_with_retry(url: String) -> Result<String, reqwest::Error> {
    fetch_with_retry()
        || async {
            reqwest::get(&url)
                .await?
                .text()
                .await
        }
}
```

```

        },
        3,
    )
    .await
}

#[tokio::main]
async fn main() {
    println!("==== Future Composition ====\n");

    match get_user_name_uppercase(42).await {
        Ok(name) => println!("User name: {}", name),
        Err(e) => println!("Error: {}", e),
    }

    match get_user_with_posts(42).await {
        Ok((name, posts)) => {
            println!("User: {}", name);
            println!("Posts: {:?}", posts);
        }
        Err(e) => println!("Error: {}", e),
    }
}

```

Future Composition Patterns: - **map**: Transform success value - **and_then**: Chain dependent operations - **or_else**: Handle errors and recover - **? operator**: Early return on error

Example: Concurrent Future Execution

Execute multiple independent futures concurrently to improve throughput.

```

use tokio;
use std::time::Duration;

```

Example: join! - wait for all futures

This example shows how to use join! to wait for all futures without over-synchronizing.

```

async fn concurrent_fetch() {
    let (result1, result2, result3) = tokio::join!(
        fetch_user_name(1),
        fetch_user_name(2),
        fetch_user_name(3),
    );

    println!("Results: {:?}, {:?}, {:?}", result1, result2, result3);
}

```

Example: try_join! - wait for all, fail fast on error

This example shows how to use try_join! to wait for all, fail fast on error without over-synchronizing.

```
async fn concurrent_fetch_fail_fast() -> Result<(String, String, String), String> {
    tokio::try_join!(
        fetch_user_name(1),
        fetch_user_name(2),
        fetch_user_name(3),
    )
}
```

Example: select! - race futures, take first to complete

This example shows how to use select! to race futures, take first to complete without over-synchronizing.

```
use tokio::time::sleep;

async fn race_requests() -> String {
    tokio::select! {
        result = fetch_user_name(1) => {
            format!("First: {:?}", result)
        }
        result = fetch_user_name(2) => {
            format!("Second: {:?}", result)
        }
        _ = sleep(Duration::from_secs(1)) => {
            "Timeout".to_string()
        }
    }
}
```

Example: Dynamic number of futures with FuturesUnordered

This example shows how to dynamic number of futures with FuturesUnordered in practice, emphasizing why it works.

```
use futures::stream::{FuturesUnordered, StreamExt};

async fn fetch_all_users(user_ids: Vec<u64>) -> Vec<Result<String, String>> {
    let futures: FuturesUnordered<_> = user_ids
        .into_iter()
        .map(|id| fetch_user_name(id))
        .collect();

    futures.collect().await
}
```

```

}

// Real-world: Parallel HTTP requests with limit
use futures::stream::FuturesOrdered;

async fn fetch_urls_concurrently(urls: Vec<String>, max_concurrent: usize) ->
    Vec<Result<String, reqwest::Error>> {
    let mut results = Vec::new();

    for chunk in urls.chunks(max_concurrent) {
        let futures: Vec<_> = chunk
            .iter()
            .map(|url| async move {
                reqwest::get(url)
                    .await?
                    .text()
                    .await
            })
            .collect();

        let chunk_results = futures::future::join_all(futures).await;
        results.extend(chunk_results);
    }

    results
}

// Real-world: Timeout wrapper
async fn with_timeout<F, T>(
    future: F,
    duration: Duration,
) -> Result<T, tokio::time::error::Elapsed>
where
    F: std::future::Future<Output = T>,
{
    tokio::time::timeout(duration, future).await
}

// Real-world: Cancellation-safe operations
async fn cancellation_safe_write(data: String) -> Result<(), std::io::Error> {
    use tokio::fs::File;
    use tokio::io::AsyncWriteExt;

    let mut file = File::create("output.txt").await?;

    // Write atomically - either all or nothing
    file.write_all(data.as_bytes()).await?;
    file.sync_all().await?;

    Ok(())
}

#[tokio::main]
async fn main() {
    println!("==> Concurrent Execution ==\n");
}

```

```

concurrent_fetch().await;

println!("==== Fail Fast ====\n");
match concurrent_fetch_fail_fast().await {
    Ok(results) => println!("All succeeded: {:?}", results),
    Err(e) => println!("One failed: {}", e),
}

println!("==== Race ====\n");
let winner = race_requests().await;
println!("Winner: {}", winner);

println!("==== Dynamic Futures ====\n");
let results = fetch_all_users(vec![1, 2, 3, 4, 5]).await;
println!("Fetched {} users", results.len());

println!("==== Timeout ====\n");
match with_timeout(
    fetch_user_name(1),
    Duration::from_millis(50),
).await {
    Ok(name) => println!("Got name: {:?}", name),
    Err(_) => println!("Timed out"),
}
}

```

Concurrent Patterns: - **join!**: All complete, collect all results - **try_join!**: All complete or fail fast - **select!**: First to complete wins - **FuturesUnordered**: Dynamic collection, unordered completion - **join_all**: Dynamic collection, ordered results

Pattern 2: Stream Processing

Problem: Processing infinite or unbounded sequences (websocket messages, sensor data, log streams) with standard iterators blocks thread. Collecting entire stream into Vec before processing wastes memory for large datasets.

Solution: Use **Stream** trait (async iterator) to yield values over time without blocking. Apply stream combinators: **.map()**, **.filter()**, **.fold()**, **.buffered()** for transformations.

Why It Matters: Streams enable processing data larger than memory—GB log file analyzed in constant memory. WebSocket connections handle millions of messages without collecting all.

Use Cases: WebSocket message processing, sensor data aggregation, log file streaming, database query result streaming, event sourcing, pub-sub systems, real-time analytics, infinite data sources.

Example: Stream Combinators

Process async sequences of data with transformations, filtering, and aggregation.

```
use tokio;
use tokio_stream::{self as stream, StreamExt};
use std::time::Duration;
```

Example: Creating streams

This example shows creating streams to illustrate where the pattern fits best.

```
async fn create_streams() {
    // From iterator
    let s = stream::iter(vec![1, 2, 3, 4, 5]);

    // From channel
    let (tx, rx) = tokio::sync::mpsc::channel(10);
    tokio::spawn(async move {
        for i in 0..5 {
            tx.send(i).await.unwrap();
        }
    });
    let s = tokio_stream::wrappers::ReceiverStream::new(rx);

    // Interval stream
    let s = stream::StreamExt::take(
        tokio_stream::wrappers::IntervalStream::new(
            tokio::time::interval(Duration::from_millis(100))
        ),
        5,
    );
}
```

Example: Map and filter

This example shows how to map and filter in practice, emphasizing why it works.

```
async fn transform_stream() {
    let stream = stream::iter(1..=10)
        .filter(|x| x % 2 == 0)
        .map(|x| x * 2);

    let results: Vec<i32> = stream.collect().await;
    println!("Transformed: {:?}", results);
}
```

Example: Then (async map)

This example shows how to then (async map) in practice, emphasizing why it works.

```

async fn async_transform_stream() {
    let stream = stream::iter(1..=5)
        .then(|x| async move {
            tokio::time::sleep(Duration::from_millis(10)).await;
            x * x
        });

    let results: Vec<i32> = stream.collect().await;
    println!("Async transformed: {:?}", results);
}

```

Example: Fold and reduce

This example shows how to fold and reduce in practice, emphasizing why it works.

```

async fn aggregate_stream() {
    let sum = stream::iter(1..=100)
        .fold(0, |acc, x| acc + x)
        .await;

    println!("Sum: {}", sum);
}

```

Example: Take and skip

This example shows how to take and skip in practice, emphasizing why it works.

```

async fn limit_stream() {
    let results: Vec<i32> = stream::iter(1..=100)
        .skip(10)
        .take(5)
        .collect()
        .await;

    println!("Limited: {:?}", results);
}

// Real-world: Rate limiting
use std::sync::Arc;
use tokio::sync::Semaphore;

async fn rate_limited_requests(urls: Vec<String>) {
    let semaphore = Arc::new(Semaphore::new(5)); // Max 5 concurrent

    let stream = stream::iter(urls)
        .map(|url| {
            let permit = Arc::clone(&semaphore);
            async move {

```

```

        let _permit = permit.acquire().await.unwrap();
        println!("Fetching: {}", url);
        // Simulate request
        tokio::time::sleep(Duration::from_millis(100)).await;
        format!("Response from {}", url)
    }
}

.buffer_unordered(10); // Process up to 10 at once

let results: Vec<String> = stream.collect().await;
println!("Fetched {} URLs", results.len());
}

// Real-world: Batch processing
async fn batch_process<T>(items: Vec<T>, batch_size: usize)
where
    T: Send + 'static,
{
    use futures::stream;

    let batches = items.chunks(batch_size);

    for (i, batch) in batches.enumerate() {
        println!("Processing batch {}: {} items", i, batch.len());
        // Process batch
        tokio::time::sleep(Duration::from_millis(50)).await;
    }
}

// Real-world: Stream merging
async fn merge_streams() {
    use tokio_stream::StreamExt;

    let stream1 = stream::iter(vec![1, 2, 3]);
    let stream2 = stream::iter(vec![4, 5, 6]);

    let merged = stream::StreamExt::merge(stream1, stream2);
    let results: Vec<i32> = merged.collect().await;
    println!("Merged: {:?}", results);
}

#[tokio::main]
async fn main() {
    println!("== Transform Stream ==\n");
    transform_stream().await;

    println!("\n== Async Transform ==\n");
    async_transform_stream().await;

    println!("\n== Aggregate ==\n");
    aggregate_stream().await;

    println!("\n== Limit ==\n");
    limit_stream().await;
}

```

```

    println!("==== Rate Limiting ===");
    let urls: Vec<_> = (0..20).map(|i| format!("https://example.com/{}", i)).collect();
    rate_limited_requests(urls).await;

    println!("==== Merge ===");
    merge_streams().await;
}

```

Stream Combinators: - **map/filter**: Synchronous transformation - **then**: Async transformation - **fold**: Aggregation - **buffer_unordered**: Concurrent processing - **merge**: Combine multiple streams

Example: Stream from Async Generators

Create custom async streams from various sources like WebSockets, file watching, or event sources.

```

use tokio;
use tokio_stream::Stream, StreamExt;
use std::pin::Pin;
use std::task::Context, Poll;

```

Example: Manual stream implementation

This example shows how to manual stream implementation while calling out the practical trade-offs.

```

struct CounterStream {
    count: u32,
    max: u32,
}

impl CounterStream {
    fn new(max: u32) -> Self {
        Self { count: 0, max }
    }
}

impl Stream for CounterStream {
    type Item = u32;

    fn poll_next(mut self: Pin<&mut Self>, _cx: &mut Context<'_>) -> Poll<Option<Self::Item>> {
        if self.count < self.max {
            let current = self.count;
            self.count += 1;
            Poll::Ready(Some(current))
        } else {
            Poll::Ready(None)
        }
    }
}

```

```
    }
}
```

Example: Async generator pattern using channels

This example shows how to async generator pattern using channels in practice, emphasizing why it works.

```
async fn number_generator(max: u32) -> impl Stream<Item = u32> {
    let (tx, rx) = tokio::sync::mpsc::channel(10);

    tokio::spawn(async move {
        for i in 0..max {
            tokio::time::sleep(std::time::Duration::from_millis(10)).await;
            if tx.send(i).await.is_err() {
                break;
            }
        }
    });
}

tokio_stream::wrappers::ReceiverStream::new(rx)
}

// Real-world: File watcher stream
use notify::{Watcher, RecursiveMode, Event};

async fn file_watcher_stream(path: String) -> impl Stream<Item = notify::Result<Event>> {
    let (tx, rx) = tokio::sync::mpsc::channel(100);

    tokio::task::spawn_blocking(move || {
        let (notify_tx, notify_rx) = std::sync::mpsc::channel();

        let mut watcher = notify::recommended_watcher(notify_tx).unwrap();
        watcher.watch(path.as_ref(), RecursiveMode::Recursive).unwrap();

        for event in notify_rx {
            if tx.blocking_send(event).is_err() {
                break;
            }
        }
    });
}

tokio_stream::wrappers::ReceiverStream::new(rx)
}

// Real-world: WebSocket message stream (simulation)
use std::time::Duration;

#[derive(Debug)]
enum WsMessage {
    Text(String),
    Binary(Vec<u8>),
    Ping,
```

```

        Close,
    }

async fn websocket_stream() -> impl Stream<Item = WsMessage> {
    let (tx, rx) = tokio::sync::mpsc::channel(100);

    tokio::spawn(async move {
        let messages = vec![
            WsMessage::Text("Hello".to_string()),
            WsMessage::Text("World".to_string()),
            WsMessage::Ping,
            WsMessage::Binary(vec![1, 2, 3]),
            WsMessage::Close,
        ];

        for msg in messages {
            tokio::time::sleep(Duration::from_millis(100)).await;
            if tx.send(msg).await.is_err() {
                break;
            }
        }
    });
}

tokio_stream::wrappers::ReceiverStream::new(rx)
}

// Real-world: Database query result stream
#[derive(Debug)]
struct Row {
    id: u64,
    data: String,
}

async fn database_query_stream(query: String) -> impl Stream<Item = Row> {
    let (tx, rx) = tokio::sync::mpsc::channel(100);

    tokio::spawn(async move {
        // Simulate database query returning rows
        for i in 0..10 {
            tokio::time::sleep(Duration::from_millis(10)).await;
            let row = Row {
                id: i,
                data: format!("Data {}", i),
            };
            if tx.send(row).await.is_err() {
                break;
            }
        }
    });
}

tokio_stream::wrappers::ReceiverStream::new(rx)
}

```

Example: Interval-based stream

This example shows interval-based stream to illustrate where the pattern fits best.

```
async fn ticker_stream(interval: Duration, count: usize) -> impl Stream<Item = u64> {
    let (tx, rx) = tokio::sync::mpsc::channel(10);

    tokio::spawn(async move {
        let mut interval = tokio::time::interval(interval);
        for i in 0..count {
            interval.tick().await;
            if tx.send(i as u64).await.is_err() {
                break;
            }
        }
    });
}

tokio_stream::wrappers::ReceiverStream::new(rx)
}

#[tokio::main]
async fn main() {
    println!("== Counter Stream ==\n");
    let mut stream = CounterStream::new(5);
    while let Some(n) = stream.next().await {
        println!("Count: {}", n);
    }

    println!("\n== Number Generator ==\n");
    let mut stream = number_generator(5).await;
    while let Some(n) = stream.next().await {
        println!("Generated: {}", n);
    }

    println!("\n== WebSocket Stream ==\n");
    let mut stream = websocket_stream().await;
    while let Some(msg) = stream.next().await {
        println!("Message: {:?}", msg);
    }

    println!("\n== Database Stream ==\n");
    let mut stream = database_query_stream("SELECT * FROM users".to_string()).await;
    while let Some(row) = stream.next().await {
        println!("Row: {:?}", row);
    }

    println!("\n== Ticker Stream ==\n");
    let mut stream = ticker_stream(Duration::from_millis(100), 5).await;
    while let Some(tick) = stream.next().await {
        println!("Tick: {}", tick);
    }
}
```

```
}
```

Stream Creation Patterns: - **Manual implementation:** Full control with `Stream` trait - **Channel-based:** Producer task sends to channel - **Interval:** Time-based events - **External sources:** File system, WebSocket, database

Pattern 3: Async/Await Patterns

Problem: Manual future polling with `.poll()` is complex and error-prone. Combinator chains (`.and_then().map()`) become unreadable for complex logic.

Solution: Use `async fn` and `.await` for sequential async code that reads like sync. Mark functions `async` to return `impl Future`.

Why It Matters: Async/await transforms async programming from callback spaghetti to readable imperative code. HTTP request handler with 5 operations: combinator chain is 20 lines of `.and_then()`, async/await is 10 lines reading like sync.

Use Cases: Web servers (async request handlers), database clients (async queries), HTTP clients (async requests), file I/O (async read/write), microservices (async RPC), chat servers, real-time systems.

Example: Task Spawning and Structured Concurrency

Spawn concurrent tasks, manage their lifecycle, and coordinate their completion.

```
use tokio;
use std::time::Duration;
```

Example: Basic task spawning

This example walks through the basics of task spawning, highlighting each step so you can reuse the pattern.

```
async fn spawn_basic_tasks() {
    let handle1 = tokio::spawn(async {
        tokio::time::sleep(Duration::from_millis(100)).await;
        println!("Task 1 complete");
        42
    });

    let handle2 = tokio::spawn(async {
        tokio::time::sleep(Duration::from_millis(200)).await;
        println!("Task 2 complete");
        100
    });
}
```

```

    let (result1, result2) = tokio::join!(handle1, handle2);
    println!("Results: {:?}, {:?}", result1, result2);
}

```

Example: Structured concurrency with JoinSet

This example shows structured concurrency with JoinSet to illustrate where the pattern fits best.

```

async fn structured_concurrency() {
    use tokio::task::JoinSet;

    let mut set = JoinSet::new();

    for i in 0..5 {
        set.spawn(async move {
            tokio::time::sleep(Duration::from_millis(i * 50)).await;
            println!("Task {} done", i);
            i
        });
    }

    // Wait for all tasks
    while let Some(result) = set.join_next().await {
        match result {
            Ok(value) => println!("Got: {}", value),
            Err(e) => println!("Task failed: {}", e),
        }
    }
}

```

Example: Scoped tasks (guaranteed completion before scope ends)

This example shows scoped tasks (guaranteed completion before scope ends) to illustrate where the pattern fits best.

```

async fn scoped_tasks() {
    let mut data = vec![1, 2, 3, 4, 5];

    tokio::task::scope(|scope| {
        for item in &mut data {
            scope.spawn(async move {
                *item *= 2;
            });
        }
    });

    println!("Modified data: {:?}", data);
}

```

Example: Task cancellation

This example shows how to task cancellation in practice, emphasizing why it works.

```
use tokio_util::sync::CancellationToken;

async fn cancellable_task() {
    let token = CancellationToken::new();
    let child_token = token.child_token();

    let task = tokio::spawn(async move {
        tokio::select! {
            _ = child_token.cancelled() => {
                println!("Task cancelled");
            }
            _ = tokio::time::sleep(Duration::from_secs(10)) => {
                println!("Task completed normally");
            }
        }
    });
    tokio::time::sleep(Duration::from_millis(100)).await;
    token.cancel();

    task.await.unwrap();
}

// Real-world: Worker pool pattern
struct WorkerPool {
    tasks: tokio::sync::mpsc::Sender<Box<dyn FnOnce() + Send + 'static>>,
}

impl WorkerPool {
    fn new(num_workers: usize) -> Self {
        let (tx, mut rx) = tokio::sync::mpsc::channel::<Box<dyn FnOnce() + Send + 'static>>(100);

        for i in 0..num_workers {
            let mut rx = rx.clone();
            tokio::spawn(async move {
                while let Some(task) = rx.recv().await {
                    println!("Worker {} executing task", i);
                    task();
                }
            });
        }

        Self { tasks: tx }
    }

    async fn submit<F>(&self, task: F)
where
    F: FnOnce() + Send + 'static
{
    self.tasks.send(task).await;
}
}
```

```

    F: FnOnce() + Send + 'static,
{
    self.tasks.send(Box::new(task)).await.unwrap();
}
}

// Real-world: Supervisor pattern (restart on failure)
async fn supervised_task<F, Fut>(
    mut task_fn: F,
    max_restarts: usize,
) where
    F: FnMut() -> Fut,
    Fut: std::future::Future<Output = ()>,
{
    for attempt in 0..=max_restarts {
        let handle = tokio::spawn(task_fn());

        match handle.await {
            Ok(_) => {
                println!("Task completed successfully");
                break;
            }
            Err(e) => {
                if attempt < max_restarts {
                    println!("Task failed (attempt {}): {}. Restarting...", attempt + 1, e);
                    tokio::time::sleep(Duration::from_secs(1)).await;
                } else {
                    println!("Task failed after {} attempts", max_restarts + 1);
                }
            }
        }
    }
}

// Real-world: Background task with graceful shutdown
async fn background_service(shutdown: tokio::sync::watch::Receiver<bool>) {
    let mut shutdown = shutdown;
    let mut interval = tokio::time::interval(Duration::from_secs(1));

    loop {
        tokio::select! {
            _ = interval.tick() => {
                println!("Background service tick");
            }
            _ = shutdown.changed() => {
                println!("Shutdown signal received");
                break;
            }
        }
    }

    println!("Background service stopped");
}

async fn run_with_graceful_shutdown() {

```

```

let (shutdown_tx, shutdown_rx) = tokio::sync::watch::channel(false);

let service = tokio::spawn(background_service(shutdown_rx));

tokio::time::sleep(Duration::from_secs(3)).await;

println!("Sending shutdown signal");
shutdown_tx.send(true).unwrap();

service.await.unwrap();
}

#[tokio::main]
async fn main() {
    println!("== Basic Task Spawning ==\n");
    spawn_basic_tasks().await;

    println!("\n== Structured Concurrency ==\n");
    structured_concurrency().await;

    println!("\n== Cancellable Task ==\n");
    cancellable_task().await;

    println!("\n== Graceful Shutdown ==\n");
    run_with_graceful_shutdown().await;
}

```

Task Management Patterns: - **spawn**: Create independent task - **JoinSet**: Manage dynamic set of tasks - **Cancellation**: Cooperative cancellation with tokens - **Supervisor**: Auto-restart on failure - **Graceful shutdown**: Clean task termination

Example: Error Handling and Recovery

Handle errors in async code, implement retry logic, and provide fallback mechanisms.

```

use tokio;
use std::time::Duration;

```

Example: Result propagation with ?

This example shows how to result propagation with ? in practice, emphasizing why it works.

```

async fn fetch_user_data(user_id: u64) -> Result<String, String> {
    if user_id == 0 {
        return Err("Invalid ID".to_string());
    }
    Ok(format!("User {}", user_id))
}

```

```

async fn get_user_profile(user_id: u64) -> Result<String, String> {
    let data = fetch_user_data(user_id).await?;
    let profile = format!("Profile: {}", data);
    Ok(profile)
}

```

Example: Retry with exponential backoff

This example shows how to retry with exponential backoff in practice, emphasizing why it works.

```

async fn retry_with_backoff<F, Fut, T, E>(
    mut operation: F,
    max_retries: u32,
    initial_delay: Duration,
) -> Result<T, E>
where
    F: FnMut() -> Fut,
    Fut: std::future::Future<Output = Result<T, E>>,
    E: std::fmt::Display,
{
    let mut delay = initial_delay;

    for attempt in 0..max_retries {
        match operation().await {
            Ok(result) => return Ok(result),
            Err(e) if attempt < max_retries - 1 => {
                println!("Attempt {} failed: {}. Retrying in {:?}...", attempt + 1, e, delay);
                tokio::time::sleep(delay).await;
                delay *= 2; // Exponential backoff
            }
            Err(e) => return Err(e),
        }
    }

    unreachable!()
}

```

Example: Circuit breaker

This example shows how to circuit breaker in practice, emphasizing why it works.

```

use std::sync::Arc;
use tokio::sync::Mutex;

#[derive(Clone, Copy, Debug)]
enum CircuitState {
    Closed,
    Open,
    HalfOpen,
}

```

```

}

struct CircuitBreaker {
    state: Arc<Mutex<CircuitState>>,
    failure_threshold: usize,
    failures: Arc<Mutex<usize>>,
    success_threshold: usize,
    timeout: Duration,
}

impl CircuitBreaker {
    fn new(failure_threshold: usize, success_threshold: usize, timeout: Duration) -> Self {
        Self {
            state: Arc::new(Mutex::new(CircuitState::Closed)),
            failure_threshold,
            failures: Arc::new(Mutex::new(0)),
            success_threshold,
            timeout,
        }
    }

    async fn call<F, Fut, T, E>(&self, operation: F) -> Result<T, E>
    where
        F: FnOnce() -> Fut,
        Fut: std::future::Future<Output = Result<T, E>>,
        E: From<String>,
    {
        let state = *self.state.lock().await;

        match state {
            CircuitState::Open => {
                return Err(E::from("Circuit breaker is open".to_string()));
            }
            CircuitState::HalfOpen => {
                // Try to recover
            }
            CircuitState::Closed => {
                // Normal operation
            }
        }

        match operation().await {
            Ok(result) => {
                // Reset failures
                *self.failures.lock().await = 0;
                if !matches!(state, CircuitState::HalfOpen) {
                    *self.state.lock().await = CircuitState::Closed;
                }
                Ok(result)
            }
            Err(e) => {
                let mut failures = self.failures.lock().await;
                *failures += 1;
                Err(e)
            }
        }
    }
}

```

```
        if *failures >= self.failure_threshold {
            println!("Circuit breaker opened due to {} failures", failures);
            *self.state.lock().await = CircuitState::Open;

            // Schedule transition to half-open
            let state = Arc::clone(&self.state);
            let timeout = self.timeout;
            tokio::spawn(async move {
                tokio::time::sleep(timeout).await;
                *state.lock().await = CircuitState::HalfOpen;
                println!("Circuit breaker transitioned to half-open");
            });
        }
    }

    Err(e)
}
}
```

Example: Fallback pattern

This example shows how to fallback pattern in practice, emphasizing why it works.

```
async fn fetch_with_fallback<F, Fut, T>(
    primary: F,
    fallback_value: T,
) -> T
where
    F: FnOnce() -> Fut,
    Fut: std::future::Future<Output = Result<T, Box<dyn std::error::Error>>>,
{
    match primary().await {
        Ok(value) => value,
        Err(e) => {
            println!("Primary failed: {}. Using fallback.", e);
            fallback_value
        }
    }
}
// Real-world: Bulkhead pattern (resource isolation)
struct Bulkhead {
    semaphore: Arc<tokio::sync::Semaphore>,
}
impl Bulkhead {
    fn new(max_concurrent: usize) -> Self {
        Self {
            semaphore: Arc::new(tokio::sync::Semaphore::new(max_concurrent)),
        }
    }
}
```

```

}

async fn execute<F, Fut, T>(&self, operation: F) -> Result<T, String>
where
    F: FnOnce() -> Fut,
    Fut: std::future::Future<Output = T>,
{
    match self.semaphore.try_acquire() {
        Ok(permit) => {
            let result = operation().await;
            drop(permit);
            Ok(result)
        }
        Err(_) => Err("Bulkhead full - request rejected".to_string()),
    }
}

#[tokio::main]
async fn main() {
    println!("==== Retry with Backoff ===\n");

    let mut attempts = 0;
    let result = retry_with_backoff(
        || async {
            attempts += 1;
            if attempts < 3 {
                Err("Temporary failure")
            } else {
                Ok("Success!")
            }
        },
        5,
        Duration::from_millis(100),
    ).await;

    println!("Final result: {:?}", result);

    println!("==== Circuit Breaker ===\n");

    let breaker = CircuitBreaker::new(3, 2, Duration::from_secs(2));

    for i in 0..10 {
        let result: Result<String, String> = breaker.call(|| async {
            if i < 5 {
                Err("Service unavailable".to_string())
            } else {
                Ok("Success".to_string())
            }
        }).await;

        println!("Call {}: {:?}", i, result);
        tokio::time::sleep(Duration::from_millis(100)).await;
    }
}

```

```
    }  
}
```

Error Handling Patterns: - **Retry**: Exponential backoff - **Circuit breaker**: Prevent cascading failures
- **Fallback**: Default value on error - **Bulkhead**: Limit concurrent requests

Pattern 4: Select and Timeout Patterns

Problem: Waiting indefinitely for async operations causes hangs—network request that never responds blocks forever. Need to handle whichever of multiple operations completes first (user input vs network response).

Solution: Use `tokio::select!` to race multiple futures, completing when first finishes. Use `tokio::time::timeout()` to bound operation duration.

Why It Matters: Timeouts prevent resource leaks from hung operations—HTTP server without timeouts accumulates connections from slow clients until memory exhausted. Select enables responsive UIs: user input cancels background computation immediately.

Use Cases: HTTP clients (request timeouts), connection management (idle timeouts), health checks (periodic pings), graceful shutdown (timeout on cleanup), rate limiting (interval-based), user cancellation (input vs background work), circuit breakers.

Example: Select Patterns

Wait on multiple async operations and react to whichever completes first.

```
use tokio;  
use tokio::sync::mpsc;  
use std::time::Duration;
```

Example: Basic select with two branches

This example walks through the basics of select with two branches, highlighting each step so you can reuse the pattern.

```
async fn select_two_channels() {  
    let (tx1, mut rx1) = mpsc::channel::<i32>(10);  
    let (tx2, mut rx2) = mpsc::channel::<String>(10);  
  
    tokio::spawn(async move {  
        tokio::time::sleep(Duration::from_millis(100)).await;  
        tx1.send(42).await.unwrap();  
    });  
  
    tokio::spawn(async move {  
        tokio::time::sleep(Duration::from_millis(200)).await;  
        tx2.send("Hello".to_string()).await.unwrap();  
    });  
}
```

```

tokio::select! {
    Some(num) = rx1.recv() => {
        println!("Got number: {}", num);
    }
    Some(msg) = rx2.recv() => {
        println!("Got message: {}", msg);
    }
}
}

```

Example: Select in a loop

This example shows how to select in a loop in practice, emphasizing why it works.

```

async fn select_loop() {
    let (tx1, mut rx1) = mpsc::channel::<i32>(10);
    let (tx2, mut rx2) = mpsc::channel::<String>(10);

    // Spawn producers
    tokio::spawn(async move {
        for i in 0..5 {
            tokio::time::sleep(Duration::from_millis(100)).await;
            tx1.send(i).await.unwrap();
        }
    });

    tokio::spawn(async move {
        for i in 0..3 {
            tokio::time::sleep(Duration::from_millis(150)).await;
            tx2.send(format!("msg_{}", i)).await.unwrap();
        }
    });

    let mut done1 = false;
    let mut done2 = false;

    loop {
        tokio::select! {
            Some(num) = rx1.recv(), if !done1 => {
                println!("Number: {}", num);
            }
            Some(msg) = rx2.recv(), if !done2 => {
                println!("Message: {}", msg);
            }
            else => {
                println!("Both channels closed");
                break;
            }
        }
    }
}

```

```
    }
}
```

Example: Biased select (priority)

This example shows biased select (priority) to illustrate where the pattern fits best.

```
async fn biased_select() {
    let (tx_high, mut rx_high) = mpsc::channel::<String>(10);
    let (tx_low, mut rx_low) = mpsc::channel::<String>(10);

    tokio::spawn(async move {
        tx_high.send("High priority".to_string()).await.unwrap();
        tx_low.send("Low priority".to_string()).await.unwrap();
    });

    tokio::time::sleep(Duration::from_millis(10)).await;

    // Biased: always checks branches in order
    tokio::select! {
        biased;

        Some(msg) = rx_high.recv() => {
            println!("High: {}", msg);
        }
        Some(msg) = rx_low.recv() => {
            println!("Low: {}", msg);
        }
    }
}

// Real-world: Request with cancellation
async fn request_with_cancel() {
    let (cancel_tx, mut cancel_rx) = mpsc::channel::<()>(1);

    let request = tokio::spawn(async move {
        tokio::time::sleep(Duration::from_secs(5)).await;
        "Request complete"
    });

    tokio::spawn(async move {
        tokio::time::sleep(Duration::from_millis(500)).await;
        cancel_tx.send(()).await.unwrap();
    });

    tokio::select! {
        result = request => {
            println!("Request finished: {:?}", result);
        }
        _ = cancel_rx.recv() => {
            println!("Request cancelled");
        }
    }
}
```

```

    }
}

// Real-world: Server with shutdown signal
async fn server_with_shutdown() {
    let (shutdown_tx, mut shutdown_rx) = mpsc::channel::<()>(1);
    let (request_tx, mut request_rx) = mpsc::channel::<String>(10);

    // Simulate incoming requests
    let request_tx_clone = request_tx.clone();
    tokio::spawn(async move {
        for i in 0..10 {
            tokio::time::sleep(Duration::from_millis(200)).await;
            if request_tx_clone.send(format!("Request {}", i)).await.is_err() {
                break;
            }
        }
    });
}

// Simulate shutdown after 1 second
tokio::spawn(async move {
    tokio::time::sleep(Duration::from_secs(1)).await;
    shutdown_tx.send(()).await.unwrap();
});

// Server loop
loop {
    tokio::select! {
        Some(req) = request_rx.recv() => {
            println!("Processing: {}", req);
        }
        _ = shutdown_rx.recv() => {
            println!("Shutdown signal received");
            break;
        }
    }
}

println!("Server stopped");
}

```

Example: Select with default (non-blocking)

This example shows how to select with default (non-blocking) in practice, emphasizing why it works.

```

async fn select_with_default() {
    let (tx, mut rx) = mpsc::channel::<i32>(10);

    tokio::spawn(async move {
        tokio::time::sleep(Duration::from_millis(500)).await;
        tx.send(42).await.unwrap();
    });
}

```

```

// Try to receive immediately
tokio::select! {
    Some(value) = rx.recv() => {
        println!("Got value: {}", value);
    }
    else => {
        println!("No value available immediately");
    }
}

tokio::time::sleep(Duration::from_secs(1)).await;

// Try again after delay
tokio::select! {
    Some(value) = rx.recv() => {
        println!("Got value: {}", value);
    }
    else => {
        println!("No value available");
    }
}
}

#[tokio::main]
async fn main() {
    println!("==> Select Two Channels ==>\n");
    select_two_channels().await;

    println!("\n==> Select Loop ==>\n");
    select_loop().await;

    println!("\n==> Biased Select ==>\n");
    biased_select().await;

    println!("\n==> Request with Cancel ==>\n");
    request_with_cancel().await;

    println!("\n==> Server with Shutdown ==>\n");
    server_with_shutdown().await;

    println!("\n==> Select with Default ==>\n");
    select_with_default().await;
}

```

Select Patterns: - **Basic select:** Race multiple futures - **Loop select:** Continuous event handling - **Biased select:** Priority ordering - **With cancellation:** Abort long operations - **Default:** Non-blocking poll

Example: Timeout and Deadline Patterns

Enforce time limits on async operations to prevent indefinite blocking.

```
use tokio;
use tokio::time::{timeout, sleep, Duration, Instant};
```

Example: Basic timeout

This example walks through the basics of timeout, highlighting each step so you can reuse the pattern.

```
async fn basic_timeout() {
    let operation = async {
        sleep(Duration::from_secs(2)).await;
        "Completed"
    };

    match timeout(Duration::from_secs(1), operation).await {
        Ok(result) => println!("Result: {}", result),
        Err(_) => println!("Operation timed out"),
    }
}
```

Example: Timeout with retry

This example shows how to timeout with retry in practice, emphasizing why it works.

```
async fn timeout_with_retry() {
    for attempt in 1..=3 {
        let operation = async {
            sleep(Duration::from_millis(attempt * 400)).await;
            if attempt < 3 {
                Err("Failed")
            } else {
                Ok("Success")
            }
        };

        match timeout(Duration::from_secs(1), operation).await {
            Ok(Ok(result)) => {
                println!("Success: {}", result);
                break;
            }
            Ok(Err(e)) => {
                println!("Attempt {} failed: {}", attempt, e);
            }
            Err(_) => {
                println!("Attempt {} timed out", attempt);
            }
        }
    }
}
```

```
        }
    }
}
```

Example: Deadline tracking

This example shows how to deadline tracking in practice, emphasizing why it works.

```
async fn with_deadline<F, T>(
    future: F,
    deadline: Instant,
) -> Result<T, &'static str>
where
    F: std::future::Future<Output = T>,
{
    let duration = deadline.saturating_duration_since(Instant::now());

    match timeout(duration, future).await {
        Ok(result) => Ok(result),
        Err(_) => Err("Deadline exceeded"),
    }
}

async fn deadline_example() {
    let deadline = Instant::now() + Duration::from_secs(1);

    let result = with_deadline(
        async {
            sleep(Duration::from_millis(500)).await;
            42
        },
        deadline,
    ).await;

    println!("Result: {:?}", result);
}
```

Example: Timeout for multiple operations

This example shows how to timeout for multiple operations in practice, emphasizing why it works.

```
async fn timeout_all() {
    let operations = vec![
        tokio::spawn(async {
            sleep(Duration::from_millis(100)).await;
            1
        }),
        tokio::spawn(async {
```

```

        sleep(Duration::from_millis(200)).await;
    2
},
tokio::spawn(async {
    sleep(Duration::from_millis(300)).await;
    3
}),
];
}

let all_done = async {
    let mut results = Vec::new();
    for handle in operations {
        results.push(handle.await.unwrap());
    }
    results
};

match timeout(Duration::from_millis(250), all_done).await {
    Ok(results) => println!("All done: {:?}", results),
    Err(_) => println!("Not all operations completed in time"),
}
}

// Real-world: Rate limiter with timeout
use std::sync::Arc;
use tokio::sync::Semaphore;

struct RateLimiter {
    semaphore: Arc<Semaphore>,
    refill_amount: usize,
    refill_interval: Duration,
}

impl RateLimiter {
    fn new(capacity: usize, refill_amount: usize, refill_interval: Duration) -> Self {
        let semaphore = Arc::new(Semaphore::new(capacity));

        // Refill task
        let sem = Arc::clone(&semaphore);
        tokio::spawn(async move {
            let mut interval = tokio::time::interval(refill_interval);
            loop {
                interval.tick().await;
                sem.add_permits(refill_amount);
            }
        });
    }

    Self {
        semaphore,
        refill_amount,
        refill_interval,
    }
}

```

```

async fn acquire_with_timeout(&self, timeout_duration: Duration) -> Result<(), &'static str> {
    match timeout(timeout_duration, self.semaphore.acquire()).await {
        Ok(Ok(permit)) => {
            permit.forget(); // Consume permit
            Ok(())
        }
        Ok(Err(_)) => Err("Semaphore closed"),
        Err(_) => Err("Timeout acquiring rate limit"),
    }
}

// Real-world: Health check with timeout
async fn health_check(url: &str) -> Result<bool, Box<dyn std::error::Error>> {
    let check = async {
        let response = reqwest::get(url).await?;
        Ok(response.status().is_success())
    };

    timeout(Duration::from_secs(5), check)
        .await
        .map_err(|_| "Health check timed out".into())?
}

// Real-world: Graceful timeout (finish current work)
async fn graceful_shutdown_with_timeout(
    workers: Vec<tokio::task::JoinHandle<()>>,
    grace_period: Duration,
) {
    let shutdown = async {
        for worker in workers {
            worker.await.ok();
        }
    };

    match timeout(grace_period, shutdown).await {
        Ok(_) => println!("All workers stopped gracefully"),
        Err(_) => println!("Timeout - forcing shutdown"),
    }
}

#[tokio::main]
async fn main() {
    println!("== Basic Timeout ==\n");
    basic_timeout().await;

    println!("\n== Timeout with Retry ==\n");
    timeout_with_retry().await;

    println!("\n== Deadline ==\n");
    deadline_example().await;

    println!("\n== Timeout All ==\n");
    timeout_all().await;
}

```

```

println!("\
==== Rate Limiter ===\
");
let limiter = RateLimiter::new(5, 2, Duration::from_millis(500));

for i in 0..10 {
    match limiter.acquire_with_timeout(Duration::from_secs(1)).await {
        Ok(_) => println!("Request {} allowed", i),
        Err(e) => println!("Request {} rejected: {}", i, e),
    }
    sleep(Duration::from_millis(100)).await;
}
}

```

Timeout Patterns: - **Basic timeout:** Single operation limit - **Deadline:** Absolute time limit - **Timeout all:** Batch operation limit - **Graceful timeout:** Allow cleanup before forcing stop - **Rate limiter:** Control request rate with timeout

Pattern 5: Runtime Comparison

Problem: Choosing wrong async runtime impacts performance, features, and maintainability. Tokio dominates ecosystem but isn't always best choice.

Solution: Use Tokio for general-purpose applications: mature, full-featured, excellent ecosystem. Use async-std for simpler API, closer to std library patterns.

Why It Matters: Runtime choice determines ecosystem access—Tokio has 10x more compatible libraries than alternatives. Performance varies: work-stealing vs single-threaded, epoll vs io_uring.

Use Cases: Tokio for web servers, databases, general applications. async-std for learning, simpler projects. smol for single-threaded, minimal overhead. embassy for embedded systems, bare-metal. Runtime-agnostic libraries for maximum compatibility.

Example: Tokio Runtime Features

Understand Tokio's features and how to configure them for different workloads.

```

use tokio;
use std::time::Duration;

```

Example: Multi-threaded runtime (default)

This example shows multi-threaded runtime (default) to illustrate where the pattern fits best.

```

#[tokio::main]
async fn multi_threaded_example() {
    println!("Running on multi-threaded runtime");

    let handles: Vec<_> = (0..10)

```

```

.map(|i| {
    tokio::spawn(async move {
        println!("Task {} on thread {:?}", i, std::thread::current().id());
        tokio::time::sleep(Duration::from_millis(10)).await;
    })
})
.collect();

for handle in handles {
    handle.await.unwrap();
}
}

```

Example: Single-threaded runtime

This example shows single-threaded runtime to illustrate where the pattern fits best.

```

#[tokio::main(flavor = "current_thread")]
async fn single_threaded_example() {
    println!("Running on single-threaded runtime");

    let thread_id = std::thread::current().id();

    for i in 0..5 {
        tokio::spawn(async move {
            println!("Task {} on thread {:?}", i, std::thread::current().id());
        }).await.unwrap();
    }

    println!("All tasks ran on thread {:?}", thread_id);
}

```

Example: Custom runtime configuration

This example shows how to custom runtime configuration while calling out the practical trade-offs.

```

fn custom_runtime_example() {
    let runtime = tokio::runtime::Builder::new_multi_thread()
        .worker_threads(4)
        .thread_name("my-worker")
        .thread_stack_size(3 * 1024 * 1024)
        .enable_all()
        .build()
        .unwrap();

    runtime.block_on(async {
        println!("Running on custom runtime");

        for i in 0..4 {
    
```

```

        tokio::spawn(async move {
            println!("Task {} started", i);
            tokio::time::sleep(Duration::from_millis(100)).await;
        });
    }

    tokio::time::sleep(Duration::from_millis(200)).await;
});
}

```

Example: Blocking operations

This example shows blocking operations to illustrate where the pattern fits best.

```

async fn handle_blocking_operations() {
    // Bad: blocks the async runtime
    // std::thread::sleep(Duration::from_secs(1));

    // Good: run blocking code on dedicated thread pool
    let result = tokio::task::spawn_blocking(|| {
        std::thread::sleep(Duration::from_secs(1));
        "Blocking operation complete"
    }).await.unwrap();

    println!("{}", result);
}

```

Example: Local task set (for !Send futures)

This example shows how to local task set (for !Send futures) in practice, emphasizing why it works.

```

use tokio::task::LocalSet;

async fn local_task_set_example() {
    use std::rc::Rc;

    let local = LocalSet::new();

    let nonsend_data = Rc::new(42);

    local.run_until(async move {
        let data = Rc::clone(&nonsend_data);

        tokio::task::spawn_local(async move {
            println!("Local task with Rc: {}", data);
        }).await.unwrap();
    }).await;
}

// Real-world: CPU-bound work with rayon

```

```
use rayon::prelude::*;

async fn cpu_bound_with_rayon() {
    let numbers: Vec<u64> = (0..1_000_000).collect();

    let sum = tokio::task::spawn_blocking(move || {
        numbers.par_iter().sum::<u64>()
    }).await.unwrap();

    println!("Sum: {}", sum);
}

// Real-world: Mixed workload (I/O and CPU)
async fn mixed_workload() {
    let io_task = tokio::spawn(async {
        for i in 0..5 {
            println!("I/O task {}", i);
            tokio::time::sleep(Duration::from_millis(100)).await;
        }
    });

    let cpu_task = tokio::task::spawn_blocking(|| {
        for i in 0..5 {
            println!("CPU task {}", i);
            std::thread::sleep(Duration::from_millis(100));

            // Simulate CPU-intensive work
            let _ = (0..1_000_000).sum::<u64>();
        }
    });

    tokio::join!(io_task, cpu_task);
}

fn main() {
    println!("== Multi-threaded Runtime ==\n");
    tokio::runtime::Runtime::new()
        .unwrap()
        .block_on(multi_threaded_example());

    println!("\n== Custom Runtime ==\n");
    custom_runtime_example();

    println!("\n== Blocking Operations ==\n");
    tokio::runtime::Runtime::new()
        .unwrap()
        .block_on(handle_blocking_operations());

    println!("\n== Local Task Set ==\n");
    tokio::runtime::Runtime::new()
        .unwrap()
        .block_on(local_task_set_example());

    println!("\n== Mixed Workload ==\n");
}
```

```

tokio::runtime::Runtime::new()
    .unwrap()
    .block_on(mixed_workload());
}

```

Tokio Features: - **Multi-threaded:** Work-stealing scheduler - **Current-thread:** Single-threaded for simpler workloads - **spawn_blocking:** Offload blocking operations - **LocalSet:** Support !Send futures - **Configurable:** Thread count, stack size, naming

Example: Runtime Comparison and Interop

Compare Tokio and async-std, understand trade-offs, and enable interoperability.

```

// Tokio version
#[cfg(feature = "tokio-runtime")]
mod tokio_example {
    use tokio;
    use std::time::Duration;

    #[tokio::main]
    pub async fn run() {
        println!("== Tokio Runtime ===");

        let handles: Vec<_> = (0..5)
            .map(|i| {
                tokio::spawn(async move {
                    tokio::time::sleep(Duration::from_millis(100)).await;
                    i * 2
                })
            })
            .collect();

        for handle in handles {
            let result = handle.await.unwrap();
            println!("Result: {}", result);
        }
    }
}

// async-std version
#[cfg(feature = "async-std-runtime")]
mod async_std_example {
    use async_std;
    use std::time::Duration;

    #[async_std::main]
    pub async fn run() {
        println!("== async-std Runtime ===");

        let handles: Vec<_> = (0..5)
            .map(|i| {
                async_std::task::spawn(async move {

```

```

        async_std::task::sleep(Duration::from_millis(100)).await;
        i * 2
    })
}
.collect();

for handle in handles {
    let result = handle.await;
    println!("Result: {}", result);
}
}

// Runtime-agnostic code using futures
mod runtime_agnostic {
    use futures::future::{join_all, FutureExt};
    use std::future::Future;
    use std::pin::Pin;

    pub async fn process_items<F, Fut>(
        items: Vec<i32>,
        process: F,
    ) -> Vec<i32>
    where
        F: Fn(i32) -> Fut,
        Fut: Future<Output = i32>,
    {
        let futures: Vec<_> = items.into_iter().map(process).collect();
        join_all(futures).await
    }
}

// Feature comparison
/***
 * Tokio vs async-std:
 *
 * Tokio:
 * - Work-stealing scheduler (better for CPU-intensive tasks)
 * - More configuration options
 * - Larger ecosystem (widely used)
 * - spawn_blocking for blocking operations
 * - Good for web servers, databases
 *
 * async-std:
 * - Simpler API (mirrors std library)
 * - Easier to learn
 * - Good for general-purpose async
 * - Less configuration needed
 * - Good for CLI tools, simpler services
 */
// Performance comparison example
#[cfg(feature = "tokio-runtime")]
async fn tokio_performance_test() {
    use tokio::time::{Instant, Duration};

```

```

let start = Instant::now();

let handles: Vec<_> = (0..1000)
    .map(|_| {
        tokio::spawn(async {
            tokio::time::sleep(Duration::from_millis(1)).await;
        })
    })
    .collect();

for handle in handles {
    handle.await.unwrap();
}

println!("Tokio: 1000 tasks in {:?}", start.elapsed());
}

#[cfg(feature = "async-std-runtime")]
async fn async_std_performance_test() {
    use async_std::task;
    use std::time::Instant;
    use std::time::Duration;

    let start = Instant::now();

    let handles: Vec<_> = (0..1000)
        .map(|_| {
            task::spawn(async {
                task::sleep(Duration::from_millis(1)).await;
            })
        })
        .collect();

    for handle in handles {
        handle.await;
    }

    println!("async-std: 1000 tasks in {:?}", start.elapsed());
}

// Interop: using futures crate for compatibility
use futures::executor::block_on;
use futures::future::join;

async fn runtime_independent_function() -> i32 {
    42
}

fn interop_example() {
    // Can run with any executor
    let result = block_on(async {
        let (a, b) = join(
            runtime_independent_function(),
            runtime_independent_function(),
    });
}

```

```

    ).await;
    a + b
});

println!("Interop result: {}", result);
}

/**
 * Choosing a Runtime:
 *
 * Use Tokio when:
 * - Building high-performance web servers
 * - Need fine-grained control over runtime
 * - Working with Tokio ecosystem (tonic, axum, etc.)
 * - CPU-bound tasks mixed with I/O
 *
 * Use async-std when:
 * - Building CLI tools or simpler services
 * - Want std-like API familiarity
 * - Primarily I/O-bound workload
 * - Simpler application with less configuration
 *
 * Use runtime-agnostic futures when:
 * - Writing libraries
 * - Need portability
 * - Want to avoid runtime lock-in
 */

```

```

fn main() {
    println!("== Runtime Interop ==\n");
    interop_example();

    #[cfg(feature = "tokio-runtime")]
    tokio_example::run();

    #[cfg(feature = "async-std-runtime")]
    async_std_example::run();
}

```

Runtime Comparison:

| Feature | Tokio | async-std |
|----------------|------------------------|-------------------------|
| Scheduler | Work-stealing | Work-stealing |
| API Style | Tokio-specific | std-like |
| Ecosystem | Large | Moderate |
| Configuration | Extensive | Minimal |
| Learning Curve | Moderate | Gentle |
| Best For | Web servers, databases | CLI tools, simpler apps |

Summary

This chapter covered async runtime patterns in Rust:

1. **Future Composition**: Combinators, concurrent execution (join/select), error handling
2. **Stream Processing**: Combinators, async generators, rate limiting, batching
3. **Async/Await**: Task spawning, structured concurrency, cancellation, error recovery
4. **Select/Timeout**: Racing futures, deadlines, graceful shutdown, rate limiting
5. **Runtime Comparison**: Tokio vs async-std, features, performance, interoperability

Key Takeaways: - **async/await** provides ergonomic async programming - **Streams** process sequences of async values - **select!** enables event-driven programming - **Timeout** prevents indefinite blocking - **Tokio** for high-performance servers, **async-std** for simpler apps - Use **spawn_blocking** for CPU-bound work - **Structured concurrency** with JoinSet ensures cleanup

Performance Guidelines: - Prefer async for I/O-bound tasks - Use spawn_blocking for CPU-bound work - Limit concurrent tasks to avoid overwhelming resources - Use streams for backpressure - Benchmark runtime choice for your workload

Common Patterns: - **Circuit breaker**: Prevent cascading failures - **Retry with backoff**: Handle transient errors - **Rate limiting**: Control resource usage - **Graceful shutdown**: Clean termination - **Request-response**: Structured communication

Safety: - Send/Sync enforce thread safety - Cancellation is cooperative - No data races (enforced by type system) - Borrow checker prevents use-after-free

Atomic Operations & Lock-Free Programming

- Locks can block threads. When a thread tries to acquire a lock that's held, it *stops* and waits. A stalled or slow thread cannot prevent others from making progress.

1. Lock-free operations never block

Example: In an audio processing thread, blocking even briefly can cause audible glitches.

- Locks guarantee nothing under contention—threads can starve.

1. Lock-free structures guarantee system-wide progress:

Lock-free: at least one thread always makes progress.

Wait-free: every thread makes progress within a bounded time.

- Locks come with hazards: Deadlocks

Priority inversion (low-priority thread holds lock; high-priority thread waits)

Convoys (one slow thread causes others to queue)

1. Lock-free code avoids all of these because it never "holds" exclusive access.

- With many CPUs hammering the same lock, performance collapses: Threads constantly block, sleep, wake up (expensive operations)
Cache lines bounce between cores like crazy

1. Lock-free operations often scale much better because:

They use atomic instructions (CAS, fetch_add) that avoid kernel involvement

They allow optimistic concurrency—many threads proceed in parallel

- Under load, locks often collapse into long queues, causing: Latency spikes
Tail latency problems (p99, p999)

1. Lock-free structures often offer:

Predictable latency

Fewer outlier delays

✖ Why NOT Use Lock-Free Everywhere?

Lock-free is **hard**: - Complex to design - Easy to introduce subtle memory-ordering bugs - ABA problem must be handled (with hazard pointers or epoch GC) - Debugging is more difficult - Unsafe code is often required

Locks are: - Simple - Correct by default - Good enough for most workloads

Rule of thumb: > Use locks unless you have a *measurable* reason not to.

| Feature | Locks | Lock-Free |
|---------------------|----------|------------|
| Blocking | Yes | No |
| Deadlocks | Possible | Impossible |
| Priority inversion | Possible | Impossible |
| Contention behavior | Poor | Often good |

| Feature | Locks | Lock-Free |
|------------|-----------|-------------|
| Latency | Can spike | More stable |
| Complexity | Low | High |
| Safety | Easy | Hard |

Pattern 1: Memory Ordering Semantics

Problem: CPU reordering and compiler optimizations can break lock-free algorithms—writes may be visible in different order than written. **Relaxed** ordering is fast but provides no guarantees, causing race conditions.

Solution: Use **Acquire** for reads that need to see all previous writes. Use **Release** for writes that make previous operations visible.

Why It Matters: Ordering determines correctness and performance. Wrong ordering: lock-free queue corrupts data, appears to work in tests, fails randomly in production.

Use Cases: Lock-free data structures (queues, stacks, maps), reference counting (Arc), flags and signals, atomic counters, synchronization primitives, wait-free algorithms.

Example: Memory Ordering Fundamentals

Understand different memory orderings and their performance/correctness trade-offs.

```
use std::sync::atomic::{AtomicBool, AtomicUsize, Ordering};
use std::sync::Arc;
use std::thread;
use std::time::Duration;
```

Example: Relaxed - No ordering guarantees (fastest)

This example shows how to use Relaxed to no ordering guarantees (fastest) without over-synchronizing.

```
// Use for: Counters where exact ordering doesn't matter
fn relaxed_ordering_example() {
    let counter = Arc::new(AtomicUsize::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        handles.push(thread::spawn(move || {
            for _ in 0..1000 {
                // Relaxed: no synchronization, just atomicity
                counter.fetch_add(1, Ordering::Relaxed);
            }
        }));
    }
}
```

```

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

// Final value is guaranteed to be correct
// but intermediate values may have appeared in any order
println!("Counter (Relaxed): {}", counter.load(Ordering::Relaxed));
}

```

Example: Acquire/Release - Synchronization without sequential consistency

This example shows how to use Acquire/Release to synchronization without sequential consistency without over-synchronizing.

```

// Use for: Producer-consumer, message passing
fn acquire_release_ordering() {
    let data = Arc::new(AtomicUsize::new(0));
    let ready = Arc::new(AtomicBool::new(false));

    let data_clone = Arc::clone(&data);
    let ready_clone = Arc::clone(&ready);

    // Producer
    let producer = thread::spawn(move || {
        // Write data
        data_clone.store(42, Ordering::Relaxed);

        // Release: all previous writes visible to thread that Acquires
        ready_clone.store(true, Ordering::Release);
    });

    // Consumer
    let consumer = thread::spawn(move || {
        // Acquire: see all writes before the Release
        while !ready.load(Ordering::Acquire) {
            thread::yield_now();
        }

        // Guaranteed to see data == 42
        let value = data.load(Ordering::Relaxed);
        println!("Consumer sees: {}", value);
        assert_eq!(value, 42);
    });

    producer.join().unwrap();
    consumer.join().unwrap();
}

```

Example: SeqCst - Sequential consistency (slowest, easiest to reason about)

This example shows how to use SeqCst to sequential consistency (slowest, easiest to reason about) without over-synchronizing.

```
// Use for: When correctness is critical and performance is secondary
fn seq_cst_ordering() {
    let x = Arc::new(AtomicBool::new(false));
    let y = Arc::new(AtomicBool::new(false));
    let z1 = Arc::new(AtomicBool::new(false));
    let z2 = Arc::new(AtomicBool::new(false));

    let x1 = Arc::clone(&x);
    let y1 = Arc::clone(&y);
    let z1_clone = Arc::clone(&z1);

    let t1 = thread::spawn(move || {
        x1.store(true, Ordering::SeqCst);
        if !y1.load(Ordering::SeqCst) {
            z1_clone.store(true, Ordering::SeqCst);
        }
    });
    let x2 = Arc::clone(&x);
    let y2 = Arc::clone(&y);
    let z2_clone = Arc::clone(&z2);

    let t2 = thread::spawn(move || {
        y2.store(true, Ordering::SeqCst);
        if !x2.load(Ordering::SeqCst) {
            z2_clone.store(true, Ordering::SeqCst);
        }
    });
    t1.join().unwrap();
    t2.join().unwrap();

    // With SeqCst: cannot have both z1 and z2 true
    // Without SeqCst: theoretically possible (hardware reordering)
    let both = z1.load(Ordering::SeqCst) && z2.load(Ordering::SeqCst);
    println!("Both flags set: {} (should be false with SeqCst)", both);
}
```

Example: AcqRel - Combine Acquire and Release

This example shows how to use AcqRel to combine acquire and release without over-synchronizing.

```
// Use for: Read-modify-write operations
fn acq_rel_ordering() {
```

```

let counter = Arc::new(AtomicUsize::new(0));

let mut handles = vec![];
for _ in 0..5 {
    let counter = Arc::clone(&counter);
    handles.push(thread::spawn(move || {
        for _ in 0..100 {
            // AcqRel: Acts as Acquire for load, Release for store
            counter.fetch_add(1, Ordering::AcqRel);
        }
    }));
}
for handle in handles {
    handle.join().unwrap();
}

println!("Counter (AcqRel): {}", counter.load(Ordering::Acquire));
}

// Real-world: Spinlock with proper ordering
struct Spinlock {
    locked: AtomicBool,
}

impl Spinlock {
    fn new() -> Self {
        Self {
            locked: AtomicBool::new(false),
        }
    }

    fn lock(&self) {
        while self
            .locked
            .compare_exchange_weak(
                false,
                true,
                Ordering::Acquire, // Success: acquire lock
                Ordering::Relaxed, // Failure: just retry
            )
            .is_err()
        {
            // Hint to CPU we're spinning
            while self.locked.load(Ordering::Relaxed) {
                std::hint::spin_loop();
            }
        }
    }

    fn unlock(&self) {
        // Release: make all previous writes visible
        self.locked.store(false, Ordering::Release);
    }
}

```

```

}

// Real-world: Double-checked locking for lazy initialization
struct LazyInit<T> {
    data: AtomicUsize, // Actually *mut T
    initialized: AtomicBool,
}

impl<T> LazyInit<T> {
    fn new() -> Self {
        Self {
            data: AtomicUsize::new(0),
            initialized: AtomicBool::new(false),
        }
    }

    fn get_or_init<F>(&self, init: F) -> &T
    where
        F: FnOnce() -> T,
    {
        // Fast path: already initialized (Acquire ensures we see the data)
        if self.initialized.load(Ordering::Acquire) {
            unsafe { &*(self.data.load(Ordering::Relaxed) as *const T) }
        } else {
            self.init_slow(init)
        }
    }

    fn init_slow<F>(&self, init: F) -> &T
    where
        F: FnOnce() -> T,
    {
        let ptr = Box::into_raw(Box::new(init()));

        // Try to publish (use SeqCst for correctness)
        match self.initialized.compare_exchange(
            false,
            true,
            Ordering::SeqCst,
            Ordering::SeqCst,
        ) {
            Ok(_) => {
                // We won the race
                self.data.store(ptr as usize, Ordering::Release);
                unsafe { &*ptr }
            }
            Err(_) => {
                // Someone else won, clean up our allocation
                unsafe { drop(Box::from_raw(ptr)) };
                unsafe { &*(self.data.load(Ordering::Acquire) as *const T) }
            }
        }
    }
}

```

```

fn main() {
    println!("== Relaxed Ordering ==\n");
    relaxed_ordering_example();

    println!("\n== Acquire/Release Ordering ==\n");
    acquire_release_ordering();

    println!("\n== Sequential Consistency ==\n");
    seq_cst_ordering();

    println!("\n== AcqRel Ordering ==\n");
    acq_rel_ordering();

    println!("\n== Spinlock ==\n");
    let lock = Arc::new(Spinlock::new());
    let mut handles = vec![];

    for i in 0..3 {
        let lock = Arc::clone(&lock);
        handles.push(thread::spawn(move || {
            lock.lock();
            println!("Thread {} acquired lock", i);
            thread::sleep(Duration::from_millis(10));
            lock.unlock();
            println!("Thread {} released lock", i);
        }));
    }

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

```

Memory Ordering Guide:

| Ordering | Guarantees | Use Case | Performance |
|----------|--------------------------------|--|-------------|
| Relaxed | Atomicity only | Counters, flags (order doesn't matter) | Fastest |
| Acquire | See writes before Release | Consumer in producer-consumer | Fast |
| Release | Publish writes to Acquire | Producer in producer-consumer | Fast |
| AcqRel | Both Acquire and Release | RMW operations | Medium |
| SeqCst | Total order across all threads | When correctness is critical | Slowest |

Example: Fence Operations

Establish memory ordering without atomic operations, or strengthen ordering of existing atomics.

```
use std::sync::atomic::{fence, AtomicBool, AtomicUsize, Ordering};
use std::sync::Arc;
use std::thread;
```

Example: Fence for non-atomic data

This example shows how to fence for non-atomic data in practice, emphasizing why it works.

```
fn fence_with_non_atomic() {
    let mut data = 0u64;
    let ready = Arc::new(AtomicBool::new(false));
    let ready_clone = Arc::clone(&ready);

    // Producer
    let producer = thread::spawn(move || {
        unsafe {
            let data_ptr = &mut data as *mut u64;

            // Write non-atomic data
            *data_ptr = 42;

            // Fence ensures all previous writes are visible
            fence(Ordering::Release);

            // Signal ready
            ready_clone.store(true, Ordering::Relaxed);
        }
    });

    // Consumer
    thread::sleep(std::time::Duration::from_millis(10));

    if ready.load(Ordering::Relaxed) {
        // Fence ensures we see all writes before the Release fence
        fence(Ordering::Acquire);

        // Now safe to read data
        println!("Data: {}", data);
    }

    producer.join().unwrap();
}
```

Example: Compiler fence (prevents compiler reordering only)

This example shows how to compiler fence (prevents compiler reordering only) in practice, emphasizing why it works.

```

fn compiler_fence_example() {
    let x = AtomicUsize::new(0);
    let y = AtomicUsize::new(0);

    x.store(1, Ordering::Relaxed);

    // Prevent compiler from reordering (hardware can still reorder)
    std::sync::atomic::compiler_fence(Ordering::SeqCst);

    y.store(2, Ordering::Relaxed);

    // Ensures compiler sees x=1 before y=2
}

// Real-world: Memory barrier for DMA/MMIO
#[repr(C)]
struct DmaBuffer {
    data: [u8; 4096],
    ready: AtomicBool,
}

impl DmaBuffer {
    fn write_for_dma(&mut self, data: &[u8]) {
        self.data[..data.len()].copy_from_slice(data);

        // Ensure all writes complete before signaling device
        fence(Ordering::Release);

        self.ready.store(true, Ordering::Relaxed);
    }

    fn read_from_dma(&mut self) -> Option<&[u8]> {
        if !self.ready.load(Ordering::Relaxed) {
            return None;
        }

        // Ensure we see all device writes
        fence(Ordering::Acquire);

        Some(&self.data)
    }
}

fn main() {
    println!("==== Fence with Non-Atomic ====\n");
    fence_with_non_atomic();

    println!("\n==== Compiler Fence ====\n");
    compiler_fence_example();
}

```

Fence Types: - **fence(Ordering)**: Hardware and compiler barrier - **compiler_fence(Ordering)**: Compiler-only barrier (no CPU fence) - Use for: MMIO, DMA, FFI boundaries

Pattern 2: Compare-and-Swap Patterns

Problem: Implementing lock-free operations requires atomic read-modify-write. Naive approaches have race conditions when multiple threads update simultaneously.

Solution: Use compare-and-swap (CAS) as fundamental building block. Load current value, compute new value, CAS to update only if unchanged.

Why It Matters: CAS is foundation of all lock-free algorithms. Without proper CAS loops, concurrent updates are lost.

Use Cases: Lock-free stacks and queues, atomic max/min tracking, conditional increments (rate limiting), version tracking, optimistic updates, retry logic.

Example: CAS Basics and Patterns

Using compare-and-swap to implement lock-free operations correctly.

```
use std::sync::atomic::{AtomicUsize, AtomicPtr, Ordering};
use std::sync::Arc;
use std::thread;
use std::ptr;
```

Example: Basic CAS loop

This example walks through the basics of cas loop, highlighting each step so you can reuse the pattern.

```
fn cas_increment(counter: &AtomicUsize) {
    loop {
        let current = counter.load(Ordering::Relaxed);
        let new_value = current + 1;

        // Try to update: succeeds only if value hasn't changed
        if counter
            .compare_exchange_weak(
                current,
                new_value,
                Ordering::Relaxed,
                Ordering::Relaxed,
            )
            .is_ok()
        {
            break;
        }

        // Spurious failure or actual contention - retry
    }
}
```

```
    }  
}
```

Example: compare_exchange vs compare_exchange_weak

This example shows how to compare_exchange vs compare_exchange_weak in practice, emphasizing why it works.

```
fn compare_exchange_variants() {  
    let value = AtomicUsize::new(0);  
  
    // compare_exchange: never spurious failure, use in non-loop  
    let result = value.compare_exchange(  
        0,  
        1,  
        Ordering::SeqCst,  
        Ordering::SeqCst,  
    );  
    assert!(!result.is_ok());  
  
    // compare_exchange_weak: may spuriously fail, use in loop  
    loop {  
        if value  
            .compare_exchange_weak(1, 2, Ordering::SeqCst, Ordering::SeqCst)  
            .is_ok()  
        {  
            break;  
        }  
    }  
  
    println!("Final value: {}", value.load(Ordering::SeqCst));  
}
```

Example: CAS with data transformation

This example shows how to cAS with data transformation in practice, emphasizing why it works.

```
fn cas_update<F>(counter: &AtomicUsize, f: F)  
where  
    F: Fn(usize) -> usize,  
{  
    let mut current = counter.load(Ordering::Relaxed);  
  
    loop {  
        let new_value = f(current);  
  
        match counter.compare_exchange_weak(  
            current,  
            new_value,  
            Ordering::Relaxed,  
            Ordering::Relaxed  
        )  
        {  
            Ok(_) => break;  
            Err(_) => current = counter.load(Ordering::Relaxed),  
        }  
    }  
}
```

```

        Ordering::Relaxed,
        Ordering::Relaxed,
    ) {
    Ok(_) => break,
    Err(actual) => current = actual, // Update current and retry
}
}

// Real-world: Lock-free max tracking
struct MaxTracker {
    max: AtomicUsize,
}

impl MaxTracker {
    fn new() -> Self {
        Self {
            max: AtomicUsize::new(0),
        }
    }

    fn update(&self, value: usize) {
        let mut current = self.max.load(Ordering::Relaxed);

        loop {
            if value <= current {
                // Already have a larger max
                break;
            }

            match self.max.compare_exchange_weak(
                current,
                value,
                Ordering::Relaxed,
                Ordering::Relaxed,
            ) {
                Ok(_) => break,
                Err(actual) => current = actual,
            }
        }
    }

    fn get(&self) -> usize {
        self.max.load(Ordering::Relaxed)
    }
}

// Real-world: Lock-free accumulator
struct Accumulator {
    sum: AtomicUsize,
    count: AtomicUsize,
}

impl Accumulator {
    fn new() -> Self {

```

```

    Self {
        sum: AtomicUsize::new(0),
        count: AtomicUsize::new(0),
    }
}

fn add(&self, value: usize) {
    self.sum.fetch_add(value, Ordering::Relaxed);
    self.count.fetch_add(1, Ordering::Relaxed);
}

fn average(&self) -> f64 {
    let sum = self.sum.load(Ordering::Relaxed);
    let count = self.count.load(Ordering::Relaxed);

    if count == 0 {
        0.0
    } else {
        sum as f64 / count as f64
    }
}

fn reset(&self) -> (usize, usize) {
    let sum = self.sum.swap(0, Ordering::Relaxed);
    let count = self.count.swap(0, Ordering::Relaxed);
    (sum, count)
}
}

// Real-world: Conditional update
struct ConditionalCounter {
    value: AtomicUsize,
}

impl ConditionalCounter {
    fn new(initial: usize) -> Self {
        Self {
            value: AtomicUsize::new(initial),
        }
    }

    fn increment_if_below(&self, threshold: usize) -> bool {
        let mut current = self.value.load(Ordering::Relaxed);

        loop {
            if current >= threshold {
                return false; // Can't increment
            }

            match self.value.compare_exchange_weak(
                current,
                current + 1,
                Ordering::Relaxed,
                Ordering::Relaxed,
            )
            {
                Err(_) => continue,
                Ok(new_value) => current = new_value,
            }
        }
    }
}

```

```

        ) {
            Ok(_) => return true,
            Err(actual) => current = actual,
        }
    }
}

fn get(&self) -> usize {
    self.value.load(Ordering::Relaxed)
}
}

fn main() {
    println!("== CAS Increment ==\n");

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

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        handles.push(thread::spawn(move || {
            for _ in 0..100 {
                cas_increment(&counter);
            }
        }));
    }

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

    println!("Final count: {}", counter.load(Ordering::Relaxed));

    println!("\n== Max Tracker ==\n");

    let tracker = Arc::new(MaxTracker::new());
    let mut handles = vec![];

    for i in 0..10 {
        let tracker = Arc::clone(&tracker);
        handles.push(thread::spawn(move || {
            for j in 0..100 {
                tracker.update(i * 100 + j);
            }
        }));
    }

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

    println!("Max value: {}", tracker.get());
}

```

```

println!("\n==== Accumulator ===\n");

let acc = Accumulator::new();

for i in 1..=100 {
    acc.add(i);
}

println!("Average: {:.2}", acc.average());
println!("Reset: {:?}", acc.reset());
println!("After reset: {:.2}", acc.average());

println!("\n==== Conditional Counter ===\n");

let counter = ConditionalCounter::new(0);

for _ in 0..15 {
    if counter.increment_if_below(10) {
        println!("Incremented to {}", counter.get());
    } else {
        println!("Threshold reached: {}", counter.get());
    }
}
}

```

CAS Patterns: - **Basic loop:** Load, compute, CAS, retry on failure - **compare_exchange_weak:** Use in loops (may spuriously fail) - **compare_exchange:** Use outside loops (stronger guarantee) - **Failure handling:** Update current value on failure

Example: ABA Problem and Solutions

Detect and prevent the ABA problem where a value changes from A to B back to A, fooling CAS.

```

use std::sync::atomic::{AtomicUsize, AtomicU64, Ordering};
use std::sync::Arc;
use std::thread;
use std::time::Duration;
// Problem: ABA without protection
struct NaiveStack<T> {
    head: AtomicUsize, // *mut Node<T>
    _phantom: std::marker::PhantomData<T>,
}

struct Node<T> {
    data: T,
    next: *mut Node<T>,
}

// This is unsafe and suffers from ABA problem!
impl<T> NaiveStack<T> {
    unsafe fn pop_unsafe(&self) -> Option<T> {
        loop {
            let head = self.head.load(Ordering::Acquire) as *mut Node<T>;

```

```

        if head.is_null() {
            return None;
        }

        let next = (*head).next;

        // ABA PROBLEM: Between load and CAS, another thread could:
        // 1. Pop this node
        // 2. Free it
        // 3. Push new nodes
        // 4. Push this node back (same address!)
        // CAS succeeds but we're in trouble!

        if self
            .head
            .compare_exchange(
                head as usize,
                next as usize,
                Ordering::Release,
                Ordering::Acquire,
            )
            .is_ok()
        {
            let data = std::ptr::read(&(*head).data);
            drop(Box::from_raw(head));
            return Some(data);
        }
    }
}

// Solution 1: Tagged pointers (version counter)
struct TaggedPtr {
    value: AtomicU64, // Upper 16 bits: tag, Lower 48 bits: pointer
}

impl TaggedPtr {
    fn new(ptr: *mut u8) -> Self {
        Self {
            value: AtomicU64::new(ptr as u64),
        }
    }

    fn load(&self, ordering: Ordering) -> (*mut u8, u16) {
        let packed = self.value.load(ordering);
        let ptr = (packed & 0x0000_FFFF_FFFF_FFFF) as *mut u8;
        let tag = (packed >> 48) as u16;
        (ptr, tag)
    }

    fn store(&self, ptr: *mut u8, tag: u16, ordering: Ordering) {
        let packed = ((tag as u64) << 48) | ((ptr as u64) & 0x0000_FFFF_FFFF_FFFF);
        self.value.store(packed, ordering);
    }
}

```

```

}

fn compare_exchange(
    &self,
    current_ptr: *mut u8,
    current_tag: u16,
    new_ptr: *mut u8,
    new_tag: u16,
    success: Ordering,
    failure: Ordering,
) -> Result<(), ()> {
    let current = ((current_tag as u64) << 48) | ((current_ptr as u64) &
    0x0000_FFFF_FFFF_FFFF);
    let new = ((new_tag as u64) << 48) | ((new_ptr as u64) & 0x0000_FFFF_FFFF_FFFF);

    self.value
        .compare_exchange(current, new, success, failure)
        .map(|_| ())
        .map_err(|_| ())
    }
}

// Solution 2: Version counter approach
struct VersionedStack<T> {
    head: AtomicU64, // Upper 32 bits: version, Lower 32 bits: index
    nodes: Vec<Option<VersionedNode<T>>>,
}

struct VersionedNode<T> {
    data: T,
    next: u32,
    version: u32,
}

impl<T> VersionedStack<T> {
    fn pack(index: u32, version: u32) -> u64 {
        ((version as u64) << 32) | (index as u64)
    }

    fn unpack(packed: u64) -> (u32, u32) {
        let index = (packed & 0xFFFF_FFFF) as u32;
        let version = (packed >> 32) as u32;
        (index, version)
    }
}

// Solution 3: Epoch-based reclamation (simplified)
struct EpochGC {
    global_epoch: AtomicUsize,
}

impl EpochGC {
    fn new() -> Self {
        Self {
            global_epoch: AtomicUsize::new(0),

```

```
        }

    }

    fn pin(&self) -> usize {
        self.global_epoch.load(Ordering::Acquire)
    }

    fn try_advance(&self) {
        self.global_epoch.fetch_add(1, Ordering::Release);
    }

    fn is_safe_to_free(&self, allocation_epoch: usize) -> bool {
        let current = self.global_epoch.load(Ordering::Acquire);
        current > allocation_epoch + 2 // Conservative: 2 epochs old
    }
}

// Real-world: ABA-safe counter
struct ABACounter {
    value: AtomicU64, // Upper 32 bits: version, Lower 32 bits: count
}

impl ABACounter {
    fn new(initial: u32) -> Self {
        Self {
            value: AtomicU64::new(initial as u64),
        }
    }

    fn increment(&self) {
        loop {
            let current = self.value.load(Ordering::Relaxed);
            let count = (current & 0xFFFF_FFFF) as u32;
            let version = (current >> 32) as u32;

            let new_count = count.wrapping_add(1);
            let new_version = version.wrapping_add(1);
            let new_value = ((new_version as u64) << 32) | (new_count as u64);

            if self
                .value
                .compare_exchange_weak(current, new_value, Ordering::Relaxed,
Ordering::Relaxed)
                .is_ok()
            {
                break;
            }
        }
    }

    fn get(&self) -> u32 {
        let packed = self.value.load(Ordering::Relaxed);
        (packed & 0xFFFF_FFFF) as u32
    }
}
```

```

fn get_with_version(&self) -> (u32, u32) {
    let packed = self.value.load(Ordering::Relaxed);
    let count = (packed & 0xFFFF_FFFF) as u32;
    let version = (packed >> 32) as u32;
    (count, version)
}

fn main() {
    println!("==== ABA Counter ====\n");

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

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        handles.push(thread::spawn(move || {
            for _ in 0..1000 {
                counter.increment();
            }
        }));
    }

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

    let (count, version) = counter.get_with_version();
    println!("Count: {}, Version: {}", count, version);

    println!("\n==== Epoch GC ====\n");

    let gc = EpochGC::new();

    let epoch1 = gc.pin();
    println!("Pinned at epoch {}", epoch1);

    gc.try_advance();
    gc.try_advance();
    gc.try_advance();

    println!("Safe to free epoch 1? {}", gc.is_safe_to_free(epoch1));
}

```

ABA Solutions: 1. **Tagged pointers:** Add version counter to pointer 2. **Double-width CAS:** CAS on (pointer, version) pair 3. **Epoch-based reclamation:** Defer deletion until safe 4. **Hazard pointers:** Track active pointers (next Pattern)

Pattern 3: Lock-Free Queues and Stacks

Problem: Mutex-based data structures serialize all access—threads wait even when operating on different elements. Lock contention causes 80% of multi-threaded time spent waiting.

Solution: Use Treiber stack (lock-free stack with CAS-based push/pop). Implement MPSC queue (multi-producer single-consumer) with atomic operations.

Why It Matters: Lock-free structures enable true parallelism. Multi-threaded counter with Mutex: serialized updates = 1 core performance.

Use Cases: Work-stealing task queues (tokio, rayon), MPMC message passing, real-time audio/video processing, high-frequency trading, concurrent data structure building blocks, actor system mailboxes.

Example: Treiber Stack (Lock-Free Stack)

*A lock-free stack that allows concurrent push/pop operations.

```
use std::sync::atomic::{AtomicPtr, Ordering};
use std::ptr;
use std::sync::Arc;
use std::thread;

struct Node<T> {
    data: T,
    next: *mut Node<T>,
}

pub struct TreiberStack<T> {
    head: AtomicPtr<Node<T>>,
}

impl<T> TreiberStack<T> {
    pub fn new() -> Self {
        Self {
            head: AtomicPtr::new(ptr::null_mut()),
        }
    }

    pub fn push(&self, data: T) {
        let new_node = Box::into_raw(Box::new(Node {
            data,
            next: ptr::null_mut(),
        }));
    }

    loop {
        let head = self.head.load(Ordering::Relaxed);
        unsafe {
            (*new_node).next = head;
        }

        if self
    }
}
```

```

        .head
        .compare_exchange_weak(head, new_node, Ordering::Release, Ordering::Relaxed)
        .is_ok()
    {
        break;
    }
}

pub fn pop(&self) -> Option<T> {
    loop {
        let head = self.head.load(Ordering::Acquire);

        if head.is_null() {
            return None;
        }

        unsafe {
            let next = (*head).next;

            if self
                .head
                .compare_exchange_weak(head, next, Ordering::Release, Ordering::Acquire)
                .is_ok()
            {
                let data = ptr::read(&(*head).data);
                // WARNING: This is unsafe! We should use hazard pointers or epoch-based
                // GC
                // For now, we leak the node to avoid use-after-free
                // drop(Box::from_raw(head));
                return Some(data);
            }
        }
    }
}

pub fn is_empty(&self) -> bool {
    self.head.load(Ordering::Acquire).is_null()
}

unsafe impl<T: Send> Send for TreiberStack<T> {}
unsafe impl<T: Send> Sync for TreiberStack<T> {}
// Real-world: Work-stealing deque (simplified)
pub struct WorkStealingDeque<T> {
    bottom: AtomicPtr<Node<T>>,
    top: AtomicPtr<Node<T>>,
}

impl<T> WorkStealingDeque<T> {
    pub fn new() -> Self {
        Self {
            bottom: AtomicPtr::new(ptr::null_mut()),

```

```

        top: AtomicPtr::new(ptr::null_mut()),
    }

}

pub fn push(&self, data: T) {
    let new_node = Box::into_raw(Box::new(Node {
        data,
        next: ptr::null_mut(),
    }));
}

loop {
    let bottom = self.bottom.load(Ordering::Relaxed);
    unsafe {
        (*new_node).next = bottom;
    }

    if self
        .bottom
        .compare_exchange_weak(bottom, new_node, Ordering::Release, Ordering::Relaxed)
        .is_ok()
    {
        break;
    }
}

pub fn pop(&self) -> Option<T> {
    // Owner pops from bottom (LIFO - better cache locality)
    loop {
        let bottom = self.bottom.load(Ordering::Acquire);

        if bottom.is_null() {
            return None;
        }

        unsafe {
            let next = (*bottom).next;

            if self
                .bottom
                .compare_exchange_weak(bottom, next, Ordering::Release, Ordering::Acquire)
                .is_ok()
            {
                let data = ptr::read(&(*bottom).data);
                return Some(data);
            }
        }
    }
}

pub fn steal(&self) -> Option<T> {
    // Thieves steal from top (FIFO - oldest work)
    loop {

```

```

        let top = self.top.load(Ordering::Acquire);

        if top.is_null() {
            return None;
        }

        unsafe {
            let next = (*top).next;

            if self
                .top
                .compare_exchange_weak(top, next, Ordering::Release, Ordering::Acquire)
                .is_ok()
            {
                let data = ptr::read(&(*top).data);
                return Some(data);
            }
        }
    }
}

unsafe impl<T: Send> Send for WorkStealingDeque<T> {}
unsafe impl<T: Send> Sync for WorkStealingDeque<T> {}

fn main() {
    println!("==== Treiber Stack ====\n");

    let stack = Arc::new(TreiberStack::new());
    let mut handles = vec![];

    // Producers
    for i in 0..5 {
        let stack = Arc::clone(&stack);
        handles.push(thread::spawn(move || {
            for j in 0..100 {
                stack.push(i * 100 + j);
            }
        }));
    }

    // Consumers
    for _ in 0..5 {
        let stack = Arc::clone(&stack);
        handles.push(thread::spawn(move || {
            let mut count = 0;
            while let Some(_) = stack.pop() {
                count += 1;
            }
            count
        }));
    }
}

```

```

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

println!("Stack empty: {}", stack.is_empty());

println!("\n==== Work Stealing Deque ====\n");

let deque = Arc::new(WorkStealingDeque::new());

// Owner thread
let owner_deque = Arc::clone(&deque);
let owner = thread::spawn(move || {
    for i in 0..100 {
        owner_deque.push(i);
    }

    let mut popped = 0;
    while owner_deque.pop().is_some() {
        popped += 1;
    }
    println!("Owner popped: {}", popped);
});

// Thief threads
let mut thieves = vec![];
for id in 0..3 {
    let thief_deque = Arc::clone(&deque);
    thieves.push(thread::spawn(move || {
        thread::sleep(std::time::Duration::from_millis(10));
        let mut stolen = 0;
        while thief_deque.steal().is_some() {
            stolen += 1;
        }
        println!("Thief {} stole: {}", id, stolen);
    }));
}

owner.join().unwrap();
for thief in thieves {
    thief.join().unwrap();
}
}

```

Treiber Stack Properties: - **Lock-free:** At least one thread makes progress - **Push:** O(1) average case - **Pop:** O(1) average case - **ABA problem:** Requires protection (hazard pointers or epoch GC)

Example: Lock-Free Queue (MPSC)

A multi-producer single-consumer lock-free queue.

```
use std::sync::atomic::{AtomicPtr, AtomicBool, Ordering, AtomicUsize};
use std::ptr;
use std::sync::Arc;
use std::thread;

struct QueueNode<T> {
    data: Option<T>,
    next: AtomicPtr<QueueNode<T>>,
}

pub struct MpscQueue<T> {
    head: AtomicPtr<QueueNode<T>>,
    tail: AtomicPtr<QueueNode<T>>,
}

impl<T> MpscQueue<T> {
    pub fn new() -> Self {
        let sentinel = Box::into_raw(Box::new(QueueNode {
            data: None,
            next: AtomicPtr::new(ptr::null_mut()),
        }));
        Self {
            head: AtomicPtr::new(sentinel),
            tail: AtomicPtr::new(sentinel),
        }
    }

    pub fn push(&self, data: T) {
        let new_node = Box::into_raw(Box::new(QueueNode {
            data: Some(data),
            next: AtomicPtr::new(ptr::null_mut()),
        }));
        // Insert at tail
        loop {
            let tail = self.tail.load(Ordering::Acquire);

            unsafe {
                let next = (*tail).next.load(Ordering::Acquire);

                if next.is_null() {
                    // Tail is actually the last node
                    if (*tail)
                        .next
                        .compare_exchange(
                            ptr::null_mut(),
                            new_node,
                            Ordering::Release,
                            Ordering::Acquire,
                        )
                        .is_ok()
                }
            }
        }
    }
}
```

```

    {
        // Try to update tail (optional, helps next push)
        let _ = self.tail.compare_exchange(
            tail,
            new_node,
            Ordering::Release,
            Ordering::Acquire,
        );
        break;
    }
} else {
    // Help other threads by updating tail
    let _ = self.tail.compare_exchange(
        tail,
        next,
        Ordering::Release,
        Ordering::Acquire,
    );
}
}

pub fn pop(&self) -> Option<T> {
    unsafe {
        let head = self.head.load(Ordering::Acquire);
        let next = (*head).next.load(Ordering::Acquire);

        if next.is_null() {
            return None;
        }

        // Move head forward
        self.head.store(next, Ordering::Release);

        // Take data from old sentinel
        let data = (*next).data.take();

        // Drop old sentinel (safe because we're single consumer)
        drop(Box::from_raw(head));

        data
    }
}

unsafe impl<T: Send> Send for MpscQueue<T> {}
unsafe impl<T: Send> Sync for MpscQueue<T> {}

// Real-world: Bounded SPSC queue (Single Producer Single Consumer)
pub struct BoundedSpscQueue<T> {
    buffer: Vec<Option<T>>,
    head: AtomicUsize,
    tail: AtomicUsize,
}

```

```
    capacity: usize,
}

impl<T> BoundedSpscQueue<T> {
    pub fn new(capacity: usize) -> Self {
        let mut buffer = Vec::with_capacity(capacity);
        for _ in 0..capacity {
            buffer.push(None);
        }

        Self {
            buffer,
            head: AtomicUsize::new(0),
            tail: AtomicUsize::new(0),
            capacity,
        }
    }

    pub fn push(&mut self, data: T) -> Result<(), T> {
        let tail = self.tail.load(Ordering::Relaxed);
        let next_tail = (tail + 1) % self.capacity;
        let head = self.head.load(Ordering::Acquire);

        if next_tail == head {
            return Err(data); // Queue full
        }

        unsafe {
            let slot = self.buffer.get_unchecked_mut(tail);
            *slot = Some(data);
        }

        self.tail.store(next_tail, Ordering::Release);
        Ok(())
    }

    pub fn pop(&mut self) -> Option<T> {
        let head = self.head.load(Ordering::Relaxed);
        let tail = self.tail.load(Ordering::Acquire);

        if head == tail {
            return None; // Queue empty
        }

        unsafe {
            let slot = self.buffer.get_unchecked_mut(head);
            let data = slot.take();

            let next_head = (head + 1) % self.capacity;
            self.head.store(next_head, Ordering::Release);

            data
        }
    }
}
```

```

}

pub fn len(&self) -> usize {
    let head = self.head.load(Ordering::Relaxed);
    let tail = self.tail.load(Ordering::Relaxed);

    if tail >= head {
        tail - head
    } else {
        self.capacity - head + tail
    }
}

unsafe impl<T: Send> Send for BoundedSpscQueue<T> {}

fn main() {
    println!("==== MPSC Queue ====\n");

    let queue = Arc::new(MpscQueue::new());

    // Multiple producers
    let mut producers = vec![];
    for i in 0..5 {
        let queue = Arc::clone(&queue);
        producers.push(thread::spawn(move || {
            for j in 0..100 {
                queue.push(i * 100 + j);
            }
        }));
    }

    for p in producers {
        p.join().unwrap();
    }

    // Single consumer
    let mut count = 0;
    while queue.pop().is_some() {
        count += 1;
    }

    println!("Consumed {} items", count);

    println!("\n==== SPSC Queue ====\n");

    let mut producer_queue = BoundedSpscQueue::new(32);
    let mut consumer_queue = unsafe {
        // This is safe because we ensure only one thread accesses each
        std::ptr::read(&producer_queue as *const _)
    };

    let producer = thread::spawn(move || {

```

```

for i in 0..100 {
    while producer_queue.push(i).is_err() {
        thread::yield_now();
    }
}

let consumer = thread::spawn(move || {
    let mut sum = 0;
    let mut received = 0;

    while received < 100 {
        if let Some(value) = consumer_queue.pop() {
            sum += value;
            received += 1;
        } else {
            thread::yield_now();
        }
    }

    sum
});

producer.join().unwrap();
let sum = consumer.join().unwrap();
println!("Sum of 0..100: {}", sum);
}

```

Queue Variants: - **MPSC**: Multi-producer, single-consumer - **SPSC**: Single-producer, single-consumer (fastest) - **MPMC**: Multi-producer, multi-consumer (hardest) - **Bounded**: Fixed size, cache-friendly

Pattern 4: Hazard Pointers

Problem: Lock-free structures need memory reclamation—can't immediately free nodes because other threads might access them. Naive deletion causes use-after-free.

Solution: Use hazard pointers to mark nodes as “in-use”. Each thread announces pointers it's accessing.

Why It Matters: Prevents crashes in production lock-free code. Without proper reclamation, lock-free queue either leaks memory or crashes with use-after-free.

Use Cases: Production lock-free stacks and queues, concurrent hash maps, lock-free linked lists, RCU-style updates, safe memory management without GC, building blocks for complex concurrent data structures.

Example: Hazard Pointer Implementation

Safely reclaim memory in lock-free structures without use-after-free.

```
use std::sync::atomic::{AtomicPtr, AtomicUsize, Ordering};
use std::ptr;
use std::collections::HashSet;

const MAX_HAZARDS: usize = 128;

struct HazardPointer {
    pointer: AtomicPtr<u8>,
}

impl HazardPointer {
    fn new() -> Self {
        Self {
            pointer: AtomicPtr::new(ptr::null_mut()),
        }
    }

    fn protect(&self, ptr: *mut u8) {
        self.pointer.store(ptr, Ordering::Release);
    }

    fn clear(&self) {
        self.pointer.store(ptr::null_mut(), Ordering::Release);
    }

    fn get(&self) -> *mut u8 {
        self.pointer.load(Ordering::Acquire)
    }
}

struct HazardPointerDomain {
    hazards: Vec<HazardPointer>,
    retired: AtomicPtr<RetiredNode>,
    retired_count: AtomicUsize,
}

struct RetiredNode {
    ptr: *mut u8,
    next: *mut RetiredNode,
    deleter: unsafe fn(*mut u8),
}

impl HazardPointerDomain {
    fn new() -> Self {
        let mut hazards = Vec::new();
        for _ in 0..MAX_HAZARDS {
            hazards.push(HazardPointer::new());
        }

        Self {
            hazards,
            retired: AtomicPtr::new(ptr::null_mut()),
```

```

        retired_count: AtomicUsize::new(0),
    }

fn acquire(&self) -> Option<usize> {
    for (i, hp) in self.hazards.iter().enumerate() {
        let current = hp.get();
        if current.is_null() {
            // Try to claim this hazard pointer
            if hp
                .pointer
                .compare_exchange(
                    ptr::null_mut(),
                    1 as *mut u8, // Non-null marker
                    Ordering::Acquire,
                    Ordering::Relaxed,
                )
                .is_ok()
            {
                return Some(i);
            }
        }
    }
    None
}

fn protect(&self, index: usize, ptr: *mut u8) {
    self.hazards[index].protect(ptr);
}

fn release(&self, index: usize) {
    self.hazards[index].clear();
}

fn retire(&self, ptr: *mut u8, deleter: unsafe fn(*mut u8)) {
    let node = Box::into_raw(Box::new(RetiredNode {
        ptr,
        next: ptr::null_mut(),
        deleter,
    }));
    // Add to retired list
    loop {
        let head = self.retired.load(Ordering::Acquire);
        unsafe {
            (*node).next = head;
        }

        if self
            .retired
            .compare_exchange_weak(head, node, Ordering::Release, Ordering::Acquire)
            .is_ok()
        {

```

```

        break;
    }

}

let count = self.retired_count.fetch_add(1, Ordering::Relaxed);

// Trigger reclamation if too many retired
if count > MAX_HAZARDS * 2 {
    self.scan();
}
}

fn scan(&self) {
    // Collect all protected pointers
    let mut protected = HashSet::new();
    for hp in &self.hazards {
        let ptr = hp.get();
        if !ptr.is_null() && ptr != 1 as *mut u8 {
            protected.insert(ptr);
        }
    }
}

// Try to reclaim retired nodes
let mut current = self.retired.swap(ptr::null_mut(), Ordering::Acquire);
let mut kept = Vec::new();

unsafe {
    while !current.is_null() {
        let next = (*current).next;

        if protected.contains(&(*current).ptr) {
            // Still protected, keep it
            kept.push(current);
        } else {
            // Safe to delete
            ((*current).deleter)((*current).ptr);
            drop(Box::from_raw(current));
            self.retired_count.fetch_sub(1, Ordering::Relaxed);
        }

        current = next;
    }
}

// Re-add kept nodes
for node in kept {
    loop {
        let head = self.retired.load(Ordering::Acquire);
        unsafe {
            (*node).next = head;
        }

        if self

```

```

        .retired
        .compare_exchange_weak(head, node, Ordering::Release, Ordering::Acquire)
        .is_ok()
    {
        break;
    }
}
}

// Example: Stack with hazard pointers
struct SafeNode<T> {
    data: T,
    next: *mut SafeNode<T>,
}

struct SafeStack<T> {
    head: AtomicPtr<SafeNode<T>>,
    hp_domain: HazardPointerDomain,
}

impl<T> SafeStack<T> {
    fn new() -> Self {
        Self {
            head: AtomicPtr::new(ptr::null_mut()),
            hp_domain: HazardPointerDomain::new(),
        }
    }

    fn push(&self, data: T) {
        let new_node = Box::into_raw(Box::new(SafeNode {
            data,
            next: ptr::null_mut(),
        }));
    }

    loop {
        let head = self.head.load(Ordering::Relaxed);
        unsafe {
            (*new_node).next = head;
        }

        if self
            .head
            .compare_exchange_weak(head, new_node, Ordering::Release, Ordering::Relaxed)
            .is_ok()
        {
            break;
        }
    }
}

fn pop(&self) -> Option<T> {
    let hp_index = self.hp_domain.acquire()?;

```

```

loop {
    let head = self.head.load(Ordering::Acquire);

    if head.is_null() {
        self.hp_domain.release(hp_index);
        return None;
    }

    // Protect head from deletion
    self.hp_domain.protect(hp_index, head as *mut u8);

    // Verify head hasn't changed (avoid ABA)
    if self.head.load(Ordering::Acquire) != head {
        continue;
    }

    unsafe {
        let next = (*head).next;

        if self
            .head
            .compare_exchange_weak(head, next, Ordering::Release, Ordering::Acquire)
            .is_ok()
        {
            let data = ptr::read(&(*head).data);

            // Retire the node for later deletion
            self.hp_domain.retire(head as *mut u8, |ptr| {
                drop(Box::from_raw(ptr as *mut SafeNode<T>));
            });

            self.hp_domain.release(hp_index);
            return Some(data);
        }
    }
}

unsafe impl<T: Send> Send for SafeStack<T> {}
unsafe impl<T: Send> Sync for SafeStack<T> {}

fn main() {
    println!("==== Safe Stack with Hazard Pointers ===\n");

    let stack = std::sync::Arc::new(SafeStack::new());
    let mut handles = vec![];

    // Producers
    for i in 0..5 {
        let stack = std::sync::Arc::clone(&stack);
        handles.push(std::thread::spawn(move || {

```

```

        for j in 0..1000 {
            stack.push(i * 1000 + j);
        }
    });

// Consumers
for _ in 0..5 {
    let stack = std::sync::Arc::clone(&stack);
    handles.push(std::thread::spawn(move || {
        let mut count = 0;
        while stack.pop().is_some() {
            count += 1;
        }
        count
    }));
}

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

println!("Stack operations completed safely");
}

```

Hazard Pointer Benefits: - **Safe reclamation:** No use-after-free - **Lock-free:** No blocking - **ABA protection:** Version checking via protection - **Memory efficient:** Bounded overhead

Pattern 5: Seqlock Pattern

Problem: Frequent reads of small data with occasional writes. Mutex too expensive (blocks readers).

Solution: Use sequence counter incremented on writes. Writers: increment (odd), write data, increment (even).

Why It Matters: 10-100x faster than locks for read-heavy workloads. Game coordinates updated 60fps, read 10,000x/sec: seqlock enables this.

Use Cases: Game entity positions/state, real-time sensor data, network statistics and metrics, configuration that changes rarely, performance counters, dashboard data, time-series snapshots, read-heavy caches.

Example: Seqlock Implementation

Allow fast, lock-free reads with occasional writes for small data structures.

```

use std::sync::atomic::{AtomicUsize, Ordering};
use std::cell::UnsafeCell;

pub struct SeqLock<T> {

```

```

    seq: AtomicUsize,
    data: UnsafeCell<T>,
}

impl<T: Copy> SeqLock<T> {
    pub fn new(data: T) -> Self {
        Self {
            seq: AtomicUsize::new(0),
            data: UnsafeCell::new(data),
        }
    }

    pub fn read(&self) -> T {
        loop {
            // Read sequence number (even = not writing)
            let seq1 = self.seq.load(Ordering::Acquire);

            if seq1 % 2 == 1 {
                // Writer is active, spin
                std::hint::spin_loop();
                continue;
            }

            // Read data
            let data = unsafe { *self.data.get() };

            // Verify sequence hasn't changed
            std::sync::atomic::fence(Ordering::Acquire);
            let seq2 = self.seq.load(Ordering::Acquire);

            if seq1 == seq2 {
                return data;
            }

            // Sequence changed during read, retry
        }
    }

    pub fn write(&self, data: T) {
        // Increment sequence (makes it odd = writing)
        let seq = self.seq.fetch_add(1, Ordering::Acquire);
        debug_assert!(seq % 2 == 0, "Concurrent writes detected");

        // Write data
        unsafe {
            *self.data.get() = data;
        }

        // Increment again (makes it even = readable)
        self.seq.fetch_add(1, Ordering::Release);
    }

    pub fn try_read(&self) -> Option<T> {

```

```

let seq1 = self.seq.load(Ordering::Acquire);

if seq1 % 2 == 1 {
    return None; // Writer active
}

let data = unsafe { *self.data.get() };

std::sync::atomic::fence(Ordering::Acquire);
let seq2 = self.seq.load(Ordering::Acquire);

if seq1 == seq2 {
    Some(data)
} else {
    None // Data changed
}
}

unsafe impl<T: Copy + Send> Send for SeqLock<T> {}
unsafe impl<T: Copy + Send> Sync for SeqLock<T> {}

// Real-world: Coordinates with seqlock
#[derive(Copy, Clone, Debug)]
struct Coordinates {
    x: f64,
    y: f64,
    z: f64,
}

fn seqlock_coordinates_example() {
    use std::sync::Arc;
    use std::thread;
    use std::time::Duration;

    let position = Arc::new(SeqLock::new(Coordinates {
        x: 0.0,
        y: 0.0,
        z: 0.0,
    }));

    // Writer thread (updates position)
    let writer_pos = Arc::clone(&position);
    let writer = thread::spawn(move || {
        for i in 0..100 {
            let coords = Coordinates {
                x: i as f64,
                y: (i * 2) as f64,
                z: (i * 3) as f64,
            };
            writer_pos.write(coords);
            thread::sleep(Duration::from_millis(10));
        }
    });
}

```

```

// Reader threads (read position frequently)
let mut readers = vec![];
for id in 0..5 {
    let reader_pos = Arc::clone(&position);
    readers.push(thread::spawn(move || {
        for _ in 0..1000 {
            let coords = reader_pos.read();
            if id == 0 && coords.x as usize % 10 == 0 {
                println!("Reader {}: {:?}", id, coords);
            }
        }
    }));
}

writer.join().unwrap();
for reader in readers {
    reader.join().unwrap();
}
}

// Real-world: Statistics snapshot
#[derive(Copy, Clone, Debug)]
struct Stats {
    count: u64,
    sum: u64,
    min: u64,
    max: u64,
}

impl Stats {
    fn new() -> Self {
        Self {
            count: 0,
            sum: 0,
            min: u64::MAX,
            max: 0,
        }
    }

    fn add(&mut self, value: u64) {
        self.count += 1;
        self.sum += value;
        self.min = self.min.min(value);
        self.max = self.max.max(value);
    }

    fn average(&self) -> f64 {
        if self.count == 0 {
            0.0
        } else {
            self.sum as f64 / self.count as f64
        }
    }
}

```

```
}
```

```
fn seqlock_stats_example() {
    use std::sync::Arc;
    use std::thread;

    let stats = Arc::new(SeqLock::new(Stats::new()));

    // Writer thread
    let writer_stats = Arc::clone(&stats);
    let writer = thread::spawn(move || {
        for i in 0..1000 {
            let mut current = writer_stats.read();
            current.add(i);
            writer_stats.write(current);
        }
    });

    // Reader threads (monitor stats)
    let mut readers = vec![];
    for id in 0..3 {
        let reader_stats = Arc::clone(&stats);
        readers.push(thread::spawn(move || {
            for _ in 0..100 {
                thread::sleep(std::time::Duration::from_millis(10));
                let snapshot = reader_stats.read();
                if id == 0 {
                    println!(
                        "Stats - Count: {}, Avg: {:.2}, Min: {}, Max: {}",
                        snapshot.count,
                        snapshot.average(),
                        snapshot.min,
                        snapshot.max
                    );
                }
            }
        }));
    }

    writer.join().unwrap();
    for reader in readers {
        reader.join().unwrap();
    }
}
```

Example: Versioned seqlock (track writes)

This example shows versioned seqlock (track writes) to illustrate where the pattern fits best.

```
pub struct VersionedSeqLock<T> {
    seqlock: SeqLock<T>,
```

```

}

impl<T: Copy> VersionedSeqLock<T> {
    pub fn new(data: T) -> Self {
        Self {
            seqlock: SeqLock::new(data),
        }
    }

    pub fn read_with_version(&self) -> (T, usize) {
        let seq1 = self.seqlock.seq.load(Ordering::Acquire);
        let data = self.seqlock.read();
        let version = seq1 / 2;
        (data, version)
    }

    pub fn write(&self, data: T) {
        self.seqlock.write(data);
    }

    pub fn version(&self) -> usize {
        self.seqlock.seq.load(Ordering::Acquire) / 2
    }
}

fn main() {
    println!("==== Seqlock Coordinates ====\n");
    seqlock_coordinates_example();

    println!("==== Seqlock Statistics ====\n");
    seqlock_stats_example();

    println!("==== Versioned Seqlock ====\n");

    let data = VersionedSeqLock::new(0u64);

    for i in 0..5 {
        data.write(i * 10);
        let (value, version) = data.read_with_version();
        println!("Value: {}, Version: {}", value, version);
    }
}

```

Seqlock Characteristics: - **Optimistic reads:** No locks for readers - **Single writer:** Only one writer at a time - **Small data:** Works best with Copy types - **Retry on write:** Readers retry if writer was active - **Use case:** Frequently read, rarely written data (coordinates, stats)

Example: Advanced Atomic Patterns

Specialized concurrent patterns using atomics.

```
use std::sync::atomic::{AtomicU64, AtomicUsize, AtomicBool, Ordering};
use std::sync::Arc;
use std::thread;
use std::time::{Duration, Instant};
```

Example: Striped counter (reduce contention)

This example shows how to striped counter (reduce contention) while calling out the practical trade-offs.

```
struct StripedCounter {
    stripes: Vec<AtomicUsize>,
}

impl StripedCounter {
    fn new(num_stripes: usize) -> Self {
        let mut stripes = Vec::new();
        for _ in 0..num_stripes {
            stripes.push(AtomicUsize::new(0));
        }

        Self { stripes }
    }

    fn increment(&self) {
        let thread_id = std::thread::current().id();
        let index = format!("{:?}", thread_id).len() % self.stripes.len();
        self.stripes[index].fetch_add(1, Ordering::Relaxed);
    }

    fn get(&self) -> usize {
        self.stripes
            .iter()
            .map(|s| s.load(Ordering::Relaxed))
            .sum()
    }
}
```

Example: Exponential backoff

This example shows how to exponential backoff in practice, emphasizing why it works.

```
struct Backoff {
    current: Duration,
    max: Duration,
}

impl Backoff {
    fn new() -> Self {
```

```

Self {
    current: Duration::from_nanos(1),
    max: Duration::from_micros(1000),
}
}

fn spin(&mut self) {
    for _ in 0..(self.current.as_nanos() / 10) {
        std::hint::spin_loop();
    }

    self.current = (self.current * 2).min(self.max);
}

fn reset(&mut self) {
    self.current = Duration::from_nanos(1);
}
}

fn cas_with_backoff(counter: &AtomicUsize) {
    let mut backoff = Backoff::new();

    loop {
        let current = counter.load(Ordering::Relaxed);

        match counter.compare_exchange_weak(
            current,
            current + 1,
            Ordering::Relaxed,
            Ordering::Relaxed,
        ) {
            Ok(_) => {
                backoff.reset();
                break;
            }
            Err(_) => {
                backoff.spin();
            }
        }
    }
}

```

Example: Atomic min/max

This example shows how to atomic min/max while calling out the practical trade-offs.

```

struct AtomicMinMax {
    min: AtomicU64,
    max: AtomicU64,
}

```

```

impl AtomicMinMax {
    fn new() -> Self {
        Self {
            min: AtomicU64::new(u64::MAX),
            max: AtomicU64::new(0),
        }
    }

    fn update(&self, value: u64) {
        // Update min
        let mut current_min = self.min.load(Ordering::Relaxed);
        while value < current_min {
            match self.min.compare_exchange_weak(
                current_min,
                value,
                Ordering::Relaxed,
                Ordering::Relaxed,
            ) {
                Ok(_) => break,
                Err(actual) => current_min = actual,
            }
        }
    }

    // Update max
    let mut current_max = self.max.load(Ordering::Relaxed);
    while value > current_max {
        match self.max.compare_exchange_weak(
            current_max,
            value,
            Ordering::Relaxed,
            Ordering::Relaxed,
        ) {
            Ok(_) => break,
            Err(actual) => current_max = actual,
        }
    }
}

fn get(&self) -> (u64, u64) {
    (
        self.min.load(Ordering::Relaxed),
        self.max.load(Ordering::Relaxed),
    )
}
}

```

Example: Once flag for initialization

This example shows how to once flag for initialization in practice, emphasizing why it works.

```

struct OnceFlag {

```

```

        state: AtomicUsize,
    }

const INCOMPLETE: usize = 0;
const RUNNING: usize = 1;
const COMPLETE: usize = 2;

impl OnceFlag {
    fn new() -> Self {
        Self {
            state: AtomicUsize::new(INCOMPLETE),
        }
    }

    fn call_once<F>(&self, f: F)
    where
        F: FnOnce(),
    {
        if self.state.load(Ordering::Acquire) == COMPLETE {
            return;
        }

        match self.state.compare_exchange(
            INCOMPLETE,
            RUNNING,
            Ordering::Acquire,
            Ordering::Acquire,
        ) {
            Ok(_) => {
                // We won the race
                f();
                self.state.store(COMPLETE, Ordering::Release);
            }
            Err(RUNNING) => {
                // Someone else is running, wait
                while self.state.load(Ordering::Acquire) == RUNNING {
                    std::hint::spin_loop();
                }
            }
            Err(COMPLETE) => {
                // Already done
            }
            _ => unreachable!(),
        }
    }

    fn is_completed(&self) -> bool {
        self.state.load(Ordering::Acquire) == COMPLETE
    }
}

```

Example: Atomic swap chain

This example shows how to atomic swap chain while calling out the practical trade-offs.

```
struct SwapChain<T> {
    value: AtomicUsize, // Actually *mut T
    _phantom: std::marker::PhantomData<T>,
}

impl<T> SwapChain<T> {
    fn new(initial: T) -> Self {
        let ptr = Box::into_raw(Box::new(initial));
        Self {
            value: AtomicUsize::new(ptr as usize),
            _phantom: std::marker::PhantomData,
        }
    }

    fn swap(&self, new_value: T) -> T {
        let new_ptr = Box::into_raw(Box::new(new_value));
        let old_ptr = self.value.swap(new_ptr as usize, Ordering::AcqRel) as *mut T;

        unsafe {
            let old_value = std::ptr::read(old_ptr);
            drop(Box::from_raw(old_ptr));
            old_value
        }
    }

    fn load(&self) -> T
    where
        T: Clone,
    {
        let ptr = self.value.load(Ordering::Acquire) as *mut T;
        unsafe { (*ptr).clone() }
    }
}

impl<T> Drop for SwapChain<T> {
    fn drop(&mut self) {
        let ptr = self.value.load(Ordering::Acquire) as *mut T;
        if !ptr.is_null() {
            unsafe {
                drop(Box::from_raw(ptr));
            }
        }
    }
}

fn main() {
    println!("== Striped Counter ==\n");
}
```

```

let counter = Arc::new(StripedCounter::new(16));
let mut handles = vec![];

let start = Instant::now();

for _ in 0..10 {
    let counter = Arc::clone(&counter);
    handles.push(thread::spawn(move || {
        for _ in 0..100_000 {
            counter.increment();
        }
    }));
}

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

println!("Count: {} in {:?}", counter.get(), start.elapsed());

println!("\n==== Backoff ===\n");

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

for _ in 0..10 {
    let counter = Arc::clone(&counter);
    handles.push(thread::spawn(move || {
        for _ in 0..10_000 {
            cas_with_backoff(&counter);
        }
    }));
}

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

println!("Count: {}", counter.load(Ordering::Relaxed));

println!("\n==== Atomic Min/Max ===\n");

let minmax = Arc::new(AtomicMinMax::new());
let mut handles = vec![];

for i in 0..10 {
    let minmax = Arc::clone(&minmax);
    handles.push(thread::spawn(move || {
        for j in 0..1000 {
            minmax.update(i * 1000 + j);
        }
    }));
}

```

```

}

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

let (min, max) = minmax.get();
println!("Min: {}, Max: {}", min, max);

println!("\n==== Once Flag ====\n");

let once = Arc::new(OnceFlag::new());
let mut handles = vec![];

for i in 0..10 {
    let once = Arc::clone(&once);
    handles.push(thread::spawn(move || {
        once.call_once(|| {
            println!("Initialization by thread {}", i);
        });
    }));
}

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

println!("Completed: {}", once.is_completed());
}

```

Summary

This chapter covered atomic operations and lock-free programming:

1. **Memory Ordering**: Relaxed, Acquire/Release, AcqRel, SeqCst with use cases
2. **Compare-and-Swap**: CAS loops, weak vs strong, ABA problem and solutions
3. **Lock-Free Structures**: Treiber stack, MPSC/SPSC queues, work-stealing deques
4. **Hazard Pointers**: Safe memory reclamation without garbage collection
5. **Seqlock**: Optimistic reads for small, frequently-read data

Key Takeaways: - **Memory ordering** is critical for correctness - **Relaxed** for counters, **Acquire/Release** for synchronization, **SeqCst** for simplicity - **CAS** is the foundation of lock-free algorithms - **ABA problem** requires version counters or hazard pointers - **Lock-free** != faster always (measure performance) - **Seqlock** excels for read-heavy small data

Performance Guidelines: - Use **Relaxed** when order doesn't matter (fastest) - **Acquire/Release** for most synchronization (good balance) - **SeqCst** when correctness is critical (slowest) - Striped counters reduce contention - Backoff reduces CPU waste during contention - Lock-free shines under high contention

Common Pitfalls: - Wrong memory ordering (too weak = race, too strong = slow) - ABA problem (use versioning or hazard pointers) - Memory leaks (need reclamation strategy) - Assuming lock-free = faster (profile!) - Over-using atomics (sometimes locks are simpler)

When to Use: - **Atomics:** Counters, flags, simple synchronization - **Lock-free:** High contention, real-time constraints - **Locks:** Complex operations, simplicity, most cases - **Seqlock:** Coordinates, stats, small frequently-read data

Safety: - Rust's type system prevents most data races - Atomic operations are safe - Raw pointers in lock-free structures require unsafe - Use existing libraries (crossbeam) when possible

Parallel Algorithms

This chapter explores parallel patterns using Rust's ecosystem, focusing on data parallelism with Rayon, work partitioning strategies, parallel reduction patterns, pipeline parallelism, and SIMD vectorization. We'll cover practical, production-ready examples for maximizing CPU utilization.

Pattern 1: Rayon Patterns

Problem: Sequential code wastes modern multi-core CPUs—a 4-core CPU runs sequential code at 25% utilization. Manual threading with `std::thread` is complex: need to partition data, spawn threads, collect results, handle errors, avoid data races.

Solution: Use Rayon's parallel iterators with `.par_iter()`. Rayon provides work-stealing scheduler that automatically balances load across threads.

Why It Matters: Trivial code change yields massive speedups. Image processing 1M pixels: sequential 500ms, parallel 80ms (6x faster on 8-core).

Use Cases: Image processing (grayscale, filters, resizing), data validation (emails, phone numbers, formats), log parsing and analysis, batch data transformations, scientific computing, map-reduce operations, any embarrassingly parallel workload.

Example: Parallel Iterator Basics

Convert sequential operations to parallel execution with minimal code changes.

```
// Note: Add to Cargo.toml:  
// rayon = "1.8"  
  
use rayon::prelude::*;

use std::time::Instant;
```

Example: Basic par_iter

This example walks through the basics of `par_iter`, highlighting each step so you can reuse the pattern.

```
fn parallel_map_example() {
```

```

let numbers: Vec<i32> = (0..1_000_000).collect();

// Sequential
let start = Instant::now();
let sequential: Vec<i32> = numbers.iter().map(|&x| x * x).collect();
let seq_time = start.elapsed();

// Parallel
let start = Instant::now();
let parallel: Vec<i32> = numbers.par_iter().map(|&x| x * x).collect();
let par_time = start.elapsed();

println!("Sequential: {:?}", seq_time);
println!("Parallel: {:?}", par_time);
println!("Speedup: {:.2}x", seq_time.as_secs_f64() / par_time.as_secs_f64());

assert_eq!(sequential, parallel);
}

```

Example: par_iter vs par_iter_mut vs into_par_iter

This example shows how to par_iter vs par_iter_mut vs into_par_iter in practice, emphasizing why it works.

```

fn iterator_variants() {
    let mut data = vec![1, 2, 3, 4, 5];

    // Immutable parallel iteration
    let sum: i32 = data.par_iter().sum();
    println!("Sum: {}", sum);

    // Mutable parallel iteration
    data.par_iter_mut().for_each(|x| *x *= 2);
    println!("Doubled: {:?}", data);

    // Consuming parallel iteration
    let owned_data = vec![1, 2, 3, 4, 5];
    let squares: Vec<i32> = owned_data.into_par_iter().map(|x| x * x).collect();
    println!("Squares: {:?}", squares);
}

```

Example: Parallel filter and map

This example shows how to parallel filter and map while calling out the practical trade-offs.

```

fn parallel_filter_map() {
    let numbers: Vec<i32> = (0..10_000).collect();

    let result: Vec<i32> = numbers

```

```

    .par_iter()
    .filter(|&&x| x % 2 == 0)
    .map(|x| x * x)
    .collect();

    println!("Filtered and squared {} numbers", result.len());
}

```

Example: Parallel flat_map

This example shows how to parallel flat_map while calling out the practical trade-offs.

```

fn parallel_flat_map() {
    let ranges: Vec<std::ops::Range<i32>> = vec![0..10, 10..20, 20..30];

    let flattened: Vec<i32> = ranges
        .into_par_iter()
        .flat_map(|range| range.into_par_iter())
        .collect();

    println!("Flattened {} items", flattened.len());
}

// Real-world: Image processing
struct Image {
    pixels: Vec<u8>,
    width: usize,
    height: usize,
}

impl Image {
    fn new(width: usize, height: usize) -> Self {
        Self {
            pixels: vec![128; width * height * 3], // RGB
            width,
            height,
        }
    }

    fn apply_filter_parallel(&mut self, filter: fn(u8) -> u8) {
        self.pixels.par_iter_mut().for_each(|pixel| {
            *pixel = filter(*pixel);
        });
    }

    fn grayscale_parallel(&mut self) {
        self.pixels
            .par_chunks_mut(3)
            .for_each(|rgb| {
                let gray = ((rgb[0] as u32 + rgb[1] as u32 + rgb[2] as u32) / 3) as u8;
                rgb[0] = gray;
                rgb[1] = gray;
            });
    }
}

```

```

        rgb[2] = gray;
    });
}

fn brightness_parallel(&mut self, delta: i16) {
    self.pixels.par_iter_mut().for_each(|pixel| {
        *pixel = (*pixel as i16 + delta).clamp(0, 255) as u8;
    });
}
}

// Real-world: Data validation
fn validate_emails_parallel(emails: Vec<String>) -> Vec<(String, bool)> {
    emails
        .into_par_iter()
        .map(|email| {
            let is_valid = email.contains('@') && email.contains('.');
            (email, is_valid)
        })
        .collect()
}
}

// Real-world: Log parsing
#[derive(Debug)]
struct LogEntry {
    timestamp: u64,
    level: String,
    message: String,
}
}

fn parse_logs_parallel(lines: Vec<String>) -> Vec<LogEntry> {
    lines
        .into_par_iter()
        .filter_map(|line| {
            let parts: Vec<&str> = line.split('|').collect();
            if parts.len() >= 3 {
                Some(LogEntry {
                    timestamp: parts[0].parse().ok()?,
                    level: parts[1].to_string(),
                    message: parts[2].to_string(),
                })
            } else {
                None
            }
        })
        .collect()
}
}

fn main() {
    println!("==== Parallel Map ====\n");
    parallel_map_example();

    println!("\n==== Iterator Variants ====\n");
    iterator_variants();
}

```

```

println!("==== Filter Map ====");
parallel_filter_map();

println!("==== Image Processing ====");
let mut img = Image::new(1920, 1080);

let start = Instant::now();
img.grayscale_parallel();
println!("Grayscale: {:?}", start.elapsed());

let start = Instant::now();
img.brightness_parallel(10);
println!("Brightness: {:?}", start.elapsed());

println!("==== Email Validation ====");
let emails = vec![
    "user@example.com".to_string(),
    "invalid-email".to_string(),
    "another@test.org".to_string(),
];
let results = validate_emails_parallel(emails);
for (email, valid) in results {
    println!("{}: {}", email, if valid { "✓" } else { "✗" });
}
}

```

Rayon Benefits: - **Minimal code changes:** Add `.par_` prefix - **Work stealing:** Automatic load balancing - **Type safe:** Same API as sequential iterators - **No data races:** Enforced by type system

Example: par_bridge for Dynamic Sources

Parallelize iterators that don't implement `ParallelIterator` directly, or process items as they arrive.

```

use rayon::prelude::*;
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

```

Example: Bridge from sequential iterator

This example shows how to bridge from sequential iterator in practice, emphasizing why it works.

```

fn par_bridge_basic() {
    let iter = (0..1000).filter(|x| x % 2 == 0);

    // Bridge to parallel
    let sum: i32 = iter.par_bridge().map(|x| x * x).sum();
}

```

```
    println!("Sum: {}", sum);
}
```

Example: Bridge from channel receiver

This example shows how to bridge from channel receiver in practice, emphasizing why it works.

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

    // Producer thread
    thread::spawn(move || {
        for i in 0..1000 {
            tx.send(i).unwrap();
            thread::sleep(Duration::from_micros(10));
        }
    });

    // Parallel processing of channel items
    let sum: i32 = rx
        .into_iter()
        .par_bridge()
        .map(|x| {
            // Expensive computation
            thread::sleep(Duration::from_micros(100));
            x * x
        })
        .sum();

    println!("Channel sum: {}", sum);
}

// Real-world: File system traversal
use std::fs;
use std::path::PathBuf;

fn find_large_files_parallel(root: &str, min_size: u64) -> Vec<PathBuf, u64> {
    fn visit_dirs(path: PathBuf) -> Box<dyn Iterator<Item = PathBuf>> {
        let entries = match fs::read_dir(&path) {
            Ok(entries) => entries,
            Err(_) => return Box::new(std::iter::empty()),
        };

        let iter = entries.filter_map(|e| e.ok()).flat_map(|entry| {
            let path = entry.path();
            if path.is_dir() {
                visit_dirs(path)
            } else {
                Box::new(std::iter::once(path))
            }
        });
        Box::new(iter)
    }

    visit_dirs(root)
}
```

```

    Box::new(iter)
}

visit_dirs(PathBuf::from(root))
    .par_bridge()
    .filter_map(|path| {
        let metadata = fs::metadata(&path).ok()?;
        let size = metadata.len();
        if size >= min_size {
            Some((path, size))
        } else {
            None
        }
    })
    .collect()
}

// Real-world: Database query results
struct DatabaseIterator {
    current: usize,
    total: usize,
}

impl Iterator for DatabaseIterator {
    type Item = i32;

    fn next(&mut self) -> Option<Self::Item> {
        if self.current < self.total {
            let value = self.current as i32;
            self.current += 1;
            // Simulate database fetch delay
            thread::sleep(Duration::from_millis(10));
            Some(value)
        } else {
            None
        }
    }
}

fn process_database_results() {
    let db_iter = DatabaseIterator {
        current: 0,
        total: 1000,
    };

    // Process results in parallel as they arrive
    let sum: i32 = db_iter
        .par_bridge()
        .map(|x| x * 2)
        .sum();

    println!("Database result sum: {}", sum);
}

// Real-world: Network stream processing

```

```

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

    // Simulate network packets arriving
    thread::spawn(move || {
        for i in 0..100 {
            let packet = format!("packet_{}", i);
            tx.send(packet).unwrap();
            thread::sleep(Duration::from_millis(5));
        }
    });
}

// Process packets in parallel
let processed: Vec<String> = rx
    .into_iter()
    .par_bridge()
    .map(|packet| {
        // Expensive processing (e.g., parsing, validation)
        thread::sleep(Duration::from_millis(10));
        format!("processed_{}", packet)
    })
    .collect();

println!("Processed {} packets", processed.len());
}

fn main() {
    println!("== par_bridge Basic ==\n");
    par_bridge_basic();

    println!("\n== par_bridge from Channel ==\n");
    par_bridge_from_channel();

    println!("\n== Database Iterator ==\n");
    process_database_results();

    println!("\n== Network Stream ==\n");
    process_network_stream();
}

```

par_bridge Use Cases: - **Channel receivers:** Process items as they arrive - **Custom iterators:** Database cursors, file system traversal - **Lazy evaluation:** Only compute when needed - **Adaptive parallelism:** Work stealing adapts to varying workload

Pattern 2: Work Partitioning Strategies

Problem: Bad work partitioning kills parallel performance. Too-small chunks (100 items) cause overhead—thread spawn/join costs dominate.

Solution: Choose grain size based on work: 1K-10K items for simple operations, larger for cache-heavy work. Use Rayon's adaptive chunking (default) for uniform work, explicit chunking for cache optimization.

Why It Matters: Grain size tuning: 2-3x performance difference. Matrix multiply with blocking: 5x faster due to cache hits.

Use Cases: Matrix operations (multiply, transpose, factorization), sorting algorithms (quicksort, mergesort), divide-and-conquer (tree traversal, expression evaluation), irregular workloads (graph algorithms), cache-sensitive operations (blocked algorithms).

Example: Chunking and Load Balancing

Partition work efficiently across threads to minimize overhead and maximize CPU utilization.

```
use rayon::prelude::*;
use std::time::Instant;
```

Example: Chunk sizes

This example shows how to chunk sizes in practice, emphasizing why it works.

```
fn chunk_size_comparison() {
    let data: Vec<i32> = (0..1_000_000).collect();

    // Default chunking (Rayon decides)
    let start = Instant::now();
    let sum1: i32 = data.par_iter().sum();
    let default_time = start.elapsed();

    // Custom chunk size (too small - more overhead)
    let start = Instant::now();
    let sum2: i32 = data.par_chunks(100).map(|chunk| chunk.iter().sum::()).sum();
    let small_chunk_time = start.elapsed();

    // Custom chunk size (balanced)
    let start = Instant::now();
    let sum3: i32 = data.par_chunks(10_000).map(|chunk| chunk.iter().sum::()).sum();
    let balanced_chunk_time = start.elapsed();

    println!("Default: {:?}", default_time);
    println!("Small chunks (100): {:?}", small_chunk_time);
    println!("Balanced chunks (10k): {:?}", balanced_chunk_time);

    assert_eq!(sum1, sum2);
    assert_eq!(sum2, sum3);
}
```

Example: Work splitting strategies

This example shows how to work splitting strategies in practice, emphasizing why it works.

```
fn work_splitting_strategies() {
    let data: Vec<i32> = (0..100_000).collect();

    // Strategy 1: Equal splits (good for uniform work)
    let chunk_size = data.len() / rayon::current_num_threads();
    let result1: Vec<i32> = data
        .par_chunks(chunk_size)
        .flat_map(|chunk| chunk.iter().map(|&x| x * x))
        .collect();

    // Strategy 2: Adaptive (good for non-uniform work)
    let result2: Vec<i32> = data.par_iter().map(|&x| x * x).collect();

    assert_eq!(result1.len(), result2.len());
}

// Real-world: Matrix multiplication with blocking
struct Matrix {
    data: Vec<f64>,
    rows: usize,
    cols: usize,
}

impl Matrix {
    fn new(rows: usize, cols: usize) -> Self {
        Self {
            data: vec![0.0; rows * cols],
            rows,
            cols,
        }
    }

    fn get(&self, row: usize, col: usize) -> f64 {
        self.data[row * self.cols + col]
    }

    fn set(&mut self, row: usize, col: usize, value: f64) {
        self.data[row * self.cols + col] = value;
    }

    // Parallel matrix multiplication with blocking for cache efficiency
    fn multiply_blocked(&self, other: &Matrix, block_size: usize) -> Matrix {
        assert_eq!(self.cols, other.rows);

        let mut result = Matrix::new(self.rows, other.cols);

        // Partition work by output blocks
        let row_blocks: Vec<usize> = (0..self.rows).step_by(block_size).collect();
```

```

let col_blocks: Vec<u8> = (0..other.cols).step_by(block_size).collect();

row_blocks.par_iter().for_each(|&row_start| {
    for &col_start in &col_blocks {
        // Process block
        let row_end = (row_start + block_size).min(self.rows);
        let col_end = (col_start + block_size).min(other.cols);

        for i in row_start..row_end {
            for j in col_start..col_end {
                let mut sum = 0.0;
                for k in 0..self.cols {
                    sum += self.get(i, k) * other.get(k, j);
                }
                unsafe {
                    let ptr = result.data.as_mut_ptr() as *mut f64;
                    *ptr.add(i * result.cols + j) = sum;
                }
            }
        }
    });
});

result
}
}

// Real-world: Parallel merge sort with optimal grain size
fn parallel_merge_sort<T: Ord + Send>(arr: &mut [T], grain_size: u8) {
    if arr.len() <= grain_size {
        arr.sort();
        return;
    }

    let mid = arr.len() / 2;
    let (left, right) = arr.split_at_mut(mid);

    rayon::join(
        || parallel_merge_sort(left, grain_size),
        || parallel_merge_sort(right, grain_size),
    );

    // Merge (in-place merge omitted for brevity)
    let mut temp = Vec::with_capacity(arr.len());
    let mut i = 0;
    let mut j = mid;

    while i < mid && j < arr.len() {
        if arr[i] <= arr[j] {
            temp.push(std::mem::replace(&mut arr[i], unsafe { std::ptr::read(&arr[0]) }));
            i += 1;
        } else {
            temp.push(std::mem::replace(&mut arr[j], unsafe { std::ptr::read(&arr[0]) }));
            j += 1;
        }
    }

    arr.copy_from_slice(&temp);
}
}

```

```

        }

    }

    while i < mid {
        temp.push(std::mem::replace(&mut arr[i], unsafe { std::ptr::read(&arr[0]) }));
        i += 1;
    }

    while j < arr.len() {
        temp.push(std::mem::replace(&mut arr[j], unsafe { std::ptr::read(&arr[0]) }));
        j += 1;
    }

    for (i, item) in temp.into_iter().enumerate() {
        arr[i] = item;
    }
}

```

Example: Dynamic load balancing

This example shows how to dynamic load balancing in practice, emphasizing why it works.

```

fn dynamic_load_balancing() {
    // Simulate irregular workload
    let work_items: Vec<u32> = (0..1000).map(|i| i % 100).collect();

    let start = Instant::now();

    // Rayon automatically balances work through work stealing
    let results: Vec<u32> = work_items
        .par_iter()
        .map(|&work| {
            // Simulate variable work
            let mut sum = 0;
            for _ in 0..work {
                sum += 1;
            }
            sum
        })
        .collect();

    println!("Dynamic balancing: {:?}", start.elapsed());
    println!("Total work: {}", results.iter().sum::());
}

```

Example: Grain size tuning

This example shows how to grain size tuning in practice, emphasizing why it works.

```

fn grain_size_tuning() {
    let data: Vec<i32> = (0..1_000_000).collect();

    for grain_size in [100, 1_000, 10_000, 100_000] {
        let start = Instant::now();

        let sum: i32 = data
            .par_chunks(grain_size)
            .map(|chunk| chunk.iter().sum::<i32>())
            .sum();

        println!("Grain size {}: {:?}", grain_size, start.elapsed());
    }
}

fn main() {
    println!("== Chunk Size Comparison ==\n");
    chunk_size_comparison();

    println!("\n== Dynamic Load Balancing ==\n");
    dynamic_load_balancing();

    println!("\n== Grain Size Tuning ==\n");
    grain_size_tuning();

    println!("\n== Parallel Merge Sort ==\n");
    let mut data: Vec<i32> = (0..100_000).rev().collect();
    let start = Instant::now();
    parallel_merge_sort(&mut data, 1000);
    println!("Sort time: {:?}", start.elapsed());
    println!("Sorted: {}", data.windows(2).all(|w| w[0] <= w[1]));
}

```

Work Partitioning Guidelines: - **Grain size:** Larger grains reduce overhead, but may cause load imbalance - **Chunk size:** Balance overhead vs parallelism (typically 1000-10000 items) - **Cache blocking:** Improve cache locality with block-based partitioning - **Work stealing:** Rayon automatically balances irregular workloads

Example: Recursive Parallelism

Parallelize divide-and-conquer algorithms efficiently.

```
use rayon::prelude::*;


```

Example: Parallel quicksort

This example shows how to parallel quicksort while calling out the practical trade-offs.

```

fn parallel_quicksort<T: Ord + Send>(arr: &mut [T]) {
    if arr.len() <= 1 {
        return;
    }

    let pivot_idx = partition(arr);

    let (left, right) = arr.split_at_mut(pivot_idx);

    // Parallelize recursion
    rayon::join(
        || parallel_quicksort(left),
        || parallel_quicksort(&mut right[1..]),
    );
}

fn partition<T: Ord>(arr: &mut [T]) -> usize {
    let len = arr.len();
    let pivot_idx = len / 2;
    arr.swap(pivot_idx, len - 1);

    let mut i = 0;
    for j in 0..len - 1 {
        if arr[j] <= arr[len - 1] {
            arr.swap(i, j);
            i += 1;
        }
    }

    arr.swap(i, len - 1);
    i
}

```

Example: Parallel tree traversal

This example shows how to parallel tree traversal while calling out the practical trade-offs.

```

#[derive(Debug)]
struct TreeNode<T> {
    value: T,
    left: Option<Box<TreeNode<T>>>,
    right: Option<Box<TreeNode<T>>>,
}

impl<T: Send> TreeNode<T> {
    fn parallel_map<F, U>(&self, f: &F) -> TreeNode<U>
    where
        F: Fn(&T) -> U + Sync,
        U: Send,
}

```

```

{
    let value = f(&self.value);

    let (left, right) = rayon::join(
        || self.left.as_ref().map(|node| Box::new(node.parallel_map(f))),
        || self.right.as_ref().map(|node| Box::new(node.parallel_map(f))),
    );
}

TreeNode { value, left, right }
}

fn parallel_sum(&self) -> T
where
    T: std::ops::Add<Output = T> + Default + Copy + Send,
{
    let mut sum = self.value;

    let (left_sum, right_sum) = rayon::join(
        || self.left.as_ref().map_or(T::default(), |node| node.parallel_sum()),
        || self.right.as_ref().map_or(T::default(), |node| node.parallel_sum()),
    );

    sum = sum + left_sum + right_sum;
    sum
}
}

```

Example: Parallel Fibonacci (demonstrative, not efficient)

This example shows how to parallel Fibonacci (demonstrative, not efficient) while calling out the practical trade-offs.

```

fn parallel_fib(n: u32) -> u64 {
    if n <= 1 {
        return n as u64;
    }

    if n < 20 {
        // Sequential threshold to avoid overhead
        return fib_sequential(n);
    }

    let (a, b) = rayon::join(
        || parallel_fib(n - 1),
        || parallel_fib(n - 2),
    );

    a + b
}

fn fib_sequential(n: u32) -> u64 {

```

```

if n <= 1 {
    return n as u64;
}
let mut a = 0;
let mut b = 1;
for _ in 2..=n {
    let c = a + b;
    a = b;
    b = c;
}
b
}

// Real-world: Parallel directory size calculation
use std::fs;
use std::path::Path;

fn parallel_dir_size<P: AsRef

```

```

                || right.parallel_eval(),
            );
            l + r
        }
    Expr::Mul(left, right) => {
        let (l, r) = rayon::join(
            || left.parallel_eval(),
            || right.parallel_eval(),
        );
        l * r
    }
    Expr::Sub(left, right) => {
        let (l, r) = rayon::join(
            || left.parallel_eval(),
            || right.parallel_eval(),
        );
        l - r
    }
}
}

fn main() {
    println!("== Parallel Quicksort ==\n");
    let mut data: Vec<i32> = (0..100_000).rev().collect();
    let start = std::time::Instant::now();
    parallel_quicksort(&mut data);
    println!("Sort time: {:?}", start.elapsed());
    println!("Sorted: {}", data.windows(2).all(|w| w[0] <= w[1]));

    println!("\n== Parallel Fibonacci ==\n");
    let start = std::time::Instant::now();
    let result = parallel_fib(35);
    println!("fib(35) = {} in {:?}", result, start.elapsed());

    println!("\n== Expression Evaluation ==\n");
    let expr = Expr::Add(
        Box::new(Expr::Mul(
            Box::new(Expr::Num(5)),
            Box::new(Expr::Num(10)),
        )),
        Box::new(Expr::Sub(
            Box::new(Expr::Num(20)),
            Box::new(Expr::Num(8)),
        )),
    );
    let result = expr.parallel_eval();
    println!("Expression result: {}", result);
}

```

Recursive Parallelism Tips: - **Sequential cutoff:** Switch to sequential below threshold - **rayon::join:** Parallel fork-join primitive - **Balance:** Ensure subtasks have similar work - **Overhead:** Avoid creating too many small tasks

Pattern 3: Parallel Reduce and Fold

Problem: Aggregating results from parallel operations is non-trivial. Simple sum/min/max work, but custom aggregations (histograms, statistics, merging maps) require careful design.

Solution: Use `reduce` for simple aggregations (sum, min, max, product). Use `fold + reduce` pattern for custom accumulators: fold builds per-thread state, reduce combines them.

Why It Matters: Statistics in one parallel pass instead of multiple sequential passes. Histogram generation: parallel fold+reduce 10x faster than sequential.

Use Cases: Statistics computation (mean, variance, stddev), histograms and frequency counting, word counting in text processing, aggregating results from parallel operations, merging sorted chunks, custom accumulators (sets, maps).

Example: Parallel Reduction Patterns

Efficiently combine results from parallel operations.

```
use rayon::prelude::*;
use std::collections::HashMap;
```

Example: Simple reduce (sum, min, max)

This example keeps things simple while focusing on reduce (sum, min, max) to make the mechanics obvious.

```
fn simple_reductions() {
    let numbers: Vec<i32> = (1..=1_000_000).collect();

    // Sum
    let sum: i32 = numbers.par_iter().sum();
    println!("Sum: {}", sum);

    // Min/Max
    let min = numbers.par_iter().min().unwrap();
    let max = numbers.par_iter().max().unwrap();
    println!("Min: {}, Max: {}", min, max);

    // Product (be careful of overflow!)
    let small_numbers: Vec<i32> = (1..=10).collect();
    let product: i32 = small_numbers.par_iter().product();
```

```
    println!("Product: {}", product);
}
```

Example: Reduce with custom operation

This example shows how to reduce with custom operation in practice, emphasizing why it works.

```
fn custom_reduce() {
    let numbers: Vec<i32> = (1..=100).collect();

    // Custom reduction: concatenate all numbers
    let concatenated = numbers
        .par_iter()
        .map(|n| n.to_string())
        .reduce(|| String::new(), |a, b| format!("{}{},{}", a, b, b));

    println!("Concatenated (first 50 chars): {}", &concatenated[..50.min(concatenated.len())]);

    // Find element closest to target
    let target = 42;
    let closest = numbers
        .par_iter()
        .reduce(
            || &numbers[0],
            |a, b| {
                if (a - target).abs() < (b - target).abs() {
                    a
                } else {
                    b
                }
            },
        );
    println!("Closest to {}: {}", target, closest);
}
```

Example: fold vs reduce

This example shows how to fold vs reduce in practice, emphasizing why it works.

```
fn fold_vs_reduce() {
    let numbers: Vec<i32> = (1..=1000).collect();

    // fold: provide identity and combine function
    let sum_fold = numbers.par_iter().fold(
        || 0, // Identity function
        |acc, &x| acc + x, // Fold function
    ).sum::<i32>(); // Reduce the folded results
```

```

// reduce: simpler but less flexible
let sum_reduce = numbers.par_iter().sum::<i32>();

assert_eq!(sum_fold, sum_reduce);
println!("Sum: {}", sum_fold);
}

```

Example: fold_with for custom accumulators

This example shows how to fold_with for custom accumulators in practice, emphasizing why it works.

```

fn fold_with_accumulator() {
    let numbers: Vec<i32> = (1..=100).collect();

    // Collect statistics in one pass
    #[derive(Default)]
    struct Stats {
        count: usize,
        sum: i64,
        min: i32,
        max: i32,
    }

    let stats = numbers
        .par_iter()
        .fold(
            || Stats {
                count: 0,
                sum: 0,
                min: i32::MAX,
                max: i32::MIN,
            },
            |mut acc, &x| {
                acc.count += 1;
                acc.sum += x as i64;
                acc.min = acc.min.min(x);
                acc.max = acc.max.max(x);
                acc
            },
        )
        .reduce(
            || Stats::default(),
            |a, b| Stats {
                count: a.count + b.count,
                sum: a.sum + b.sum,
                min: a.min.min(b.min),
                max: a.max.max(b.max),
            },
        );
}

```

```

    println!("Count: {}", stats.count);
    println!("Average: {:.2}", stats.sum as f64 / stats.count as f64);
    println!("Min: {}, Max: {}", stats.min, stats.max);
}

// Real-world: Parallel histogram
fn parallel_histogram(data: Vec<i32>) -> HashMap<i32, usize> {
    data.par_iter()
        .fold(
            || HashMap::new(),
            |mut map, &value| {
                *map.entry(value).or_insert(0) += 1;
                map
            },
        )
        .reduce(
            || HashMap::new(),
            |mut a, b| {
                for (key, count) in b {
                    *a.entry(key).or_insert(0) += count;
                }
                a
            },
        ),
    }
}

// Real-world: Word frequency count
fn word_frequency_parallel(text: String) -> HashMap<String, usize> {
    text.par_split_whitespace()
        .fold(
            || HashMap::new(),
            |mut map, word| {
                *map.entry(word.to_lowercase()).or_insert(0) += 1;
                map
            },
        )
        .reduce(
            || HashMap::new(),
            |mut a, b| {
                for (word, count) in b {
                    *a.entry(word).or_insert(0) += count;
                }
                a
            },
        ),
    }
}

// Real-world: Parallel variance calculation
fn parallel_variance(numbers: &[f64]) -> (f64, f64) {
    // Two-pass algorithm (more numerically stable)
    let mean = numbers.par_iter().sum::<f64>() / numbers.len() as f64;

    let variance = numbers
        .par_iter()
        .map(|&x| (x - mean).powi(2))
        .sum::<f64>()
}

```

```

    / numbers.len() as f64;

    (mean, variance)
}

// Real-world: Merge sorted chunks
fn parallel_merge_reduce(mut chunks: Vec<Vec<i32>>) -> Vec<i32> {
    while chunks.len() > 1 {
        chunks = chunks
            .par_chunks(2)
            .map(|pair| {
                if pair.len() == 2 {
                    merge(&pair[0], &pair[1])
                } else {
                    pair[0].clone()
                }
            })
            .collect();
    }

    chunks.into_iter().next().unwrap_or_default()
}

fn merge(a: &[i32], b: &[i32]) -> Vec<i32> {
    let mut result = Vec::with_capacity(a.len() + b.len());
    let mut i = 0;
    let mut j = 0;

    while i < a.len() && j < b.len() {
        if a[i] <= b[j] {
            result.push(a[i]);
            i += 1;
        } else {
            result.push(b[j]);
            j += 1;
        }
    }

    result.extend_from_slice(&a[i..]);
    result.extend_from_slice(&b[j..]);
    result
}

fn main() {
    println!("== Simple Reductions ==\n");
    simple_reductions();

    println!("\n== Custom Reduce ==\n");
    custom_reduce();

    println!("\n== Fold with Accumulator ==\n");
    fold_with_accumulator();

    println!("\n== Parallel Histogram ==\n");
}

```

```

let data: Vec<i32> = (0..10000).map(|i| i % 100).collect();
let histogram = parallel_histogram(data);
println!("Histogram buckets: {}", histogram.len());
println!("Bucket 50: {}", histogram.get(&50).unwrap_or(&0));

println!("\n==== Word Frequency ===\n");
let text = "the quick brown fox jumps over the lazy dog the fox".to_string();
let freq = word_frequency_parallel(text);
for (word, count) in freq.iter().take(5) {
    println!("{}: {}", word, count);
}

println!("\n==== Variance ===\n");
let numbers: Vec<f64> = (1..=100).map(|x| x as f64).collect();
let (mean, variance) = parallel_variance(&numbers);
println!("Mean: {:.2}, Variance: {:.2}, StdDev: {:.2}", mean, variance, variance.sqrt());
}

```

Reduction Patterns: - **reduce**: Simple aggregation (sum, min, max) - **fold + reduce**: Custom accumulator, then combine - **Associative operations**: Required for correctness - **Commutative**: Not required but helps performance

Pattern 4: Pipeline Parallelism

Problem: Multi-stage data processing often bottlenecks on slowest stage. Sequential pipeline wastes CPU—decode thread idle while enhance runs.

Solution: Use channel-based pipelines with separate threads per stage. Rayon’s `par_iter` at each stage for intra-stage parallelism.

Why It Matters: Image processing pipeline: sequential 300ms, staged parallel 100ms (3x faster). ETL pipeline processing 1M records: sequential 10min, parallel pipeline 2min (5x speedup).

Use Cases: ETL (Extract-Transform-Load) data pipelines, image/video processing (decode→enhance→compress), log analysis (parse→enrich→filter→aggregate), data transformation chains, streaming data processing, multi-stage batch jobs.

Example: Multi-Stage Pipelines

Process data through multiple transformation stages with different computational costs.:

```

use rayon::prelude::*;
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

```

Example: Simple pipeline with channels

This example keeps things simple while focusing on pipeline with channels to make the mechanics obvious.

```
fn simple_pipeline() {
    let (stage1_tx, stage1_rx) = mpsc::sync_channel(100);
    let (stage2_tx, stage2_rx) = mpsc::sync_channel(100);
    let (stage3_tx, stage3_rx) = mpsc::sync_channel(100);

    // Stage 1: Data generation
    let producer = thread::spawn(move || {
        for i in 0..1000 {
            stage1_tx.send(i).unwrap();
        }
    });

    // Stage 2: Transform (parallel)
    let stage2 = thread::spawn(move || {
        stage1_rx
            .into_iter()
            .par_bridge()
            .for_each_with(stage2_tx, |tx, item| {
                let transformed = item * 2;
                tx.send(transformed).unwrap();
            });
    });

    // Stage 3: Filter (parallel)
    let stage3 = thread::spawn(move || {
        stage2_rx
            .into_iter()
            .par_bridge()
            .filter(|&x| x % 4 == 0)
            .for_each_with(stage3_tx, |tx, item| {
                tx.send(item).unwrap();
            });
    });

    // Consumer
    let consumer = thread::spawn(move || {
        let sum: i32 = stage3_rx.into_iter().sum();
        sum
    });

    producer.join().unwrap();
    stage2.join().unwrap();
    stage3.join().unwrap();
    let result = consumer.join().unwrap();

    println!("Pipeline result: {}", result);
}
```

```
}

// Real-world: Image processing pipeline
struct ImagePipeline;

impl ImagePipeline {
    fn process_batch(images: Vec<Vec<u8>>) -> Vec<Vec<u8>> {
        images
            .into_par_iter()
            .map(|img| Self::stage1_decode(img))
            .map(|img| Self::stage2_enhance(img))
            .map(|img| Self::stage3_compress(img))
            .collect()
    }

    fn stage1_decode(data: Vec<u8>) -> Vec<u8> {
        // Simulate decoding
        thread::sleep(Duration::from_millis(100));
        data
    }

    fn stage2_enhance(mut data: Vec<u8>) -> Vec<u8> {
        // Simulate enhancement
        for pixel in &mut data {
            *pixel = pixel.saturating_add(10);
        }
        data
    }

    fn stage3_compress(data: Vec<u8>) -> Vec<u8> {
        // Simulate compression
        thread::sleep(Duration::from_millis(50));
        data
    }
}

// Real-world: Log processing pipeline
#[derive(Debug, Clone)]
struct RawLog(String);

#[derive(Debug, Clone)]
struct ParsedLog {
    timestamp: u64,
    level: String,
    message: String,
}

#[derive(Debug, Clone)]
struct EnrichedLog {
    log: ParsedLog,
    metadata: String,
}

struct LogPipeline;
```

```

impl LogPipeline {
    fn process(logs: Vec<RawLog>) -> Vec<EnrichedLog> {
        logs.into_par_iter()
            .filter_map(|raw| Self::parse(raw))
            .map(|parsed| Self::enrich(parsed))
            .filter(|enriched| enriched.log.level == "ERROR")
            .collect()
    }

    fn parse(raw: RawLog) -> Option<ParsedLog> {
        let parts: Vec<&str> = raw.0.split('|').collect();
        if parts.len() >= 3 {
            Some(ParsedLog {
                timestamp: parts[0].parse().ok()?,
                level: parts[1].to_string(),
                message: parts[2].to_string(),
            })
        } else {
            None
        }
    }

    fn enrich(log: ParsedLog) -> EnrichedLog {
        EnrichedLog {
            log: log.clone(),
            metadata: format!("enriched_{}", log.timestamp),
        }
    }
}

```

Example: Parallel stages with different parallelism

This example shows how to parallel stages with different parallelism while calling out the practical trade-offs.

```

fn multi_stage_parallel() {
    let data: Vec<i32> = (0..10000).collect();

    // Stage 1: Light processing (high parallelism)
    let stage1: Vec<i32> = data
        .par_iter()
        .map(|&x| x + 1)
        .collect();

    // Stage 2: Heavy processing (moderate parallelism)
    let stage2: Vec<i32> = stage1
        .par_chunks(100) // Larger chunks for heavy work
        .flat_map(|chunk| {
            chunk.iter().map(|&x| {
                // Simulate heavy computation
                let mut result = x;
                result *= 2;
                result
            })
        })
        .collect();
}

```

```

        for _ in 0..100 {
            result = (result * 2) % 1000;
        }
        result
    })
}
.collect();

// Stage 3: Aggregation
let sum: i32 = stage2.par_iter().sum();

println!("Multi-stage result: {}", sum);
}

// Real-world: ETL pipeline (Extract, Transform, Load)
struct EtlPipeline;

impl EtlPipeline {
    fn run(input_files: Vec<String>) -> Vec<(String, usize)> {
        input_files
            .into_par_iter()
            // Extract: Read files in parallel
            .filter_map(|file| Self::extract(&file))
            // Transform: Process data in parallel
            .map(|data| Self::transform(data))
            // Load: Aggregate results
            .collect()
    }

    fn extract(file: &str) -> Option<Vec<String>> {
        // Simulate file reading
        Some(vec![format!("data_from_{}", file)])
    }

    fn transform(data: Vec<String>) -> (String, usize) {
        // Simulate transformation
        let processed = data
            .par_iter()
            .map(|s| s.to_uppercase())
            .collect::<Vec<_>>();

        ("transformed".to_string(), processed.len())
    }
}

fn main() {
    println!("== Simple Pipeline ==\n");
    simple_pipeline();

    println!("\n== Image Processing Pipeline ==\n");
    let images: Vec<Vec<u8>> = (0..100).map(|_| vec![128; 1000]).collect();
    let start = std::time::Instant::now();
    let processed = ImagePipeline::process_batch(images);
    println!("Processed {} images in {:?}", processed.len(), start.elapsed());
}

```

```

    println!("\\n==== Log Processing Pipeline ===\\n");
    let logs: Vec<RawLog> = (0..1000)
        .map(|i| RawLog(format!("{}|{}|message_{}", i, if i % 10 == 0 { "ERROR" } else {
            "INFO" }, i)))
        .collect();

    let errors = LogPipeline::process(logs);
    println!("Found {} errors", errors.len());

    println!("\\n==== Multi-Stage Parallel ===\\n");
    multi_stage_parallel();

    println!("\\n==== ETL Pipeline ===\\n");
    let files: Vec<String> = (0..100).map(|i| format!("file_{}.csv", i)).collect();
    let results = EtlPipeline::run(files);
    println!("Processed {} files", results.len());
}

```

Pipeline Patterns: - **Channel-based:** Explicit stages with bounded buffers - **Iterator-based:** Chain transformations with `par_iter` - **Staged parallelism:** Different parallelism per stage - **Backpressure:** Bounded channels prevent memory issues

Pattern 5: SIMD Parallelism

Problem: CPU vector units (AVX2: 8 floats, AVX-512: 16 floats) sit idle with scalar code. Data-level parallelism untapped—process 1 element when hardware can do 8.

Solution: Write SIMD-friendly code: contiguous arrays, simple operations, no branches in hot loops. Use Struct-of-Arrays (SoA) instead of Array-of-Structs (AoS) for better vectorization.

Why It Matters: Matrix multiply: 10x speedup with SIMD+threading vs scalar sequential. Dot product: 4-8x faster with vectorization.

Use Cases: Matrix operations (multiply, transpose, dot product), image processing (convolution, filters), signal processing (FFT, filters), scientific computing (numerical methods), vector arithmetic, statistical computations.

Example: Portable SIMD with `std::simd`

Vectorize operations across array elements for maximum throughput.

```

// Note: Requires nightly Rust
// Add to Cargo.toml:
// [dependencies]
// packed_simd = "0.3"
// For stable Rust, we'll use a portable SIMD approach

```

Example: Manual SIMD-friendly code

This example shows how to manual SIMD-friendly code while calling out the practical trade-offs.

```
fn simd_friendly_sum(data: &[f32]) -> f32 {
    // Process 4 elements at a time (compiler can auto-vectorize)
    let chunks = data.chunks_exact(4);
    let remainder = chunks.remainder();

    let mut sums = [0.0f32; 4];

    for chunk in chunks {
        sums[0] += chunk[0];
        sums[1] += chunk[1];
        sums[2] += chunk[2];
        sums[3] += chunk[3];
    }

    let chunk_sum: f32 = sums.iter().sum();
    let remainder_sum: f32 = remainder.iter().sum();

    chunk_sum + remainder_sum
}
```

Example: Array operations (SIMD-friendly)

This example shows how to array operations (SIMD-friendly) in practice, emphasizing why it works.

```
fn vector_add(a: &[f32], b: &[f32], result: &mut [f32]) {
    assert_eq!(a.len(), b.len());
    assert_eq!(a.len(), result.len());

    // Compiler can auto-vectorize this
    for i in 0..a.len() {
        result[i] = a[i] + b[i];
    }
}

fn vector_add_parallel(a: &[f32], b: &[f32]) -> Vec<f32> {
    use rayon::prelude::*;

    // Combine SIMD and thread parallelism
    a.par_iter()
        .zip(b.par_iter())
        .map(|(&x, &y)| x + y)
        .collect()
}
```

Example: Dot product (SIMD-friendly)

This example shows how to dot product (SIMD-friendly) in practice, emphasizing why it works.

```
fn dot_product(a: &[f32], b: &[f32]) -> f32 {
    assert_eq!(a.len(), b.len());

    a.iter()
        .zip(b.iter())
        .map(|(&x, &y)| x * y)
        .sum()
}

fn dot_product_parallel(a: &[f32], b: &[f32]) -> f32 {
    use rayon::prelude::*;

    assert_eq!(a.len(), b.len());

    a.par_iter()
        .zip(b.par_iter())
        .map(|(&x, &y)| x * y)
        .sum()
}

// Real-world: Matrix multiplication with SIMD hints
fn matrix_multiply_simd(a: &[f32], b: &[f32], result: &mut [f32], n: usize) {
    // Matrix dimensions: n x n
    assert_eq!(a.len(), n * n);
    assert_eq!(b.len(), n * n);
    assert_eq!(result.len(), n * n);

    for i in 0..n {
        for j in 0..n {
            let mut sum = 0.0;

            // Inner loop is SIMD-friendly
            for k in 0..n {
                sum += a[i * n + k] * b[k * n + j];
            }

            result[i * n + j] = sum;
        }
    }
}

// Parallel + SIMD matrix multiplication
fn matrix_multiply_parallel_simd(a: &[f32], b: &[f32], n: usize) -> Vec<f32> {
    use rayon::prelude::*;

    let mut result = vec![0.0; n * n];

    (0..n).into_par_iter().for_each(|i| {
        for j in 0..n {
```

```

    let mut sum = 0.0;

    // This loop can be auto-vectorized
    for k in 0..n {
        sum += a[i * n + k] * b[k * n + j];
    }

    result[i * n + j] = sum;
}
});

result
}

```

Example: Blocked matrix operations (cache + SIMD friendly)

This example shows blocked matrix operations (cache + SIMD friendly) to illustrate where the pattern fits best.

```

fn blocked_matrix_multiply(a: &[f32], b: &[f32], result: &mut [f32], n: usize, block_size: usize) {
    use rayon::prelude::*;

    for i_block in (0..n).step_by(block_size) {
        for j_block in (0..n).step_by(block_size) {
            for k_block in (0..n).step_by(block_size) {
                // Process block
                let i_end = (i_block + block_size).min(n);
                let j_end = (j_block + block_size).min(n);
                let k_end = (k_block + block_size).min(n);

                for i in i_block..i_end {
                    for j in j_block..j_end {
                        let mut sum = result[i * n + j];

                        // Inner loop is SIMD-friendly
                        for k in k_block..k_end {
                            sum += a[i * n + k] * b[k * n + j];
                        }

                        result[i * n + j] = sum;
                    }
                }
            }
        }
    }
}

// Real-world: Image convolution
fn convolve_2d(image: &[f32], kernel: &[f32], width: usize, height: usize, kernel_size: usize) -> Vec<f32> {
    use rayon::prelude::*;

```

```

let offset = kernel_size / 2;
let mut result = vec![0.0; width * height];

(offset..height - offset).into_par_iter().for_each(|y| {
    for x in offset..width - offset {
        let mut sum = 0.0;

        // Convolution kernel (SIMD-friendly inner loops)
        for ky in 0..kernel_size {
            for kx in 0..kernel_size {
                let img_y = y + ky - offset;
                let img_x = x + kx - offset;
                let img_idx = img_y * width + img_x;
                let kernel_idx = ky * kernel_size + kx;

                sum += image[img_idx] * kernel[kernel_idx];
            }
        }

        result[y * width + x] = sum;
    }
});

result
}

```

Example: Reduction with SIMD

This example shows how to reduction with SIMD in practice, emphasizing why it works.

```

fn parallel_sum_simd(data: &[f32]) -> f32 {
    use rayon::prelude::*;

    // Split into chunks for parallel processing
    data.par_chunks(1024)
        .map(|chunk| {
            // Each chunk can be SIMD-vectorized
            chunk.iter().sum::<f32>()
        })
        .sum()
}

fn main() {
    println!("==== SIMD-Friendly Sum ====\n");

    let data: Vec<f32> = (0..1_000_000).map(|x| x as f32).collect();

    let start = std::time::Instant::now();
    let sum = simd_friendly_sum(&data);
    println!("Sum: {} in {:?}", sum, start.elapsed());
}

```

```

println!("\
    Vector Operations ===\n");

let a: Vec<f32> = (0..1000).map(|x| x as f32).collect();
let b: Vec<f32> = (0..1000).map(|x| (x * 2) as f32).collect();

let start = std::time::Instant::now();
let result = vector_add_parallel(&a, &b);
println!("Vector add: {} elements in {:?}", result.len(), start.elapsed());

println!("\
    Dot Product ===\n");

let start = std::time::Instant::now();
let dot = dot_product_parallel(&a, &b);
println!("Dot product: {} in {:?}", dot, start.elapsed());

println!("\
    Matrix Multiplication ===\n");

let n = 512;
let a: Vec<f32> = (0..n * n).map(|x| x as f32).collect();
let b: Vec<f32> = (0..n * n).map(|x| (x * 2) as f32).collect();

let start = std::time::Instant::now();
let result = matrix_multiply_parallel_simd(&a, &b, n);
println!("Matrix multiply ({}x{}) : {:?}", n, n, start.elapsed());
println!("Result checksum: {}", result.iter().sum::<f32>());

println!("\
    Parallel Sum with SIMD ===\n");

let start = std::time::Instant::now();
let sum = parallel_sum_simd(&data);
println!("Parallel sum: {} in {:?}", sum, start.elapsed());
}

```

SIMD Optimization Tips: - **Alignment:** Align data to 16/32-byte boundaries - **Contiguous memory:** Use arrays/slices, not scattered data - **Inner loops:** Make innermost loops SIMD-friendly - **Combine with threading:** Rayon + SIMD for maximum performance - **Profile:** Use compiler output to verify vectorization

Example: Auto-Vectorization and Hints

Help the compiler generate SIMD code effectively.

Example: Iterator patterns that auto-vectorize

This example shows how to iterator patterns that auto-vectorize in practice, emphasizing why it works.

```
fn auto_vectorize_examples() {
```

```

let data: Vec<f32> = (0..10000).map(|x| x as f32).collect();

// Good: Simple map (auto-vectorizes)
let doubled: Vec<f32> = data.iter().map(|&x| x * 2.0).collect();

// Good: Zip and map (auto-vectorizes)
let summed: Vec<f32> = data
    .iter()
    .zip(data.iter())
    .map(|(&a, &b)| a + b)
    .collect();

// Good: Chunks with fold (auto-vectorizes)
let chunk_sums: Vec<f32> = data
    .chunks(4)
    .map(|chunk| chunk.iter().sum())
    .collect();
}

```

Example: Explicit chunking for better vectorization

This example shows how to explicit chunking for better vectorization in practice, emphasizing why it works.

```

fn chunked_operations(data: &[f32]) -> Vec<f32> {
    let mut result = Vec::with_capacity(data.len());

    // Process in SIMD-width chunks
    const SIMD_WIDTH: usize = 8; // Typical for AVX

    for chunk in data.chunks(SIMD_WIDTH) {
        for &value in chunk {
            result.push(value * 2.0 + 1.0);
        }
    }

    result
}

```

Example: Struct of Arrays (SoA) vs Array of Structs (AoS)

This example shows how to struct of Arrays (SoA) vs Array of Structs (AoS) in practice, emphasizing why it works.

```

// Bad for SIMD: Array of Structs
#[derive(Copy, Clone)]
struct PointAoS {
    x: f32,
    y: f32,
}

```

```

    z: f32,
}

fn process_aos(points: &[PointAoS]) -> Vec<f32> {
    // Poor vectorization: scattered access
    points.iter().map(|p| p.x + p.y + p.z).collect()
}

// Good for SIMD: Struct of Arrays
struct PointsSoA {
    x: Vec<f32>,
    y: Vec<f32>,
    z: Vec<f32>,
}

impl PointsSoA {
    fn process(&self) -> Vec<f32> {
        // Good vectorization: contiguous access
        self.x
            .iter()
            .zip(self.y.iter())
            .zip(self.z.iter())
            .map(|((&x, &y), &z)| x + y + z)
            .collect()
    }
}

// Real-world: Parallel + SIMD Monte Carlo
fn monte_carlo_pi_parallel_simd(samples: usize) -> f64 {
    use rayon::prelude::*;
    use rand::Rng;

    let inside: usize = (0..samples)
        .into_par_iter()
        .chunks(1024) // Process in batches
        .map(|chunk| {
            let mut rng = rand::thread_rng();
            let mut count = 0;

            // Inner loop can be vectorized
            for _ in chunk {
                let x: f32 = rng.gen();
                let y: f32 = rng.gen();

                if x * x + y * y <= 1.0 {
                    count += 1;
                }
            }
            count
        })
        .sum();
}

```

```
    4.0 * inside as f64 / samples as f64
}
```

Example: Benchmarking SIMD effectiveness

This example shows benchmarking SIMD effectiveness to illustrate where the pattern fits best.

```
fn benchmark_vectorization() {
    let data: Vec<f32> = (0..10_000_000).map(|x| x as f32).collect();

    // Version 1: Simple loop
    let start = std::time::Instant::now();
    let mut result1 = Vec::with_capacity(data.len());
    for &x in &data {
        result1.push(x * 2.0 + 1.0);
    }
    let time1 = start.elapsed();

    // Version 2: Iterator (likely vectorized)
    let start = std::time::Instant::now();
    let result2: Vec<f32> = data.iter().map(|&x| x * 2.0 + 1.0).collect();
    let time2 = start.elapsed();

    // Version 3: Parallel + potential vectorization
    use rayon::prelude::*;
    let start = std::time::Instant::now();
    let result3: Vec<f32> = data.par_iter().map(|&x| x * 2.0 + 1.0).collect();
    let time3 = start.elapsed();

    println!("Simple loop: {:?}", time1);
    println!("Iterator: {:?}", time2);
    println!("Parallel: {:?}", time3);
    println!("Speedup (iter vs loop): {:.2}x", time1.as_secs_f64() / time2.as_secs_f64());
    println!("Speedup (parallel vs iter): {:.2}x", time2.as_secs_f64() / time3.as_secs_f64());
}

fn main() {
    println!("== Auto-Vectorization ==\n");
    auto_vectorize_examples();

    println!("\n== SoA vs AoS ==\n");

    let points_soa = PointsSoA {
        x: (0..10000).map(|i| i as f32).collect(),
        y: (0..10000).map(|i| (i * 2) as f32).collect(),
        z: (0..10000).map(|i| (i * 3) as f32).collect(),
    };

    let start = std::time::Instant::now();
    let sums = points_soa.process();
    println!("SoA processing: {:?}", start.elapsed());
```

```

    println!("Checksum: {}", sums.iter().sum::<f32>());
    println!("\n==== Monte Carlo Pi ===\n");

    let start = std::time::Instant::now();
    let pi = monte_carlo_pi_parallel_simd(10_000_000);
    println!("Pi estimate: {} in {:?}", pi, start.elapsed());

    println!("\n==== Vectorization Benchmark ===\n");
    benchmark_vectorization();
}


```

Vectorization Guidelines: 1. **Contiguous data:** Use slices/arrays 2. **Simple operations:** +, -, *, / vectorize well 3. **No branching:** Avoid if/else in hot loops 4. **Struct of Arrays:** Better than Array of Structs 5. **Verify:** Use `cargo rustc -- --emit asm` to check

Summary

This chapter covered parallel algorithm patterns in Rust:

1. **Rayon Patterns:** `par_iter`, `par_bridge` for easy parallelization
2. **Work Partitioning:** Chunking, load balancing, recursive parallelism
3. **Parallel Reduce/Fold:** Aggregation patterns, custom accumulators
4. **Pipeline Parallelism:** Multi-stage processing with channels
5. **SIMD Parallelism:** Auto-vectorization, SoA layout, parallel + SIMD

Key Takeaways: - **Rayon** makes data parallelism trivial (add `.par_`) - **Work stealing** automatically balances irregular workloads - **Grain size** matters: too small = overhead, too large = imbalance - **fold + reduce** for custom aggregations - **Pipeline** stages can have different parallelism levels - **SIMD + threading** for maximum performance - **SoA layout** enables better vectorization

Performance Guidelines: - Use Rayon for CPU-bound data parallelism - Combine thread parallelism and SIMD for best results - Profile to find bottlenecks (CPU, memory, cache) - Tune chunk size based on workload - Use blocked algorithms for cache efficiency

Common Patterns: - **Map:** Transform each element independently - **Filter:** Select elements based on predicate - **Reduce:** Aggregate to single value - **Fold:** Accumulate with custom state - **Pipeline:** Chain transformations

When to Parallelize: - **Large datasets:** >10,000 items typically - **CPU-bound:** Compute-intensive operations - **Independent work:** No dependencies between items - **Speedup > overhead:** Measure, don't assume

Pitfalls to Avoid: - Over-parallelization (too many small tasks) - False sharing (cache line contention) - Sequential bottlenecks (Amdahl's law) - Ignoring memory bandwidth limits - Not profiling actual performance

Smart Pointer Patterns

This chapter explores smart pointer patterns in Rust, covering heap allocation with Box, reference counting with Rc/Arc, preventing memory leaks with Weak references, implementing custom smart pointers, intrusive data structures, and optimization techniques.

Pattern 1: Box, Rc, Arc Usage Patterns

Problem: Rust's default stack allocation breaks with recursive types (infinite size), large structs (stack overflow risk), and trait objects (size unknown at compile time). Need heap allocation with single ownership.

Solution: Use `Box<T>` for single-ownership heap allocation—recursive types wrapped in Box, large structs on heap instead of stack. Use `Rc<T>` for single-threaded shared ownership—multiple Rc pointers to same data, automatic cleanup when last owner drops.

Why It Matters: Box enables recursive types (compiler error without it). Box prevents stack overflow: 1MB struct on stack crashes, Box reduces it to 8 bytes.

Use Cases: Box for binary trees, linked lists, large structs, trait object collections, AST nodes. Rc for graphs, DAGs, shared configuration, document versions, immutable shared data. Arc for thread pools, shared caches, concurrent config, work queues, parallel processing.

Example: Box for Heap Allocation

Store data on the heap for recursive types, large data, or trait objects.

```
use std::mem;
```

Example: Recursive types require Box

This example shows how to recursive types require Box in practice, emphasizing why it works.

```
#[derive(Debug)]
enum List<T> {
    Cons(T, Box<List<T>>),
    Nil,
}

impl<T> List<T> {
    fn new() -> Self {
        List::Nil
    }

    fn prepend(self, elem: T) -> Self {
        List::Cons(elem, Box::new(self))
    }
}
```

```

fn len(&self) -> usize {
    match self {
        List::Cons(_, tail) => 1 + tail.len(),
        List::Nil => 0,
    }
}

fn iter(&self) -> ListIter<T> {
    ListIter { current: self }
}

struct ListIter<'a, T> {
    current: &'a List<T>,
}

impl<'a, T> Iterator for ListIter<'a, T> {
    type Item = &'a T;

    fn next(&mut self) -> Option<Self::Item> {
        match self.current {
            List::Cons(value, tail) => {
                self.current = tail;
                Some(value)
            }
            List::Nil => None,
        }
    }
}

```

Example: Large structs on heap

This example shows how to large structs on heap in practice, emphasizing why it works.

```

struct LargeData {
    buffer: [u8; 1024 * 1024], // 1MB
}

fn stack_overflow_risk() -> LargeData {
    // This could overflow the stack!
    LargeData {
        buffer: [0; 1024 * 1024],
    }
}

fn heap_allocation() -> Box<LargeData> {
    // Safe: allocated on heap
    Box::new(LargeData {
        buffer: [0; 1024 * 1024],
    })
}

```

```
    }
}
```

Example: Trait objects require Box

This example shows how to trait objects require Box in practice, emphasizing why it works.

```
trait Drawable {
    fn draw(&self);
}

struct Circle {
    radius: f64,
}

impl Drawable for Circle {
    fn draw(&self) {
        println!("Drawing circle with radius {}", self.radius);
    }
}

struct Rectangle {
    width: f64,
    height: f64,
}

impl Drawable for Rectangle {
    fn draw(&self) {
        println!("Drawing rectangle {}x{}", self.width, self.height);
    }
}

fn trait_objects() {
    let shapes: Vec<Box<dyn Drawable>> = vec![
        Box::new(Circle { radius: 5.0 }),
        Box::new(Rectangle {
            width: 10.0,
            height: 20.0,
        }),
    ];
    for shape in shapes {
        shape.draw();
    }
}

// Real-world: Binary tree
#[derive(Debug)]
struct TreeNode<T> {
    value: T,
    left: Option<Box<TreeNode<T>>>,
    right: Option<Box<TreeNode<T>>>,
}
```

```
}

impl<T: Ord> TreeNode<T> {
    fn new(value: T) -> Self {
        Self {
            value,
            left: None,
            right: None,
        }
    }

    fn insert(&mut self, value: T) {
        if value < self.value {
            match &mut self.left {
                Some(node) => node.insert(value),
                None => self.left = Some(Box::new(TreeNode::new(value))),
            }
        } else {
            match &mut self.right {
                Some(node) => node.insert(value),
                None => self.right = Some(Box::new(TreeNode::new(value))),
            }
        }
    }

    fn contains(&self, value: &T) -> bool {
        if value == &self.value {
            true
        } else if value < &self.value {
            self.left.as_ref().map_or(false, |node| node.contains(value))
        } else {
            self.right.as_ref().map_or(false, |node| node.contains(value))
        }
    }

    fn inorder_iter(&self) -> Vec<&T> {
        let mut result = Vec::new();

        if let Some(left) = &self.left {
            result.extend(left.inorder_iter());
        }

        result.push(&self.value);

        if let Some(right) = &self.right {
            result.extend(right.inorder_iter());
        }

        result
    }
}
```

Example: Box for ownership transfer

This example shows how to box for ownership transfer in practice, emphasizing why it works.

```
fn process_large_data(data: Box<LargeData>) {
    // Takes ownership without copying
    println!("Processing {} bytes", data.buffer.len());
}

fn main() {
    println!("==== Linked List ===\\n");

    let list = List::new()
        .prepend(3)
        .prepend(2)
        .prepend(1);

    println!("List length: {}", list.len());
    println!("List items: {:?}", list.iter().collect::<Vec<_>>());

    println!("\\n==== Stack vs Heap ===\\n");

    println!("LargeData size: {} bytes", mem::size_of::<LargeData>());
    println!("Box<LargeData> size: {} bytes", mem::size_of::<Box<LargeData>>());

    let heap_data = heap_allocation();
    process_large_data(heap_data);

    println!("\\n==== Trait Objects ===\\n");
    trait_objects();

    println!("\\n==== Binary Tree ===\\n");

    let mut tree = TreeNode::new(5);
    for value in [3, 7, 1, 4, 6, 9] {
        tree.insert(value);
    }

    println!("Tree contains 4: {}", tree.contains(&4));
    println!("Tree contains 8: {}", tree.contains(&8));
    println!("Inorder: {:?}", tree.inorder_iter());
}
```

Box Use Cases: - **Recursive types:** Lists, trees, graphs - **Large data:** Avoid stack overflow - **Trait objects:** Dynamic dispatch - **Ownership transfer:** Move without copying

Example: Rc for Shared Ownership

Multiple owners need read-only access to the same data.

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

Example: Shared configuration

This example shows shared configuration to illustrate where the pattern fits best.

```
struct Config {
    database_url: String,
    max_connections: usize,
    timeout_ms: u64,
}

struct DatabasePool {
    config: Rc<Config>,
}

struct CacheService {
    config: Rc<Config>,
}

struct ApiServer {
    config: Rc<Config>,
}

impl DatabasePool {
    fn new(config: Rc<Config>) -> Self {
        println!("DB Pool using: {}", config.database_url);
        Self { config }
    }
}

impl CacheService {
    fn new(config: Rc<Config>) -> Self {
        println!("Cache using timeout: {}ms", config.timeout_ms);
        Self { config }
    }
}

impl ApiServer {
    fn new(config: Rc<Config>) -> Self {
        println!("API Server max connections: {}", config.max_connections);
        Self { config }
    }
}

fn shared_config() {
    let config = Rc::new(Config {
        database_url: "postgresql://localhost/mydb".to_string(),
        max_connections: 100,
        timeout_ms: 5000,
    })
}
```

```

});  
  

println!("Initial ref count: {}", Rc::strong_count(&config));  
  

let db_pool = DatabasePool::new(Rc::clone(&config));
println!("After DB pool: {}", Rc::strong_count(&config));  
  

let cache = CacheService::new(Rc::clone(&config));
println!("After cache: {}", Rc::strong_count(&config));  
  

let api = ApiServer::new(Rc::clone(&config));
println!("After API: {}", Rc::strong_count(&config));  
  

// config, db_pool, cache, api all dropped at end of scope
// Reference count goes to 0, memory freed
}

```

Example: Shared data in graph

This example shows shared data in graph to illustrate where the pattern fits best.

```

#[derive(Debug)]
struct Node {
    id: usize,
    value: String,
}  
  

struct Graph {
    nodes: Vec<Rc<Node>>,
    edges: Vec<(Rc<Node>, Rc<Node>)>,
}  
  

impl Graph {
    fn new() -> Self {
        Self {
            nodes: Vec::new(),
            edges: Vec::new(),
        }
    }
  

    fn add_node(&mut self, id: usize, value: String) -> Rc<Node> {
        let node = Rc::new(Node { id, value });
        self.nodes.push(Rc::clone(&node));
        node
    }
  

    fn add_edge(&mut self, from: Rc<Node>, to: Rc<Node>) {
        self.edges.push((from, to));
    }
  

    fn print_edges(&self) {

```

```

        for (from, to) in &self.edges {
            println!("{} -> {}", from.value, to.value);
        }
    }
}

// Real-world: Immutable data sharing
#[derive(Debug, Clone)]
struct Document {
    content: String,
    metadata: String,
}

struct DocumentVersion {
    version: usize,
    doc: Rc<Document>,
}

struct VersionControl {
    versions: Vec<DocumentVersion>,
}

impl VersionControl {
    fn new(initial_content: String) -> Self {
        let doc = Rc::new(Document {
            content: initial_content,
            metadata: "v1".to_string(),
        });

        Self {
            versions: vec![DocumentVersion { version: 1, doc }],
        }
    }

    fn add_version(&mut self, content: String) {
        let version = self.versions.len() + 1;
        let doc = Rc::new(Document {
            content,
            metadata: format!("v{}", version),
        });

        self.versions.push(DocumentVersion { version, doc });
    }

    fn get_version(&self, version: usize) -> Option<Rc<Document>> {
        self.versions
            .get(version - 1)
            .map(|v| Rc::clone(&v.doc))
    }

    fn compare_versions(&self, v1: usize, v2: usize) {
        if let (Some(doc1), Some(doc2)) = (self.get_version(v1), self.get_version(v2)) {
            println!("Version {}: {}", v1, doc1.content);
            println!("Version {}: {}", v2, doc2.content);
        }
    }
}

```

```
    }
}
}
```

Example: Rc with interior mutability

This example shows how to rc with interior mutability while calling out the practical trade-offs.

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

struct Sensor {
    id: usize,
    reading: RefCell<f64>,
}

struct SensorNetwork {
    sensors: Vec<Rc<Sensor>>,
}

impl SensorNetwork {
    fn new() -> Self {
        Self {
            sensors: Vec::new(),
        }
    }

    fn add_sensor(&mut self, id: usize) -> Rc<Sensor> {
        let sensor = Rc::new(Sensor {
            id,
            reading: RefCell::new(0.0),
        });
        self.sensors.push(Rc::clone(&sensor));
        sensor
    }

    fn update_readings(&self) {
        for sensor in &self.sensors {
            *sensor.reading.borrow_mut() = rand::random::<f64>() * 100.0;
        }
    }

    fn average_reading(&self) -> f64 {
        let sum: f64 = self.sensors.iter().map(|s| *s.reading.borrow()).sum();
        sum / self.sensors.len() as f64
    }
}

fn main() {
    println!("--- Shared Config ---\n");
    shared_config();
```

```

println!("==> Graph with Shared Nodes ==>");

let mut graph = Graph::new();

let a = graph.add_node(1, "A".to_string());
let b = graph.add_node(2, "B".to_string());
let c = graph.add_node(3, "C".to_string());

graph.add_edge(Rc::clone(&a), Rc::clone(&b));
graph.add_edge(Rc::clone(&b), Rc::clone(&c));
graph.add_edge(Rc::clone(&a), Rc::clone(&c));

println!("Edges:");
graph.print_edges();

println!("Node A ref count: {}", Rc::strong_count(&a));

println!("==> Version Control ==>");

let mut vc = VersionControl::new("Initial content".to_string());
vc.add_version("Updated content".to_string());
vc.add_version("Final content".to_string());

vc.compare_versions(1, 3);

println!("==> Sensor Network ==>");

let mut network = SensorNetwork::new();
let sensor1 = network.add_sensor(1);
let sensor2 = network.add_sensor(2);

network.update_readings();
println!("Sensor 1: {:.2}", sensor1.reading.borrow());
println!("Sensor 2: {:.2}", sensor2.reading.borrow());
println!("Average: {:.2}", network.average_reading());
}

```

Rc Characteristics: - **Single-threaded:** Not thread-safe - **Shared ownership:** Multiple owners, last one frees - **Reference counting:** Overhead of counter updates - **Interior mutability:** Combine with RefCell for mutation

Example: Arc for Thread-Safe Sharing

Share data across threads safely.

```

use std::sync::{Arc, Mutex, RwLock};
use std::thread;
use std::time::Duration;

```

Example: Shared read-only data across threads

This example shows shared read-only data across threads to illustrate where the pattern fits best.

```
fn arc_READONLY() {
    let data = Arc::new(vec![1, 2, 3, 4, 5]);

    let mut handles = vec![];

    for i in 0..5 {
        let data = Arc::clone(&data);
        handles.push(thread::spawn(move || {
            println!("Thread {}: sum = {}", i, data.iter().sum::());
        }));
    }

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

Example: Arc with Mutex for shared mutable state

This example shows how to use Arc with Mutex for shared mutable state while calling out the practical trade-offs.

```
struct SharedCounter {
    count: Arc<Mutex<usize>>,
}

impl SharedCounter {
    fn new() -> Self {
        Self {
            count: Arc::new(Mutex::new(0)),
        }
    }

    fn increment(&self) {
        let mut count = self.count.lock().unwrap();
        *count += 1;
    }

    fn get(&self) -> usize {
        *self.count.lock().unwrap()
    }

    fn clone_handle(&self) -> Self {
        Self {
            count: Arc::clone(&self.count),
        }
    }
}
```

```

        }

    }

fn arc_mutex_example() {
    let counter = SharedCounter::new();
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = counter.clone_handle();
        handles.push(thread::spawn(move || {
            for _ in 0..1000 {
                counter.increment();
            }
        }));
    }

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

    println!("Final count: {}", counter.get());
}

```

Example: Arc with RwLock for read-heavy workloads

This example shows how to arc with RwLock for read-heavy workloads while calling out the practical trade-offs.

```

struct Cache {
    data: Arc<RwLock<std::collections::HashMap<String, String>>,
}

impl Cache {
    fn new() -> Self {
        Self {
            data: Arc::new(RwLock::new(std::collections::HashMap::new())),
        }
    }

    fn get(&self, key: &str) -> Option<String> {
        self.data.read().unwrap().get(key).cloned()
    }

    fn set(&self, key: String, value: String) {
        self.data.write().unwrap().insert(key, value);
    }

    fn clone_handle(&self) -> Self {
        Self {
            data: Arc::clone(&self.data),
        }
    }
}

```

```

        }
    }

fn arc_rwlock_example() {
    let cache = Cache::new();

    // Writer thread
    let writer_cache = cache.clone_handle();
    let writer = thread::spawn(move || {
        for i in 0..100 {
            writer_cache.set(format!("key_{}", i), format!("value_{}", i));
            thread::sleep(Duration::from_millis(10));
        }
    });

    // Reader threads
    let mut readers = vec![];
    for id in 0..5 {
        let reader_cache = cache.clone_handle();
        readers.push(thread::spawn(move || {
            for i in 0..50 {
                if let Some(value) = reader_cache.get(&format!("key_{}", i * 2)) {
                    if id == 0 && i % 10 == 0 {
                        println!("Reader {}: {}", id, value);
                    }
                }
                thread::sleep(Duration::from_millis(5));
            }
        }));
    }

    writer.join().unwrap();
    for reader in readers {
        reader.join().unwrap();
    }
}

// Real-world: Thread pool with shared work queue
use std::sync::mpsc;

struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Job>,
}

type Job = Box<dyn FnOnce() + Send + 'static>;

struct Worker {
    id: usize,
    thread: Option<thread::JoinHandle<()>>,
}

impl Worker {

```

```
fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Self {
    let thread = thread::spawn(move || loop {
        let job = receiver.lock().unwrap().recv();

        match job {
            Ok(job) => {
                println!("Worker {} executing job", id);
                job();
            }
            Err(_) => {
                println!("Worker {} shutting down", id);
                break;
            }
        }
    });
}

Worker {
    id,
    thread: Some(thread),
}
}

impl ThreadPool {
    fn new(size: usize) -> Self {
        let (sender, receiver) = mpsc::channel();
        let receiver = Arc::new(Mutex::new(receiver));

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, Arc::clone(&receiver)));
        }

        ThreadPool { workers, sender }
    }

    fn execute<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
        self.sender.send(Box::new(f)).unwrap();
    }
}

impl Drop for ThreadPool {
    fn drop(&mut self) {
        for worker in &mut self.workers {
            if let Some(thread) = worker.thread.take() {
                thread.join().unwrap();
            }
        }
    }
}
```

```

}

fn main() {
    println!("== Arc Read-Only ==\n");
    arc_readonly();

    println!("\n== Arc + Mutex ==\n");
    arc_mutex_example();

    println!("\n== Arc + RwLock ==\n");
    arc_rwlock_example();

    println!("\n== Thread Pool ==\n");

    let pool = ThreadPool::new(4);

    for i in 0..10 {
        pool.execute(move || {
            println!("Job {} executing", i);
            thread::sleep(Duration::from_millis(100));
        });
    }

    thread::sleep(Duration::from_secs(2));
}

```

Arc vs Rc: - **Arc:** Atomic reference counting (thread-safe) - **Rc:** Non-atomic (single-threaded only) -
Performance: Rc is faster (no atomic operations) - **Use case:** Arc for multi-threaded, Rc for single-threaded

Pattern 2: Weak References and Cycles

Problem: Reference cycles cause memory leaks—parent and child both hold strong Rc pointers, reference count never reaches 0. Doubly-linked list with strong prev/next pointers leaks.

Solution: Use `Weak<T>` for back-references—doesn't increment strong count, breaks cycles. Child holds Weak to parent, parent holds strong (Rc) to child.

Why It Matters: Prevents production memory leaks. Tree with strong parent pointers: 100MB tree never freed.

Use Cases: Tree parent pointers, doubly-linked lists (prev pointer), observer pattern, caches (entries can expire), breaking any reference cycle, temporary references.

Example: Breaking Reference Cycles

Prevent memory leaks when data structures have circular references.

```

use std::rc::{Rc, Weak};
use std::cell::RefCell;
// Problem: This creates a reference cycle and leaks memory

```

```

#[derive(Debug)]
struct NodeWithCycle {
    value: i32,
    next: Option<Rc<RefCell<NodeWithCycle>>,
    prev: Option<Rc<RefCell<NodeWithCycle>>, // Strong reference - BAD!
}

// Solution: Use Weak for back-references
#[derive(Debug)]
struct Node {
    value: i32,
    next: Option<Rc<RefCell<Node>>,
    prev: Option<Weak<RefCell<Node>>, // Weak reference - GOOD!
}

impl Node {
    fn new(value: i32) -> Rc<RefCell<Self>> {
        Rc::new(RefCell::new(Node {
            value,
            next: None,
            prev: None,
        }))
    }
}

```

Example: Doubly-linked list with Weak

This example shows doubly-linked list with Weak to illustrate where the pattern fits best.

```

struct DoublyLinkedList {
    head: Option<Rc<RefCell<Node>>,
    tail: Option<Rc<RefCell<Node>>,
}

impl DoublyLinkedList {
    fn new() -> Self {
        Self {
            head: None,
            tail: None,
        }
    }

    fn push_back(&mut self, value: i32) {
        let new_node = Node::new(value);

        match self.tail.take() {
            Some(old_tail) => {
                old_tail.borrow_mut().next = Some(Rc::clone(&new_node));
                new_node.borrow_mut().prev = Some(Rc::downgrade(&old_tail));
                self.tail = Some(new_node);
            }
            None => {

```

```

        self.head = Some(Rc::clone(&new_node));
        self.tail = Some(new_node);
    }
}

fn print_forward(&self) {
    let mut current = self.head.as_ref().map(Rc::clone);

    while let Some(node) = current {
        print!("{} -> ", node.borrow().value);
        current = node.borrow().next.as_ref().map(Rc::clone);
    }
    println!("None");
}

fn print_backward(&self) {
    let mut current = self.tail.as_ref().map(Rc::clone);

    while let Some(node) = current {
        print!("{} -> ", node.borrow().value);
        current = node
            .borrow()
            .prev
            .as_ref()
            .and_then(|weak| weak.upgrade());
    }
    println!("None");
}
}

```

Example: Tree with parent pointers

This example shows how to tree with parent pointers in practice, emphasizing why it works.

```

#[derive(Debug)]
struct TreeNode {
    value: i32,
    parent: Option<Weak<RefCell<TreeNode>>>,
    children: Vec<Rc<RefCell<TreeNode>>>,
}

impl TreeNode {
    fn new(value: i32) -> Rc<RefCell<Self>> {
        Rc::new(RefCell::new(TreeNode {
            value,
            parent: None,
            children: Vec::new(),
        }))
    }
}

```

```

fn add_child(parent: &Rc<RefCell<TreeNode>>, child_value: i32) -> Rc<RefCell<TreeNode>> {
    let child = TreeNode::new(child_value);
    child.borrow_mut().parent = Some(Rc::downgrade(parent));
    parent.borrow_mut().children.push(Rc::clone(&child));
    child
}

fn print_path_to_root(node: &Rc<RefCell<TreeNode>>) {
    let mut path = Vec::new();
    let mut current = Some(Rc::clone(node));

    while let Some(node_rc) = current {
        path.push(node_rc.borrow().value);
        current = node_rc
            .borrow()
            .parent
            .as_ref()
            .and_then(|weak| weak.upgrade());
    }

    println!("Path to root: {:?}", path);
}
}

// Real-world: Observer pattern with Weak
trait Observer {
    fn notify(&self, message: &str);
}

struct Subject {
    observers: Vec<Weak<dyn Observer>>|,
}

impl Subject {
    fn new() -> Self {
        Self {
            observers: Vec::new(),
        }
    }

    fn attach(&mut self, observer: Weak<dyn Observer>) {
        self.observers.push(observer);
    }

    fn notify_all(&mut self, message: &str) {
        // Clean up dead observers and notify living ones
        self.observers.retain(|weak| {
            if let Some(observer) = weak.upgrade() {
                observer.notify(message);
                true // Keep
            } else {
                false // Remove dead observer
            }
        });
    }
}

```

```

    }

}

struct ConcreteObserver {
    id: usize,
}

impl Observer for ConcreteObserver {
    fn notify(&self, message: &str) {
        println!("Observer {} received: {}", self.id, message);
    }
}

// Real-world: Cache with weak references
struct WeakCache<K, V> {
    cache: std::collections::HashMap<K, Weak<V>>,
}

impl<K, V> WeakCache<K, V>
where
    K: Eq + std::hash::Hash,
{
    fn new() -> Self {
        Self {
            cache: std::collections::HashMap::new(),
        }
    }

    fn get(&mut self, key: &K) -> Option<Rc<V>> {
        self.cache.get(key).and_then(|weak| weak.upgrade())
    }

    fn insert(&mut self, key: K, value: Rc<V>) {
        self.cache.insert(key, Rc::downgrade(&value));
    }

    fn cleanup(&mut self) {
        self.cache.retain(|_, weak| weak.strong_count() > 0);
    }

    fn len(&self) -> usize {
        self.cache.len()
    }
}

fn main() {
    println!("==== Doubly-Linked List ===\n");

    let mut list = DoublyLinkedList::new();
    list.push_back(1);
    list.push_back(2);
    list.push_back(3);

    print!("Forward: ");
}

```

```

list.print_forward();
print!("Backward: ");
list.print_backward();

println!("\n==== Tree with Parent Pointers ===\n");

let root = TreeNode::new(1);
let child1 = TreeNode::add_child(&root, 2);
let child2 = TreeNode::add_child(&root, 3);
let grandchild = TreeNode::add_child(&child1, 4);

TreeNode::print_path_to_root(&grandchild);
TreeNode::print_path_to_root(&child2);

println!("\n==== Observer Pattern ===\n");

let mut subject = Subject::new();

let observer1 = Rc::new(ConcreteObserver { id: 1 });
let observer2 = Rc::new(ConcreteObserver { id: 2 });

subject.attach(Rc::downgrade(&observer1));
subject.attach(Rc::downgrade(&observer2));

subject.notify_all("First message");

drop(observer1); // Observer 1 goes away

subject.notify_all("Second message"); // Only observer 2 gets this

println!("\n==== Weak Cache ===\n");

let mut cache = WeakCache::new();

{
    let value = Rc::new("cached data".to_string());
    cache.insert("key1", Rc::clone(&value));
    println!("Cache size: {}", cache.len());

    if let Some(cached) = cache.get(&"key1") {
        println!("Found in cache: {}", cached);
    }

    // value dropped here
}

// Try to get after value is dropped
if cache.get(&"key1").is_none() {
    println!("Cache entry expired");
}

cache.cleanup();

```

```
    println!("Cache size after cleanup: {}", cache.len());
}
```

Weak Reference Patterns: - **Parent-child:** Child holds Weak to parent - **Observers:** Subject holds Weak to observers - **Cache:** Cache holds Weak to values - **Breaking cycles:** Use Weak for back-references

Pattern 3: Custom Smart Pointers

Problem: Need specialized pointer behavior beyond Box/Rc/Arc. Want to track access patterns (reads/writes) for debugging.

Solution: Implement `Deref` trait for `*` operator and method calls. Implement `Drop` for cleanup logic.

Why It Matters: Logging pointer reveals hot paths: 1000 reads, 10 writes → optimize for reads. Lazy initialization saves 500MB when feature unused.

Use Cases: Logging pointers for debugging, lazy initialization for expensive resources, copy-on-write for shared-immutable patterns, custom allocation tracking, instrumentation and profiling, domain-specific ownership.

Example: Implementing Custom Smart Pointers

Create custom pointer types with specialized behavior.

```
use std::ops::{Deref, DerefMut};
use std::fmt;
```

Example: Simple custom Box

This example keeps things simple while focusing on custom box to make the mechanics obvious.

```
struct MyBox<T> {
    data: *mut T,
}

impl<T> MyBox<T> {
    fn new(value: T) -> Self {
        let data = Box::into_raw(Box::new(value));
        MyBox { data }
    }
}

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        unsafe { &*self.data }
    }
}
```

```

    }
}

impl<T> DerefMut for MyBox<T> {
    fn deref_mut(&mut self) -> &mut Self::Target {
        unsafe { &mut *self.data }
    }
}

impl<T> Drop for MyBox<T> {
    fn drop(&mut self) {
        unsafe {
            drop(Box::from_raw(self.data));
        }
    }
}

```

Example: Logging pointer (tracks access)

This example shows logging pointer (tracks access) to illustrate where the pattern fits best.

```

struct LoggingPtr<T> {
    data: Box<T>,
    reads: std::cell::Cell<usize>,
    writes: std::cell::Cell<usize>,
}

impl<T> LoggingPtr<T> {
    fn new(value: T) -> Self {
        Self {
            data: Box::new(value),
            reads: std::cell::Cell::new(0),
            writes: std::cell::Cell::new(0),
        }
    }

    fn get_stats(&self) -> (usize, usize) {
        (self.reads.get(), self.writes.get())
    }
}

impl<T> Deref for LoggingPtr<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        self.reads.set(self.reads.get() + 1);
        &self.data
    }
}

impl<T> DerefMut for LoggingPtr<T> {

```

```

fn deref_mut(&mut self) -> &mut Self::Target {
    self.writes.set(self.writes.get() + 1);
    &mut self.data
}
}

```

Example: Lazy initialization pointer

This example shows how to lazy initialization pointer in practice, emphasizing why it works.

```

struct Lazy<T, F>
where
    F: FnOnce() -> T,
{
    value: std::cell::UnsafeCell<Option<T>>,
    init: std::cell::UnsafeCell<Option<F>>,
}

impl<T, F> Lazy<T, F>
where
    F: FnOnce() -> T,
{
    fn new(init: F) -> Self {
        Self {
            value: std::cell::UnsafeCell::new(None),
            init: std::cell::UnsafeCell::new(Some(init)),
        }
    }

    fn get(&self) -> &T {
        unsafe {
            if (*self.value.get()).is_none() {
                let init = (*self.init.get()).take().unwrap();
                *self.value.get() = Some(init());
            }

            (*self.value.get()).as_ref().unwrap()
        }
    }
}

// Real-world: Reference-counted string (like Arc<str> but custom)
struct RcStr {
    data: *mut RcStrInner,
}

struct RcStrInner {
    ref_count: std::sync::atomic::AtomicUsize,
    data: str,
}

impl RcStr {

```

```
fn new(s: &str) -> Self {
    let layout = std::alloc::Layout::from_size_align(
        std::mem::size_of::<std::sync::atomic::AtomicUsize>() + s.len(),
        std::mem::align_of::<std::sync::atomic::AtomicUsize>(),
    )
    .unwrap();
}

unsafe {
    let ptr = std::alloc::alloc(layout) as *mut RcStrInner;

    std::ptr::write(
        &mut (*ptr).ref_count,
        std::sync::atomic::AtomicUsize::new(1),
    );
}

let data_ptr = (&mut (*ptr).data) as *mut str as *mut u8;
std::ptr::copy_nonoverlapping(s.as_ptr(), data_ptr, s.len());

RcStr { data: ptr }
}

fn as_str(&self) -> &str {
    unsafe { &(*self.data).data }
}

fn ref_count(&self) -> usize {
    unsafe { (*self.data).ref_count.load(std::sync::atomic::Ordering::Acquire) }
}

impl Clone for RcStr {
    fn clone(&self) -> Self {
        unsafe {
            (*self.data)
                .ref_count
                .fetch_add(1, std::sync::atomic::Ordering::Release);
        }
        RcStr { data: self.data }
    }
}

impl Drop for RcStr {
    fn drop(&mut self) {
        unsafe {
            if (*self.data)
                .ref_count
                .fetch_sub(1, std::sync::atomic::Ordering::Release)
                == 1
            {
                let layout = std::alloc::Layout::from_size_align(
                    std::mem::size_of::<std::sync::atomic::AtomicUsize>() +
                    (*self.data).data.len(),

```

```

        std::mem::align_of::<std::sync::atomic::AtomicUsize>(),
    )
    .unwrap();

    std::alloc::dealloc(self.data as *mut u8, layout);
}
}
}

impl Deref for RcStr {
    type Target = str;

    fn deref(&self) -> &Self::Target {
        self.as_str()
    }
}

impl fmt::Display for RcStr {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "{}", self.as_str())
    }
}
}

```

Example: Owned-or-borrowed pointer

This example shows owned-or-borrowed pointer to illustrate where the pattern fits best.

```

enum Cow<'a, T: 'a + ToOwned + ?Sized> {
    Borrowed(&'a T),
    Owned(<T as ToOwned>::Owned),
}

impl<'a, T> Cow<'a, T>
where
    T: ToOwned + ?Sized,
{
    fn to_mut(&mut self) -> &mut <T as ToOwned>::Owned {
        match self {
            Cow::Owned(owned) => owned,
            Cow::Borrowed(borrowed) => {
                *self = Cow::Owned(borrowed.to_owned());
                match self {
                    Cow::Owned(owned) => owned,
                    _ => unreachable!(),
                }
            }
        }
    }
}

```

```

fn main() {
    println!("==== Custom MyBox ===\n");

    let mut my_box = MyBox::new(42);
    println!("Value: {}", *my_box);
    *my_box = 100;
    println!("Updated: {}", *my_box);

    println!("\n==== Logging Pointer ===\n");

    let mut logged = LoggingPtr::new(String::from("hello"));

    let _read1 = logged.len();
    let _read2 = logged.chars().count();
    logged.push_str(" world");

    let (reads, writes) = logged.get_stats();
    println!("Reads: {}, Writes: {}", reads, writes);
    println!("Value: {}", *logged);

    println!("\n==== Lazy Initialization ===\n");

    let lazy = Lazy::new(|| {
        println!("Initializing expensive computation...");
        42
    });

    println!("Before access");
    println!("Value: {}", lazy.get());
    println!("Value again: {}", lazy.get()); // No re-initialization

    println!("\n==== Custom RcStr ===\n");

    let s1 = RcStr::new("Hello, world!");
    println!("s1: {}", s1);
    println!("s1 ref count: {}", s1.ref_count());

    let s2 = s1.clone();
    println!("s1 ref count after clone: {}", s1.ref_count());
    println!("s2 ref count: {}", s2.ref_count());

    drop(s2);
    println!("s1 ref count after drop s2: {}", s1.ref_count());
}

```

Custom Smart Pointer Requirements: - **Deref:** Enable `*` and method calls - **Drop:** Clean up resources - **Clone** (optional): For reference counting - **Send/Sync** (optional): For thread safety

Pattern 4: Intrusive Data Structures

Problem: Standard data structures waste memory with separate node allocations. Poor cache locality from scattered allocations.

Solution: Embed pointers directly in data nodes. Use raw pointers (*mut T) for manual management.

Why It Matters: Intrusive LRU cache: 50% memory savings vs standard implementation. Better cache locality: 2x faster on sequential access.

Use Cases: LRU caches (web servers, databases), kernel data structures (Linux intrusive lists), high-performance queues, embedded systems, memory pools, any cache-critical structure.

Example: Intrusive Linked Lists

Efficient linked lists where the nodes are embedded in objects.

```
use std::ptr;
use std::marker::PhantomData;
```

Example: Intrusive singly-linked list

This example shows how to intrusive singly-linked list while calling out the practical trade-offs.

```
struct IntrusiveList<T> {
    head: *mut ListNode<T>,
    _phantom: PhantomData<T>,
}

struct ListNode<T> {
    next: *mut ListNode<T>,
    data: T,
}

impl<T> IntrusiveList<T> {
    fn new() -> Self {
        Self {
            head: ptr::null_mut(),
            _phantom: PhantomData,
        }
    }

    fn push_front(&mut self, data: T) {
        let node = Box::into_raw(Box::new(ListNode {
            next: self.head,
            data,
        }));
        self.head = node;
    }
}
```

```

fn pop_front(&mut self) -> Option<T> {
    if self.head.is_null() {
        return None;
    }

    unsafe {
        let node = Box::from_raw(self.head);
        self.head = node.next;
        Some(node.data)
    }
}

fn iter(&self) -> IntrusiveListIter<T> {
    IntrusiveListIter {
        current: self.head,
        _phantom: PhantomData,
    }
}
}

impl<T> Drop for IntrusiveList<T> {
    fn drop(&mut self) {
        while self.pop_front().is_some() {}
    }
}

struct IntrusiveListIter<'a, T> {
    current: *mut ListNode<T>,
    _phantom: PhantomData<&'a T>,
}

impl<'a, T> Iterator for IntrusiveListIter<'a, T> {
    type Item = &'a T;

    fn next(&mut self) -> Option<Self::Item> {
        if self.current.is_null() {
            None
        } else {
            unsafe {
                let data = &(*self.current).data;
                self.current = (*self.current).next;
                Some(data)
            }
        }
    }
}

// Real-world: Intrusive doubly-linked list for LRU cache
struct LruCache<K, V> {
    map: std::collections::HashMap<K, *mut LruNode<K, V>>,
    head: *mut LruNode<K, V>,
    tail: *mut LruNode<K, V>,
    capacity: usize,
}

```

```

}

struct LruNode<K, V> {
    key: K,
    value: V,
    prev: *mut LruNode<K, V>,
    next: *mut LruNode<K, V>,
}

impl<K, V> LruCache<K, V>
where
    K: Eq + std::hash::Hash + Clone,
{
    fn new(capacity: usize) -> Self {
        Self {
            map: std::collections::HashMap::new(),
            head: ptr::null_mut(),
            tail: ptr::null_mut(),
            capacity,
        }
    }

    fn get(&mut self, key: &K) -> Option<&V> {
        let node_ptr = *self.map.get(key)?;

        unsafe {
            // Move to front
            self.detach(node_ptr);
            self.attach_front(node_ptr);

            Some(&(*node_ptr).value)
        }
    }

    fn put(&mut self, key: K, value: V) {
        if let Some(&node_ptr) = self.map.get(&key) {
            unsafe {
                (*node_ptr).value = value;
                self.detach(node_ptr);
                self.attach_front(node_ptr);
            }
            return;
        }

        // Evict if at capacity
        if self.map.len() >= self.capacity {
            unsafe {
                if !self.tail.is_null() {
                    let tail_key = (*self.tail).key.clone();
                    self.detach(self.tail);
                    drop(Box::from_raw(self.tail));
                    self.map.remove(&tail_key);
                }
            }
        }
    }
}

```

```

        }

    }

    // Create new node
    let node = Box::into_raw(Box::new(LruNode {
        key: key.clone(),
        value,
        prev: ptr::null_mut(),
        next: ptr::null_mut(),
    }));

    self.map.insert(key, node);
    unsafe {
        self.attach_front(node);
    }
}

unsafe fn detach(&mut self, node: *mut LruNode<K, V>) {
    let prev = (*node).prev;
    let next = (*node).next;

    if !prev.is_null() {
        (*prev).next = next;
    } else {
        self.head = next;
    }

    if !next.is_null() {
        (*next).prev = prev;
    } else {
        self.tail = prev;
    }

    (*node).prev = ptr::null_mut();
    (*node).next = ptr::null_mut();
}

unsafe fn attach_front(&mut self, node: *mut LruNode<K, V>) {
    (*node).next = self.head;
    (*node).prev = ptr::null_mut();

    if !self.head.is_null() {
        (*self.head).prev = node;
    }

    self.head = node;

    if self.tail.is_null() {
        self.tail = node;
    }
}
}

```

```

impl<K, V> Drop for LruCache<K, V> {
    fn drop(&mut self) {
        unsafe {
            let mut current = self.head;
            while !current.is_null() {
                let next = (*current).next;
                drop(Box::from_raw(current));
                current = next;
            }
        }
    }
}

fn main() {
    println!("==== Intrusive List ====\n");

    let mut list = IntrusiveList::new();

    list.push_front(3);
    list.push_front(2);
    list.push_front(1);

    println!("List items:");
    for item in list.iter() {
        println!("  {}", item);
    }

    while let Some(item) = list.pop_front() {
        println!("Popped: {}", item);
    }

    println!("\n==== LRU Cache ====\n");

    let mut cache = LruCache::new(3);

    cache.put("a", 1);
    cache.put("b", 2);
    cache.put("c", 3);

    println!("Get a: {:?}", cache.get(&"a"));
    println!("Get b: {:?}", cache.get(&"b"));

    cache.put("d", 4); // Evicts c (least recently used)

    println!("Get c (evicted): {:?}", cache.get(&"c"));
    println!("Get d: {:?}", cache.get(&"d"));
}

```

Intrusive List Benefits: - **No extra allocations:** Node is part of data - **Cache-friendly:** Better locality - **Constant-time removal:** No search needed - **Use case:** Kernel data structures, high-performance caches

Pattern 5: Reference Counting Optimization

Problem: Reference counting adds overhead—every `Rc::clone` increments counter, every drop decrements. Excessive cloning wastes CPU cycles.

Solution: Borrow (`&Rc`) instead of clone when possible. Use `try_unwrap` to get owned data if sole owner.

Why It Matters: `Rc::clone` costs ~10ns, borrow costs ~0ns. Hot loop with 1M iterations: `Rc` clone wastes 10ms, borrow is free.

Use Cases: Hot loops (avoid clones), string interning (deduplication), temporary access (use borrows), sole ownership extraction (`try_unwrap`), conditional mutation (Cow), profiling-guided optimization.

Example: Reference Counting Optimizations

Reduce the overhead of reference counting operations.

```
use std::rc::Rc;
use std::sync::Arc;
```

Example: Avoid unnecessary clones

This example shows how to avoid unnecessary clones in practice, emphasizing why it works.

```
fn inefficient_clones(data: &Rc<Vec<i32>>) {
    // Bad: Clone for every operation
    let clone1 = Rc::clone(data);
    println!("Length: {}", clone1.len());

    let clone2 = Rc::clone(data);
    println!("First: {}", clone2[0]);
}

fn efficient_borrows(data: &Rc<Vec<i32>>) {
    // Good: Borrow directly
    println!("Length: {}", data.len());
    println!("First: {}", data[0]);
}
```

Example: Make owned data when possible

This example shows how to make owned data when possible in practice, emphasizing why it works.

```
fn make_owned(data: Rc<Vec<i32>>) -> Vec<i32> {
    // Try to unwrap if we're the only owner
```

```
Rc::try_unwrap(data).unwrap_or_else(|rc| (*rc).clone())
}
```

Example: Batch reference counting updates

This example shows how to batch reference counting updates in practice, emphasizing why it works.

```
fn batch_updates() {
    let data = Rc::new(vec![1, 2, 3, 4, 5]);

    // Bad: Multiple increments/decrements
    {
        let _clone1 = Rc::clone(&data);
        let _clone2 = Rc::clone(&data);
        let _clone3 = Rc::clone(&data);
    }

    // Better: Pass references when possible
    {
        process_data(&data);
        process_data(&data);
        process_data(&data);
    }
}

fn process_data(data: &Rc<Vec<i32>>) {
    println!("Processing {} items", data.len());
}
```

Example: Use Cow for clone-on-write

This example shows how to use Cow for clone-on-write in practice, emphasizing why it works.

```
use std::borrow::Cow;

fn process_string(s: Cow<str>) -> Cow<str> {
    if s.contains("replace") {
        // Need to modify - convert to owned
        Cow::Owned(s.replace("replace", "replaced"))
    } else {
        // No modification - keep borrowed
        s
    }
}

// Real-world: String interning with Rc
use std::collections::HashMap;

struct StringInterner {
    map: HashMap<String, Rc<str>>,
```

```

}

impl StringIntern {
    fn new() -> Self {
        Self {
            map: HashMap::new(),
        }
    }

    fn intern(&mut self, s: &str) -> Rc<str> {
        if let Some(interned) = self.map.get(s) {
            // Already interned - just clone Rc
            Rc::clone(interned)
        } else {
            // Not interned - create new Rc<str>
            let rc: Rc<str> = Rc::from(s);
            self.map.insert(s.to_string(), Rc::clone(&rc));
            rc
        }
    }

    fn len(&self) -> usize {
        self.map.len()
    }

    fn memory_saved(&self, total_strings: usize) -> usize {
        // Estimate memory saved by interning
        let unique = self.len();
        let duplicates = total_strings - unique;
        duplicates * std::mem::size_of::<String>()
    }
}

```

Example: Weak upgrades to avoid clones

This example shows how to weak upgrades to avoid clones in practice, emphasizing why it works.

```

use std::rc::Weak;

struct Observer {
    subject: Weak<Vec<i32>>,
}

impl Observer {
    fn observe(&self) {
        // Upgrade temporarily, don't keep strong reference
        if let Some(subject) = self.subject.upgrade() {
            println!("Observing: {} items", subject.len());
        }
    }
}

```

```
    }
}
```

Example: Arc performance comparison

This example shows how to arc performance comparison while calling out the practical trade-offs.

```
fn arc_performance_test() {
    use std::time::Instant;

    let data = Arc::new(vec![0; 1_000_000]);

    // Test 1: Many Arc clones
    let start = Instant::now();
    let mut clones = Vec::new();
    for _ in 0..1000 {
        clones.push(Arc::clone(&data));
    }
    let clone_time = start.elapsed();

    drop(clones);

    // Test 2: Many borrows (no cloning)
    let start = Instant::now();
    for _ in 0..1000 {
        let _borrow = &data;
    }
    let borrow_time = start.elapsed();

    println!("Arc clone time: {:?}", clone_time);
    println!("Borrow time: {:?}", borrow_time);
    println!("Speedup: {:.2}x", clone_time.as_nanos() as f64 / borrow_time.as_nanos() as f64);
}
```

Example: Rc vs owned in hot loops

This example shows how to rc vs owned in hot loops while calling out the practical trade-offs.

```
fn rc_vs_owned_benchmark() {
    use std::time::Instant;

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

    // With Rc (reference counting overhead)
    let start = Instant::now();
    let rc_data = Rc::new(data.clone());
    for _ in 0..1_000_000 {
        let _clone = Rc::clone(&rc_data);
    }
}
```

```

let rc_time = start.elapsed();

// With owned (no overhead but more memory)
let start = Instant::now();
for _ in 0..1_000_000 {
    let _clone = data.clone();
}
let owned_time = start.elapsed();

println!("Rc clones: {:?}", rc_time);
println!("Owned clones: {:?}", owned_time);
}

fn main() {
    println!("==== Efficient vs Inefficient ====\n");

    let data = Rc::new(vec![1, 2, 3, 4, 5]);
    println!("Ref count: {}", Rc::strong_count(&data));

    efficient_borrows(&data);
    println!("After borrows: {}", Rc::strong_count(&data));

    println!("\n==== Make Owned ====\n");

    let data = Rc::new(vec![1, 2, 3]);
    println!("Initial ref count: {}", Rc::strong_count(&data));

    let owned = make_owned(data);
    println!("Owned data: {:?}", owned);

    println!("\n==== String Interning ====\n");

    let mut interner = StringInterner::new();

    let strings = vec!["hello", "world", "hello", "rust", "world", "hello"];
    let mut interned = Vec::new();

    for s in &strings {
        interned.push(interner.intern(s));
    }

    println!("Total strings: {}", strings.len());
    println!("Unique strings: {}", interner.len());
    println!("Memory saved: ~{} bytes", interner.memory_saved(strings.len()));

    println!("\n==== Arc Performance ====\n");
    arc_performance_test();

    println!("\n==== Rc vs Owned ====\n");
    rc_vs_owned_benchmark();
}

```

Optimization Strategies: 1. **Borrow instead of clone:** Use `&Rc<T>` instead of `Rc::clone()` 2. **try_unwrap:** Get owned data if only owner 3. **Weak references:** Avoid strong references when possible 4. **String interning:** Share common strings 5. **Cow:** Clone-on-write for conditional modification 6. **Profile:** Measure before optimizing

Summary

This chapter covered smart pointer patterns in Rust:

1. **Box, Rc, Arc:** Heap allocation, single-threaded sharing, thread-safe sharing
2. **Weak References:** Prevent cycles, observer pattern, caches
3. **Custom Smart Pointers:** Deref, Drop, logging, lazy initialization
4. **Intrusive Structures:** Embedded pointers, LRU cache, kernel-style lists
5. **RC Optimization:** Avoid clones, try_unwrap, string interning, Cow

Key Takeaways: - **Box:** Heap allocation, recursive types, trait objects - **Rc:** Single-threaded shared ownership - **Arc:** Thread-safe shared ownership (atomic overhead) - **Weak:** Break cycles, prevent memory leaks - **Custom pointers:** Deref + Drop for domain logic - **Intrusive:** Embed pointers for efficiency

Smart Pointer Selection Guide:

| Pattern | Use Case | Thread-Safe Overhead | |
|--------------|----------------------------------|----------------------|--------------------|
| Box | Heap allocation, recursion | No | Minimal |
| Rc | Shared ownership (single-thread) | No | Reference counting |
| Arc | Shared ownership (multi-thread) | Yes | Atomic RC |
| Weak | Break cycles, observers | Depends | Weak counter |
| RefCell + Rc | Interior mutability | No | Runtime checks |
| Mutex + Arc | Shared mutable state | Yes | Lock overhead |

Common Patterns: - **Rc<RefCell>:** Single-threaded shared mutation - **Arc<Mutex>:** Multi-threaded shared mutation - **Arc<RwLock>:** Read-heavy multi-threaded - **Weak:** Observer, cache, parent pointers

Performance Tips: - Borrow `&Rc<T>` instead of cloning - Use `try_unwrap` to get owned data - Intern strings to reduce duplicates - Profile before optimizing RC - Consider `Cow` for conditional cloning

Memory Leak Prevention: - Use Weak for back-references - Break cycles in Drop - Use Weak in observer pattern - Profile with valgrind/leak sanitizer

Safety: - Smart pointers are safe (type-checked) - Custom pointers need unsafe (be careful!) - Rc/Arc prevent use-after-free - Weak prevents dangling pointers

Unsafe Rust Patterns

Unsafe Rust is not a separate language. It's a escape hatch that allows you to tell the compiler "I know what I'm doing, trust me on this." While Rust's safety guarantees are powerful, they can not express

every valid program. Low-level systems programming, hardware interaction, foreign function interfaces, and certain performance optimizations require operations that the compiler cannot verify as safe.

The `unsafe` keyword doesn't disable Rust's safety checks; it expands what you're allowed to do. You're still protected from type confusion, use-after-free in safe code surrounding your unsafe blocks, and many other pitfalls. What `unsafe` enables are five specific superpowers that the compiler cannot verify automatically:

1. **Dereferencing raw pointers** – Reading or writing through `*const T` and `*mut T`
2. **Calling unsafe functions** – Functions that have unchecked preconditions
3. **Accessing or modifying static mutable variables** – Global mutable state
4. **Implementing unsafe traits** – Traits with invariants the compiler can't verify
5. **Accessing fields of unions** – Type-punning and low-level tricks

This chapter explores patterns for using unsafe code responsibly. The goal is not to avoid unsafe code—that would be impossible for low-level libraries—but to **build safe abstractions over unsafe foundations**. Every unsafe block should be surrounded by safe APIs that enforce invariants, document preconditions, and prevent misuse.

The patterns we'll explore show how to:

- Manipulate raw pointers safely while maintaining invariants
- Interface with C code without compromising Rust's safety
- Handle uninitialized memory correctly using `MaybeUninit`
- Use transmute sparingly and correctly
- Build safe APIs that encapsulate unsafe internals

The golden rule: Unsafe code is not about being unsafe. It's about maintaining safety invariants that the compiler cannot verify. Every unsafe block should have a comment explaining why it's correct. If you can't explain why it's safe, it probably isn't.

Pattern 1: Raw Pointer Manipulation

Problem: Need manual memory management for custom data structures. Borrow checker can't express bidirectional relationships (tree with parent pointers).

Solution: Use raw pointers (`*const T`, `*mut T`) with explicit safety. Creating pointers is safe, dereferencing requires `unsafe`.

Why It Matters: Enables implementing `Vec`, `LinkedList`, `HashMap` from scratch. Custom allocators power memory pools.

Use Cases: Custom collections (linked lists, trees, graphs), custom allocators and memory pools, memory-mapped I/O, FFI with C code, intrusive data structures, zero-copy parsing, hardware drivers.

Raw pointers (`*const T` and `*mut T`) are Rust's unmanaged pointers. Unlike references, they have no borrowing rules, no lifetime tracking, and no automatic dereferencing. They're what you get when you need manual memory management or when interfacing with systems that don't speak Rust's language of ownership.

When raw pointers are necessary:

- Implementing custom data structures (linked lists, trees with parent pointers)
- FFI with C code that expects raw pointers
- Memory-mapped I/O for hardware access
- Custom allocators and memory pools
- Performance-critical code avoiding bounds checks

The key difference from references: raw pointers don't promise validity. A `*const T` might point to valid memory, freed memory, or complete garbage. The compiler won't stop you from creating them, but dereferencing requires `unsafe` because that's where things can go wrong.

Example: Raw Pointer Usage

Creating raw pointers is safe—it's just taking an address. The danger comes when you dereference them, because you're asserting "this memory is valid and properly aligned," and the compiler can't verify that claim.

```
fn raw_pointer_basics() {
    let mut num = 42;
    let r1: *const i32 = &num;           // Immutable raw pointer
    let r2: *mut i32 = &mut num;        // Mutable raw pointer
    let address = 0x12345usize;
    let r3 = address as *const i32;     // Might point to invalid memory!

    unsafe {
        println!("r1 points to: {}", *r1);
        *r2 = 100;
        println!("num is now: {}", num);
        // Dereferencing r3 would be UB - it points to random memory!
    }
}
```

Why this pattern exists: Sometimes you need to store pointers in data structures where the borrow checker can't track the relationships. A tree node with a parent pointer, for example—the parent outlives the child, but Rust's borrow checker can't express that bidirectional relationship without causing issues.

Example: Pointer Arithmetic

Pointer arithmetic lets you navigate through memory by calculating offsets. This is fundamental for implementing custom collections and working with contiguous memory layouts. However, it's also where many C bugs come from—off-by-one errors lead to buffer overruns, corrupted data, and security vulnerabilities.

```
fn pointer_arithmetic() {
    let arr = [1, 2, 3, 4, 5];
    let ptr: *const i32 = arr.as_ptr();

    unsafe {
        for i in 0..arr.len() {
            let element_ptr = ptr.add(i); // Equivalent to ptr + i * sizeof(i32)
            println!("Element {}: {}", i, *element_ptr);
        }

        let third = ptr.add(2); // Points to third element
        println!("Third element: {}", *third);
    }
}
```

```

        // ptr.add(10) would be UB - out of bounds!
    }
}

```

The critical rule: Pointer arithmetic must stay within the bounds of the original allocation (or one byte past the end). Going beyond is undefined behavior even if you don't dereference. The CPU's memory protection won't save you—UB means the compiler can assume it never happens and optimize accordingly, leading to bizarre bugs.

When to use this: Implementing iterators over custom collections, parsing binary protocols, working with memory-mapped files where you need to jump to specific offsets.

Example: Building a Raw Vec-like Structure

Let's build a simplified vector to understand how raw pointers, allocation, and unsafe come together. This pattern appears in countless Rust libraries that need custom memory management.

The strategy: separate allocation from element storage. `RawVec` handles raw memory, while a higher-level `Vec` (not shown) would track initialization.

```

use std::alloc::{alloc, dealloc, realloc, Layout};
use std::ptr;

/// Manages raw memory allocation for a vector.
/// Does NOT track which elements are initialized!
pub struct RawVec<T> {
    ptr: *mut T,      // Pointer to allocated memory
    cap: usize,       // Capacity (number of T that fit)
}

impl<T> RawVec<T> {
    /// Creates an empty RawVec with no allocation.
    pub fn new() -> Self {
        RawVec {
            ptr: std::ptr::null_mut(), // null_mut() is a safe operation
            cap: 0,
        }
    }

    /// Allocates memory for `cap` elements.
    pub fn with_capacity(cap: usize) -> Self {
        let layout = Layout::array::<T>(cap).unwrap();
        let ptr = unsafe { alloc(layout) as *mut T };

        if ptr.is_null() {
            panic!("Allocation failed");
        }

        RawVec { ptr, cap }
    }

    /// Doubles capacity, or sets it to 1 if currently zero.
}

```

```

pub fn grow(&mut self) {
    let new_cap = if self.cap == 0 { 1 } else { self.cap * 2 };
    let new_layout = Layout::array::<T>(new_cap).unwrap();

    let new_ptr = if self.cap == 0 {
        unsafe { alloc(new_layout) as *mut T }
    } else {
        let old_layout = Layout::array::<T>(self.cap).unwrap();
        unsafe {
            realloc(
                self.ptr as *mut u8, // realloc works with u8 pointers
                old_layout,
                new_layout.size()
            ) as *mut T
        }
    };
}

if new_ptr.is_null() {
    panic!("Allocation failed");
}

self.ptr = new_ptr;
self.cap = new_cap;
}

pub fn ptr(&self) -> *mut T {
    self.ptr
}

pub fn cap(&self) -> usize {
    self.cap
}
}

impl<T> Drop for RawVec<T> {
    fn drop(&mut self) {
        if self.cap != 0 {
            let layout = Layout::array::<T>(self.cap).unwrap();
            unsafe {
                dealloc(self.ptr as *mut u8, layout);
            }
        }
    }
}

```

Why this pattern? Separating raw allocation from element management clarifies responsibilities.

RawVec handles memory, higher-level code handles Drop for elements. This is exactly how std::vec::Vec is implemented.

Safety invariants we maintain:

1. `ptr` is either null (when `cap == 0`) or points to valid allocated memory
2. `cap` accurately reflects the allocation size
3. Deallocation uses the same layout as allocation
4. We never dereference `ptr` here (no assumptions about initialization)

Example: Null Pointer Optimization with NonNull

`NonNull<T>` is a raw pointer wrapper that guarantees non-nullness. This enables the “null pointer optimization”—`Option<NonNull<T>>` has the same size as `*mut T` because it can use null as the `None` representation.

```
use std::ptr::NonNull;

/// A node in a linked list using NonNull for efficiency.
struct Node<T> {
    value: T,
    next: Option<NonNull<Node<T>>>, // Same size as *mut Node<T>
}

impl<T> Node<T> {
    fn new(value: T) -> Self {
        Node { value, next: None }
    }

    fn set_next(&mut self, next: NonNull<Node<T>>) {
        self.next = Some(next);
    }
}

/// A simple singly-linked list.
struct LinkedList<T> {
    head: Option<NonNull<Node<T>>>,
    tail: Option<NonNull<Node<T>>>,
    len: usize,
}

impl<T> LinkedList<T> {
    fn new() -> Self {
        LinkedList {
            head: None,
            tail: None,
            len: 0,
        }
    }

    fn push_back(&mut self, value: T) {
        let node = Box::new(Node::new(value));
        let node_ptr = NonNull::new(Box::into_raw(node)).unwrap();

        unsafe {
            if let Some(mut tail) = self.tail {
                tail.as_mut().next = Some(node_ptr);
            } else {
                self.head = Some(node_ptr);
            }
        }

        self.tail = Some(node_ptr);
    }
}
```

```

        }

        self.len += 1;
    }

}

impl<T> Drop for LinkedList<T> {
    fn drop(&mut self) {
        let mut current = self.head;

        while let Some(node_ptr) = current {
            unsafe {
                let node = Box::from_raw(node_ptr.as_ptr());
                current = node.next;
            }
        }
    }
}

```

Why NonNull? Three benefits: 1. **Memory efficiency:** `Option<NonNull<T>>` is the same size as a raw pointer 2. **Safety marker:** Guarantees non-null, catching bugs at creation time 3. **Covariance:** Unlike `*mut T, NonNull<T>` is covariant over `T`

When to use: Linked data structures, graph nodes, intrusive collections. Anywhere you'd use raw pointers but can guarantee non-null.

Pattern 2: FFI and C Interop

Problem: Need to call C libraries (operating system, drivers, databases, graphics). C has no concept of ownership, borrowing, or lifetimes.

Solution: Use `extern "C"` to declare/export functions with C ABI. `#[repr(C)]` for compatible struct layout.

Why It Matters: Unlocks entire C ecosystem (millions of libraries). System programming requires OS APIs (all C).

Use Cases: OS APIs (filesystem, networking, processes), database bindings (PostgreSQL, MySQL, SQLite), graphics libraries (OpenGL, Vulkan), compression (zlib, lz4), cryptography (OpenSSL), embedded drivers, legacy C code integration.

Foreign Function Interface (FFI) is how Rust talks to C, and by extension, the vast ecosystem of C libraries. This is unavoidable in systems programming—the operating system, graphics drivers, databases, and countless libraries all speak C at their boundaries.

The challenge: C has no concept of Rust's ownership, borrowing, or lifetimes. A `char*` in C could be stack-allocated, heap-allocated, a string literal, or dangling memory. Rust must bridge this gap carefully.

Key FFI principles: - Rust can call C functions, C can call Rust functions marked `extern "C"` - Types must have compatible memory layouts (use `#[repr(C)]`) - Ownership transfer must be explicit and documented - String encoding matters: C uses null-terminated, Rust uses UTF-8 slices

Example: Basic C Function Binding

Declaring external C functions is straightforward, but calling them requires `unsafe` because Rust can't verify their contracts.

```
//=====
// Declare external C functions from the standard C library
//=====

extern "C" {
    fn abs(input: i32) -> i32;
    fn strlen(s: *const std::os::raw::c_char) -> usize;
    fn malloc(size: usize) -> *mut std::os::raw::c_void;
    fn free(ptr: *mut std::os::raw::c_void);
}

fn use_c_functions() {
    unsafe {
        let result = abs(-42);
        println!("abs(-42) = {}", result);

        let c_str = b"Hello\0";
        let len = strlen(c_str.as_ptr() as *const std::os::raw::c_char);
        println!("String length: {}", len);

        let ptr = malloc(100);
        if !ptr.is_null() {
            free(ptr);
        }
    }
}
```

Why unsafe? The compiler can't verify that: - `strlen` won't read past the end of the string - `malloc` returns are checked before use - `free` is called exactly once per `malloc`

These are contracts you must uphold, documented in C headers and man pages.

Example: Working with C Strings

C strings are null-terminated byte arrays. Rust strings are UTF-8 slices with explicit length. Converting between them requires care to avoid buffer overruns and encoding issues.

```
use std::ffi::CStr, CString;
use std::os::raw::c_char;

/// Converts a Rust string to a C-owned string.
/// Caller must free with free_rust_c_string().
fn rust_to_c_string(s: &str) -> *mut c_char {
```

```

let c_string = CString::new(s).expect("CString::new failed");
c_string.into_raw() // Transfers ownership to caller
}

/// Converts a C string to a Rust String (copying data).
///
/// # Safety
/// - `c_str` must be a valid null-terminated C string
/// - The memory must remain valid for the duration of this call
unsafe fn c_to_rust_string(c_str: *const c_char) -> String {
    let c_str = CStr::from_ptr(c_str);
    c_str.to_string_lossy().into_owned()
}

/// Frees a C string created by rust_to_c_string().
///
/// # Safety
/// - `ptr` must have been created by rust_to_c_string()
/// - `ptr` must not be used after this call (use-after-free)
unsafe fn free_rust_c_string(ptr: *mut c_char) {
    if !ptr.is_null() {
        let _ = CString::from_raw(ptr);
    }
}

//=====
// Example usage
//=====

fn c_string_example() {
    let c_str = rust_to_c_string("Hello from Rust");

    unsafe {
        let rust_str = c_to_rust_string(c_str);
        println!("Back to Rust: {}", rust_str);
        free_rust_c_string(c_str);
    }
}

```

Critical pattern: Ownership transfer must be explicit. `into_raw()` says “Rust, stop tracking this.” `from_raw()` says “Resume tracking so you can drop it.” Missing either causes memory leaks or double-frees.

Encoding issues: C strings are byte arrays, not necessarily UTF-8. Use `to_string_lossy()` to handle invalid UTF-8 gracefully, or `to_str()` to fail fast.

Example: C Struct Interop

When passing structs between Rust and C, memory layout must match exactly. `#[repr(C)]` tells Rust to use C’s layout rules instead of optimizing field order.

```

use std::os::raw::{c_int, c_char};

/// A point with C-compatible layout.
#[repr(C)]
struct Point {
    x: c_int, // Use c_int, not i32 (they're the same on most platforms but not guaranteed)
    y: c_int,
}

/// A person struct that C can understand.
#[repr(C)]
struct Person {
    name: *const c_char, // C expects raw pointers, not &str
    age: c_int,
    height: f64,
}

/// An enum with explicit discriminant values for C.
#[repr(C)]
enum Status {
    Success = 0,
    Error = 1,
    Pending = 2,
}

//=====
// Declare C functions that work with these structs
//=====

extern "C" {
    fn process_point(point: *const Point) -> c_int;
    fn create_person(name: *const c_char, age: c_int) -> *mut Person;
    fn free_person(person: *mut Person);
}

fn use_c_structs() {
    let point = Point { x: 10, y: 20 };

    unsafe {
        let result = process_point(&point);
        println!("Result: {}", result);
    }
}

```

Why `#[repr(C)]`? Rust can reorder struct fields for optimization. C can't—field order is part of the ABI contract. `#[repr(C)]` locks in C's layout.

Common pitfall: Using Rust types (`String`, `&str`, `Option<T>`) in `#[repr(C)]` structs. These have Rust-specific layouts. Use raw pointers and C-compatible types instead.

Example: Creating a Safe Wrapper for C Libraries

Raw FFI is unsafe and error-prone. The pattern: create a safe Rust wrapper that encapsulates the unsafe calls and maintains invariants.

```
use std::ffi::CString;
use std::os::raw::c_char;

//=====
// Unsafe C API (these would come from a real C library)
//=====

extern "C" {
    fn create_context() -> *mut std::os::raw::c_void;
    fn destroy_context(ctx: *mut std::os::raw::c_void);
    fn context_do_work(ctx: *mut std::os::raw::c_void, data: *const c_char) -> i32;
}

/// Safe Rust wrapper for the C context API.
///
/// Ensures the context is properly created and destroyed.
pub struct Context {
    inner: *mut std::os::raw::c_void,
}

impl Context {
    /// Creates a new context.
    ///
    /// Returns None if the C library fails to create a context.
    pub fn new() -> Option<Self> {
        let ptr = unsafe { create_context() };

        if ptr.is_null() {
            None
        } else {
            Some(Context { inner: ptr })
        }
    }

    /// Performs work with the given data.
    ///
    /// Returns Ok(result) on success, Err(message) on failure.
    pub fn do_work(&mut self, data: &str) -> Result<i32, String> {
        let c_data = CString::new(data).map_err(|e| e.to_string())?;

        let result = unsafe {
            context_do_work(self.inner, c_data.as_ptr())
        };

        if result >= 0 {
            Ok(result)
        } else {
            Err(format!("Operation failed with code: {}", result))
        }
    }
}
```

```

        }

    }

impl Drop for Context {
    fn drop(&mut self) {
        unsafe {
            destroy_context(self.inner);
        }
    }
}

//=====
// If the C library is thread-safe, we can mark this safe to send between threads
//=====
// SAFETY: The C library documentation claims thread-safety
unsafe impl Send for Context {}

```

This pattern solves multiple problems: 1. **Lifetime management:** `Drop` ensures cleanup 2. **Type safety:** Users can't misuse the raw pointer 3. **Error handling:** Converts C error codes to Rust `Result` 4. **String safety:** Handles null termination automatically

When to use: Every time you wrap a C library. Users should never see `unsafe` in the public API unless absolutely necessary.

Example: Callback Functions (C to Rust)

Sometimes C libraries need to call back into your code. Callbacks must use the C calling convention and can't panic (unwinding across FFI is undefined behavior).

```

use std::os::raw::c_int;

//=====
// Type alias for C callback
//=====
type Callback = extern "C" fn(c_int) -> c_int;

extern "C" {
    fn register_callback(cb: Callback);
    fn trigger_callback(value: c_int);
}

/// Rust function with C calling convention.
///
/// This can be called from C code.
extern "C" fn my_callback(value: c_int) -> c_int {
    println!("Callback called with: {}", value);
    value * 2
}

fn callback_example() {
    unsafe {

```

```

        register_callback(my_callback);
        trigger_callback(42);
    }

}

//=====
// Advanced: Callback with user data (context pointer)
//=====

type CallbackWithData = extern "C" fn(*mut std::os::raw::c_void, c_int) -> c_int;

extern "C" fn callback_with_context(user_data: *mut std::os::raw::c_void, value: c_int) ->
    c_int {
    unsafe {
        let data = &mut *(user_data as *mut i32);
        *data += value;
        *data
    }
}

fn callback_with_state_example() {
    let mut state = 0i32;

    extern "C" {
        fn register_callback_with_data(cb: CallbackWithData, user_data: *mut std::os::raw::c_void);
        fn trigger_callback_with_data(value: c_int);
    }

    unsafe {
        register_callback_with_data(
            callback_with_context,
            &mut state as *mut i32 as *mut std::os::raw::c_void
        );
        trigger_callback_with_data(10);
    }

    println!("State after callback: {}", state);
}

```

Critical rules for callbacks: 1. **Use `extern "C"`:** Ensures C calling convention 2. **No panics:**

Unwinding through C code is UB. Use `catch_unwind` if needed 3. **Document `user_data`:** What type must be passed? Who owns it? 4. **Lifetime safety:** Ensure callback doesn't outlive data it references

User data pattern: C libraries pass a `void*` context pointer through to callbacks. This lets you maintain state without globals.

Pattern 3: Uninitialized Memory Handling

Problem: Large arrays (1MB) on stack cause overflow. Reading from I/O into buffers wastes initialization.

Solution: Use `MaybeUninit<T>` to work with possibly-uninitialized memory safely.

`MaybeUninit::uninit()` creates uninitialized, `write()` initializes, `assume_init()` asserts initialization.

Why It Matters: Prevents stack overflow: `[i32; 1_000_000]` crashes, `MaybeUninit` array succeeds.

2-3x faster for bulk initialization—no double-init.

Use Cases: Large stack arrays (>4KB), reading from files/sockets/FFI into buffers, performance-critical initialization, FFI out-parameters, deserializing from binary, reusing buffers without clearing.

Memory starts uninitialized. Creating a `Vec` doesn't fill it with zeros; allocating a buffer doesn't clear it. For performance, you often want to initialize memory piecemeal—read into it from a file, compute values on demand, or skip initialization for data you'll immediately overwrite.

The problem: Rust's safety model assumes all values are initialized. Reading uninitialized memory is instant undefined behavior. Even casting an `i32` from uninitialized memory (without using it) is UB.

`MaybeUninit<T>` solves this: it's a type that may or may not hold a valid `T`. You can work with it safely, then assert initialization when you're ready.

Example: Using `MaybeUninit` for Arrays

Large arrays on the stack can overflow if initialized naively. `MaybeUninit` lets you initialize elements one at a time without paying upfront cost.

```
use std::mem::MaybeUninit;

// Create a large array efficiently without stack overflow.
fn create_array_uninit() -> [i32; 1000] {
    let mut arr: [MaybeUninit<i32>; 1000] = unsafe {
        MaybeUninit::uninit().assume_init()
    };

    for (i, elem) in arr.iter_mut().enumerate() {
        *elem = MaybeUninit::new(i as i32);
    }

    unsafe {
        std::mem::transmute(arr)
    }
}

//=====
// Better: Use the newer stabilized API
//=====
fn create_array_uninit_safe() -> [i32; 1000] {
    let mut arr = MaybeUninit::uninit_array::<1000>();

    for (i, elem) in arr.iter_mut().enumerate() {
        elem.write(i as i32);
    }
}
```

```
unsafe { MaybeUninit::array_assume_init(arr) } }
```

Why this works: `MaybeUninit<T>` is the same size as `T`, but Rust knows it might not be initialized. Operations on it are safe until you call `assume_init()`, which asserts “I promise this is initialized.”

When to use: Large stack arrays, reading data from external sources (sockets, files), performance-critical initialization.

Example: Partial Initialization

Sometimes you need to initialize a struct field-by-field, perhaps because constructing one field depends on earlier fields, or because you’re reading from a stream.

```
use std::mem::MaybeUninit;

struct ComplexStruct {
    field1: String,
    field2: Vec<i32>,
    field3: Box<i32>,
}

fn initialize_complex_struct() -> ComplexStruct {
    let mut uninit: MaybeUninit<ComplexStruct> = MaybeUninit::uninit();
    let ptr = uninit.as_mut_ptr();

    unsafe {
        // SAFETY: Using addr_of_mut! to get field pointers without creating references
        std::ptr::addr_of_mut!((*ptr).field1).write(String::from("hello"));
        std::ptr::addr_of_mut!((*ptr).field2).write(vec![1, 2, 3]);
        std::ptr::addr_of_mut!((*ptr).field3).write(Box::new(42));

        uninit.assume_init()
    }
}
```

Critical detail: Use `addr_of_mut!` to get field pointers without creating references. Creating a `&mut T` to uninitialized memory is UB, even if you don’t read it. `addr_of_mut!` avoids this.

When to use: Deserializing from binary formats, constructing objects with complex dependencies, FFI out-parameters.

Example: Reading Uninitialized Memory (What NOT to Do)

Let’s be crystal clear: reading uninitialized memory is undefined behavior. The compiler can assume it never happens and optimize based on that assumption.

```
use std::mem::MaybeUninit;

fn undefined_behavior_example() {
    let mut uninit: MaybeUninit<i32> = MaybeUninit::uninit();
```

```

// ✅ SAFE: Writing to uninitialized memory
uninit.write(42);
let value = unsafe { uninit.assume_init() };
println!("Value: {}", value);
}

fn actual_undefined_behavior() {
    let uninit: MaybeUninit<i32> = MaybeUninit::uninit();

    // ❌ UB: Reading uninitialized memory!
    // let value = unsafe { uninit.assume_init() }; // DON'T DO THIS
}

```

What “undefined behavior” means: Not just “might crash.” The compiler can: - Delete your code entirely - Produce inconsistent results - Corrupt memory elsewhere in your program - Work fine in debug builds but break in release

Miri (Rust’s interpreter for detecting UB) will catch these issues. Use it: `cargo +nightly miri test`.

Example: Out-Parameter Pattern for C FFI

Many C functions write results through pointer arguments instead of returning them. `MaybeUninit` handles this pattern safely.

```

use std::mem::MaybeUninit;
use std::os::raw::c_int;

extern "C" {
    /// C function that writes to an out parameter.
    /// Returns 0 on success, non-zero on error.
    fn get_value(out: *mut c_int) -> c_int;
}

fn call_out_parameter_function() -> Option<i32> {
    let mut value = MaybeUninit::uninit();

    let result = unsafe {
        get_value(value.as_mut_ptr())
    };

    if result == 0 {
        Some(unsafe { value.assume_init() })
    } else {
        None
    }
}

```

Pattern: Create `MaybeUninit`, pass its pointer to C, check return code, assume init only on success.

Why this is safe: `MaybeUninit::as_mut_ptr()` gives a raw pointer that C can write to. If C doesn't write (error case), we don't call `assume_init()`, avoiding UB.

Example: Initializing Arrays from External Functions

When filling a buffer from external sources, `MaybeUninit` prevents double-initialization and enables efficient bulk operations.

```
use std::mem::MaybeUninit;

extern "C" {
    /// Fills buffer with data.
    /// Returns 0 on success, non-zero on error.
    fn fill_buffer(buffer: *mut u8, size: usize) -> i32;
}

fn read_into_buffer(size: usize) -> Option<Vec<u8>> {
    let mut buffer: Vec<MaybeUninit<u8>> = Vec::with_capacity(size);

    unsafe {
        buffer.set_len(size); // Set length without initializing
    }

    let result = unsafe {
        fill_buffer(buffer.as_mut_ptr() as *mut u8, size)
    };

    if result == 0 {
        let buffer = unsafe {
            std::mem::transmute::<Vec<MaybeUninit<u8>>, Vec<u8>>(buffer)
        };
        Some(buffer)
    } else {
        None
    }
}
```

Why transmute? `Vec<MaybeUninit<u8>>` and `Vec<u8>` have identical memory layout. `transmute` reinterprets the type without copying data.

Alternative: Use `MaybeUninit::slice_assume_init_ref()` for slices if you don't need to transfer ownership.

Pattern 4: Transmute and Type Punning

Problem: Need bit-level reinterpretation for binary protocols. Numerical code needs bit manipulation (float bits).

Solution: Use `transmute` sparingly—most dangerous function in Rust. Prefer safe alternatives: `to_bits()/from_bits()` for floats, `as` casts for integers, pointer casts for references.

Why It Matters: Wrong transmute = instant UB (lifetime extension, size mismatch, invalid values). Proper use enables zero-copy parsing (10x faster).

Use Cases: Binary protocol parsing (network, file formats), bit manipulation in numerical code, zero-copy serialization/deserialization, converting between types with identical layout, enum discrimination, hardware register access.

`std::mem::transmute` is the most dangerous function in Rust. It reinterprets bytes from one type as another, no questions asked. Get it wrong and you invoke undefined behavior. Use it only when necessary and document why it's correct.

Valid uses: - Converting between types with identical layout (e.g., `[u8; 4] ? u32`) - Implementing zero-copy protocols - Optimized numerical code

Invalid uses: - Extending lifetimes (creates dangling references) - Converting between different-sized types (compile error) - Type confusion (pointers to different types)

Example: Basic Transmute

The simplest uses: converting between types that have the same size and compatible representations.

```
use std::mem;

fn transmute_basics() {
    let a: u32 = 0x12345678;
    let b: [u8; 4] = unsafe { mem::transmute(a) };
    println!("Bytes: {:?}", b); // Depends on endianness!

    let f: f32 = 3.14;
    let bits: u32 = unsafe { mem::transmute(f) };
    println!("Float bits: 0x{:08x}", bits);

    // ✅ BETTER: Use safe built-in methods
    let bits_safe = f.to_bits();
    assert_eq!(bits, bits_safe);

    let f2 = f32::from_bits(bits);
    assert_eq!(f, f2);
}
```

Rule: If a safe alternative exists (`.to_bits()`, `.from_bits()`, `as casts`), use it. Transmute should be a last resort.

Endianness matters: `u32` to `[u8; 4]` gives different byte orders on little-endian (x86) vs big-endian (some ARM, network byte order) systems.

Example: Transmuting References (Dangerous!)

Transmuting references is one of the easiest ways to create undefined behavior. The compiler assumes references point to valid data of their type.

```

use std::mem;

//=====
// ❌ DANGEROUS: Transmuting references
//=====

fn transmute_reference_unsafe() {
    let x: &i32 = &42;
    let y: &u32 = unsafe { mem::transmute(x) };
    println!("Transmuted: {}", y);
}

//=====
// ✅ BETTER: Using pointer casting
//=====

fn transmute_reference_safer() {
    let x: i32 = 42;
    let ptr = &x as *const i32 as *const u32;
    let y = unsafe { &*ptr };
    println!("Casted: {}", y);
}

//=====
// ✅ SAFEST: Just use from_ne_bytes or as cast
//=====

fn safe_conversion() {
    let x: i32 = 42;
    let y = x as u32; // Sign-extends negative values
    println!("Converted: {}", y);
}

```

Why pointer casting is better: It's explicit about what you're doing and doesn't accidentally change const-ness or lifetimes.

When transmuting references is UB: - If the types have different alignment (e.g., `&u8` to `&u32`) - If the memory doesn't satisfy the target type's validity invariant (e.g., transmuting to `&bool` with value 2)

Example: Converting Between Slice Types

Reinterpreting slices is common in binary protocol parsing and numerical computing, but requires careful size calculations.

```

use std::slice;

fn slice_transmute() {
    let data: Vec<u32> = vec![0x12345678, 0x9abcdef0];

    let bytes: &[u8] = unsafe {
        slice::from_raw_parts(
            data.as_ptr() as *const u8,
            data.len() * std::mem::size_of::<u32>(),
        )
    }
}

```

```

};

println!("Bytes: {:?}", bytes);

// Reverse: bytes to u32 (must ensure alignment!)
}

```

Size calculation must be exact: `data.len()` elements × `size_of::<u32>()` bytes per element.

Safer alternatives: The `bytemuck` crate provides `cast_slice`, which only compiles if the transmutation is proven safe (no padding, correct alignment, etc.).

Example: Enum Discrimination

Getting an enum's discriminant (which variant it is) as a raw number.

```

use std::mem;

#[repr(u8)]
enum MyEnum {
    A = 0,
    B = 1,
    C = 2,
}

fn get_discriminant(e: &MyEnum) -> u8 {
    unsafe { *(e as *const MyEnum as *const u8) }
}

fn enum_discriminant_safe(e: &MyEnum) -> u8 {
    match e {
        MyEnum::A => 0,
        MyEnum::B => 1,
        MyEnum::C => 2,
    }
}

//=====
// ✅ Also safe: std::mem::discriminant
//=====

fn enum_discriminant_std(e: &MyEnum) -> std::mem::Discriminant<MyEnum> {
    std::mem::discriminant(e)
}

```

Why the `unsafe` version is wrong: Enums might have niches (unused bit patterns) that the compiler uses for optimization. Reading the raw bytes might give you unexpected values.

When you need the number: Use `match` or `std::mem::discriminant`. The latter returns an opaque type, useful for equality comparisons.

Example: Type Punning for Optimized Code

Type punning—reinterpreting data as a different type—is sometimes necessary for bit manipulation and numerical tricks. Unions provide a safer alternative to transmute.

```
union FloatUnion {
    f: f32,
    u: u32,
}

fn fast_float_bits(f: f32) -> u32 {
    let union = FloatUnion { f };
    unsafe { union.u } // Reading inactive union field is unsafe
}

//=====
// For real code, use the built-in method
//=====

fn correct_float_bits(f: f32) -> u32 {
    f.to_bits()
}
```

Union safety: Writing one field and reading another is safe only if both types are valid for all bit patterns (e.g., integers, floats). Reading a `bool` from a union where you wrote `u8` would be UB if the byte is not 0 or 1.

Modern Rust: Unions are less necessary now that we have methods like `to_bits()` and `from_bits()`. Use library methods when available.

Example: When NOT to Use Transmute

Some uses of transmute are always wrong. Here are the common mistakes:

```
//=====
// ✗ WRONG: Extending lifetimes
//=====

fn extend_lifetime_bad<'a>(x: &'a str) -> &'static str {
    unsafe { std::mem::transmute(x) } // UB: dangling reference
}

fn extend_lifetime_good<'a>(x: &'a str) -> &'a str {
    x // Just return the original with its real lifetime
}

//=====
// ✗ WRONG: Different sized types
//=====

fn different_sizes_bad() {
    let x: u32 = 42;
    // Won't compile: size mismatch
    // let y: u64 = unsafe { std::mem::transmute(x) };
}
```

```

}

//=====
// ✗ WRONG: Changing mutability
//=====
fn change_mutability_bad(x: &i32) -> &mut i32 {
    // UB: violates aliasing rules
    // unsafe { std::mem::transmute(x) }
    panic!("Can't safely do this")
}

//=====
// ✗ WRONG: Bypassing type safety
//=====
fn type_confusion_bad() {
    let x: &str = "hello";
    // UB: str has invariants (valid UTF-8) that might be violated
    // let y: &[u8] = unsafe { std::mem::transmute(x) };
}

```

Why these are UB: - Lifetime extension creates dangling references - Size mismatch writes to unintended memory - Mutability changes violate aliasing rules - Type confusion breaks type invariants

If the compiler accepts `transmute`, it doesn't mean it's safe! `Transmute` has a single compile-time check: sizes must match. Everything else is on you.

Pattern 5: Safe Abstractions Over Unsafe

Problem: Unsafe code scattered everywhere is error-prone. Hard to audit and maintain invariants.

Solution: Build safe types that encapsulate unsafe internals. Public API has no `unsafe`.

Why It Matters: `Vec`/`String`/`Arc`/`Mutex` prove pattern works—millions use them safely. Single audit point instead of scattered unsafe.

Use Cases: Custom collections (`Vec`, `HashMap`, `LinkedList`), synchronization primitives (`Mutex`, `RwLock`, `atomics`), custom allocators, FFI wrappers for C libraries, type-state APIs (builder patterns), intrusive data structures.

The goal of unsafe code is not to scatter `unsafe` blocks throughout your codebase. It's to build safe abstractions—types and functions that encapsulate unsafe operations and expose only safe interfaces.

This pattern is everywhere in the standard library: `Vec`, `String`, `Arc`, `Mutex` all use unsafe internally but are safe to use. You can achieve the same.

Example: Building a Safe Vec

Let's implement a simplified vector to see how unsafe internals create safe APIs. This teaches the principles of invariant maintenance and careful boundary design.

```

use std::ptr;
use std::mem;
use std::alloc::{alloc, realloc, dealloc, Layout};

pub struct MyVec<T> {
    ptr: *mut T,
    len: usize,
    cap: usize,
}

impl<T> MyVec<T> {
    /// Creates an empty vector.
    pub fn new() -> Self {
        MyVec {
            ptr: std::ptr::null_mut(), // Null is fine when cap == 0
            len: 0,
            cap: 0,
        }
    }

    /// Adds an element to the end of the vector.
    pub fn push(&mut self, value: T) {
        if self.len == self.cap {
            self.grow();
        }

        unsafe {
            ptr::write(self.ptr.add(self.len), value);
        }

        self.len += 1;
    }

    /// Removes and returns the last element, or None if empty.
    pub fn pop(&mut self) -> Option<T> {
        if self.len == 0 {
            None
        } else {
            self.len -= 1;
            unsafe {
                Some(ptr::read(self.ptr.add(self.len)))
            }
        }
    }

    /// Returns a reference to the element at the given index.
    pub fn get(&self, index: usize) -> Option<&T> {
        if index < self.len {
            unsafe {
                Some(&*self.ptr.add(index))
            }
        } else {

```

```

        None
    }
}

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

/// Grows the capacity, doubling it or setting to 1 if currently 0.
fn grow(&mut self) {
    let new_cap = if self.cap == 0 { 1 } else { self.cap * 2 };
    let new_layout = Layout::array::<T>(new_cap).unwrap();

    let new_ptr = if self.cap == 0 {
        unsafe { alloc(new_layout) as *mut T }
    } else {
        let old_layout = Layout::array::<T>(self.cap).unwrap();
        unsafe {
            realloc(
                self.ptr as *mut u8,
                old_layout,
                new_layout.size(),
            ) as *mut T
        }
    };
}

if new_ptr.is_null() {
    panic!("Allocation failed");
}

self.ptr = new_ptr;
self.cap = new_cap;
}

impl<T> Drop for MyVec<T> {
    fn drop(&mut self) {
        // Drop all elements
        while self.pop().is_some() {}

        // Deallocate memory
        if self.cap != 0 {
            let layout = Layout::array::<T>(self.cap).unwrap();
            unsafe {
                dealloc(self.ptr as *mut u8, layout);
            }
        }
    }
}

//=====
// Safety: MyVec<T> can be sent to another thread if T can
//=====
```

```
unsafe impl<T: Send> Send for MyVec<T> {}
unsafe impl<T: Sync> Sync for MyVec<T> {}
```

Invariants we maintain: 1. `ptr` is either null (when `cap == 0`) or points to valid allocated memory for `cap` elements 2. Elements `0..len` are initialized, `len..cap` are uninitialized 3. `len <= cap` always 4. Deallocation uses the same layout as allocation

Why this is safe: Public methods (`push`, `pop`, `get`) never break invariants. Users can't create invalid states.

Send/Sync: We implement these unsafe traits because `MyVec<T>` upholds the same safety guarantees as `Vec<T>`.

Example: Invariants and Documentation

When writing unsafe code, document your invariants clearly. Future maintainers (including yourself) need to know what assumptions the code relies on.

```
/// A slice type that is guaranteed to be non-empty.
///
/// # Safety Invariants
/// - The inner slice must always have at least one element
/// - The pointer must be valid and properly aligned
/// - The data must be valid for lifetime 'a
pub struct NonEmptySlice<'a, T> {
    slice: &'a [T],
}

impl<'a, T> NonEmptySlice<'a, T> {
    /// Creates a non-empty slice from a regular slice.
    ///
    /// Returns None if the slice is empty.
    pub fn new(slice: &'a [T]) -> Option<Self> {
        if slice.is_empty() {
            None
        } else {
            Some(NonEmptySlice { slice })
        }
    }

    /// Creates a non-empty slice without checking.
    ///
    /// # Safety
    /// The caller must ensure that the slice is not empty.
    pub unsafe fn new_unchecked(slice: &'a [T]) -> Self {
        debug_assert!(!slice.is_empty());
        NonEmptySlice { slice }
    }

    /// Returns the first element (always exists).
    pub fn first(&self) -> &T {
        &self.slice[0]
```

```

}

/// Returns the last element (always exists).
pub fn last(&self) -> &T {
    &self.slice[self.slice.len() - 1]
}

pub fn as_slice(&self) -> &[T] {
    self.slice
}
}

```

Pattern: Every `unsafe fn` needs a “# Safety” section. Every `unsafe` block should have a “SAFETY:” comment explaining why it’s correct.

Invariants: Document what must always be true. These are the assumptions your unsafe code relies on.

Example: PhantomData for Type Safety

`PhantomData` is a zero-sized type that tells the compiler “I logically own a `T`” without actually storing one. This affects drop checking and variance.

```

use std::marker::PhantomData;
use std::ptr::NonNull;

/// A raw pointer wrapper that tracks ownership.
pub struct RawPtr<T> {
    ptr: NonNull<T>,
    // PhantomData tells compiler we "own" a T
    _marker: PhantomData<T>,
}

impl<T> RawPtr<T> {
    pub fn new(value: T) -> Self {
        let boxed = Box::new(value);
        RawPtr {
            ptr: NonNull::new(Box::into_raw(boxed)).unwrap(),
            _marker: PhantomData,
        }
    }

    pub fn as_ref(&self) -> &T {
        unsafe { self.ptr.as_ref() }
    }

    pub fn as_mut(&mut self) -> &mut T {
        unsafe { self.ptr.as_mut() }
    }
}

impl<T> Drop for RawPtr<T> {

```

```

fn drop(&mut self) {
    unsafe {
        drop(Box::from_raw(self.ptr.as_ptr()));
    }
}

//=====
// Safe because we own the T
//=====

unsafe impl<T: Send> Send for RawPtr<T> {}
unsafe impl<T: Sync> Sync for RawPtr<T> {}

```

Why PhantomData? Without it, the compiler wouldn't know we "own" a `T`, so drop check wouldn't run `T`'s destructor before `RawPtr`'s. The `_marker` field has zero runtime cost but provides compile-time guarantees.

Variance: `PhantomData<T>` makes `RawPtr<T>` covariant over `T`, meaning `RawPtr<&'long T>` can be used where `RawPtr<&'short T>` is expected.

Example: Building Safe APIs with Unsafe Internals

Here's a complete example: a spinlock that uses unsafe internally but exposes a safe API through RAII guards.

```

use std::cell::UnsafeCell;
use std::sync::atomic::{AtomicBool, Ordering};

/// A simple spinlock implementation.
///
/// Uses atomic operations and unsafe cell internally,
/// but provides safe locking through RAII guards.
pub struct SpinLock<T> {
    locked: AtomicBool,
    data: UnsafeCell<T>,
}

pub struct SpinLockGuard<'a, T> {
    lock: &'a SpinLock<T>,
}

impl<T> SpinLock<T> {
    pub fn new(data: T) -> Self {
        SpinLock {
            locked: AtomicBool::new(false),
            data: UnsafeCell::new(data),
        }
    }

    /// Acquires the lock, blocking until available.
    ///

```

```

/// Returns a guard that provides access to the data
/// and releases the lock when dropped.
pub fn lock(&self) -> SpinLockGuard<T> {
    while self.locked.swap(true, Ordering::Acquire) {
        std::hint::spin_loop();
    }

    SpinLockGuard { lock: self }
}

impl<'a, T> std::ops::Deref for SpinLockGuard<'a, T> {
    type Target = T;

    fn deref(&self) -> &T {
        unsafe { &*self.lock.data.get() }
    }
}

impl<'a, T> std::ops::DerefMut for SpinLockGuard<'a, T> {
    fn deref_mut(&mut self) -> &mut T {
        unsafe { &mut *self.lock.data.get() }
    }
}

impl<'a, T> Drop for SpinLockGuard<'a, T> {
    fn drop(&mut self) {
        self.lock.locked.store(false, Ordering::Release);
    }
}

//=====
// SAFETY: SpinLock properly synchronizes access to T
//=====

// The Acquire/Release ordering ensures memory visibility
unsafe impl<T: Send> Send for SpinLock<T> {}
unsafe impl<T: Send> Sync for SpinLock<T> {}

```

Safe API: Users never see `unsafe`. The lock/unlock mechanism is enforced by the type system—you can't forget to unlock because the guard drops automatically.

Unsafe implementation: `UnsafeCell` allows interior mutability, atomics provide synchronization, but these are encapsulated.

Ordering matters: `Acquire` on lock, `Release` on unlock. This ensures memory writes before unlock are visible after lock on other threads.

Example: Compile-Time Type-State Programming

Type-state uses phantom types to encode state machine transitions at compile-time, preventing invalid state transitions.

```
use std::marker::PhantomData;

//=====
// Type states
//=====

struct Locked;
struct Unlocked;

/// A lock that uses the type system to ensure correct usage.
///
/// Can only access data when in the Locked state.
pub struct TypeStateLock<T, State = Unlocked> {
    data: *mut T,
    _state: PhantomData<State>,
}

impl<T> TypeStateLock<T, Unlocked> {
    pub fn new(data: T) -> Self {
        TypeStateLock {
            data: Box::into_raw(Box::new(data)),
            _state: PhantomData,
        }
    }

    /// Locks the data, transitioning to Locked state.
    pub fn lock(self) -> TypeStateLock<T, Locked> {
        TypeStateLock {
            data: self.data,
            _state: PhantomData,
        }
    }
}

impl<T> TypeStateLock<T, Locked> {
    /// Unlocks the data, transitioning back to Unlocked state.
    pub fn unlock(self) -> TypeStateLock<T, Unlocked> {
        TypeStateLock {
            data: self.data,
            _state: PhantomData,
        }
    }

    /// Accesses the data (only available when locked).
    pub fn access(&mut self) -> &mut T {
        unsafe { &mut *self.data }
    }
}

impl<T, State> Drop for TypeStateLock<T, State> {
    fn drop(&mut self) {
        unsafe {
            let _ = Box::from_raw(self.data);
        }
    }
}
```

```

        }
    }

fn typestate_example() {
    let lock = TypeStateLock::new(vec![1, 2, 3]);

    // Can't access unlocked data - no access() method!

    let mut locked = lock.lock();
    locked.access().push(4); // OK, we're in Locked state

    let unlocked = locked.unlock();
    // Can't access again - back to Unlocked state
}

```

Power of type-state: Impossible states are unrepresentable. You can't access unlocked data because there's no `access()` method in that state.

Real-world use: Builder APIs that require certain methods be called (typestate ensures required fields are set), protocol state machines, resource lifecycle management.

Example: Testing Unsafe Code

Unsafe code requires rigorous testing, including tests specifically designed to trigger potential UB.

```

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

    #[test]
    fn test_my_vec_basic() {
        let mut vec = MyVec::new();
        vec.push(1);
        vec.push(2);
        vec.push(3);

        assert_eq!(vec.get(0), Some(&1));
        assert_eq!(vec.get(1), Some(&2));
        assert_eq!(vec.get(2), Some(&3));
        assert_eq!(vec.get(3), None);
    }

    #[test]
    fn test_my_vec_pop() {
        let mut vec = MyVec::new();
        vec.push(1);
        vec.push(2);

        assert_eq!(vec.pop(), Some(2));
        assert_eq!(vec.pop(), Some(1));
        assert_eq!(vec.pop(), None);
    }
}

```

```

}

#[test]
fn test_my_vec_drop() {
    use std::sync::atomic::{AtomicUsize, Ordering};

    static DROP_COUNT: AtomicUsize = AtomicUsize::new(0);

    struct DropCounter;
    impl Drop for DropCounter {
        fn drop(&mut self) {
            DROP_COUNT.fetch_add(1, Ordering::SeqCst);
        }
    }

    {
        let mut vec = MyVec::new();
        vec.push(DropCounter);
        vec.push(DropCounter);
        vec.push(DropCounter);
    } // vec dropped here

    assert_eq!(DROP_COUNT.load(Ordering::SeqCst), 3);
}

#[test]
fn test_thread_safety() {
    use std::sync::Arc;
    use std::thread;

    let vec = Arc::new(SpinLock::new(MyVec::new()));
    let mut handles = vec![];

    for i in 0..10 {
        let vec_clone = Arc::clone(&vec);
        handles.push(thread::spawn(move || {
            vec_clone.lock().push(i);
        }));
    }

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

    assert_eq!(vec.lock().len(), 10);
}
}

```

Testing strategies: 1. **Basic correctness:** Does it work for normal cases? 2. **Edge cases:** Empty collections, single elements, capacity boundaries 3. **Drop tracking:** Use `DropCounter` to ensure no

Leaks or double-drops 4. **Thread safety:** Test concurrent access with multiple threads 5. **Miri:** Run tests with Miri to detect UB

Miri is essential: It interprets your code at the MIR level and detects undefined behavior that compiles fine but is wrong.

Pattern 5: Best Practices for Unsafe Code

Unsafe Rust is powerful but dangerous. These practices help you wield that power responsibly.

1. Minimize Unsafe Boundaries

Keep unsafe code localized. Encapsulate it in small, well-tested functions or types.

```
//=====
// ❌ BAD: Unsafe spreads throughout the code
//=====
pub fn bad_api(data: *mut u8, len: usize) {
    // Users must pass raw pointers
}

//=====
// ✅ GOOD: Unsafe is contained internally
//=====
pub fn good_api(data: &mut [u8]) {
    unsafe {
        // Unsafe code is hidden from users
    }
}
```

Principle: Unsafe should be an implementation detail, not a user-facing requirement.

2. Document Safety Requirements

Every unsafe function must document its preconditions. This is for your future self and other maintainers.

```
/// Interprets a raw pointer as a slice.
///
/// # Safety
///
/// The caller must ensure that:
/// - `ptr` is valid for reads of `len * size_of::<T>()` bytes
/// - `ptr` is properly aligned for type `T`
/// - The memory referenced by `ptr` is not concurrently accessed for writes
/// - `len` is exactly the number of elements in the allocation
/// - The memory contains valid values of type `T`
pub unsafe fn from_raw_parts<T>(ptr: *const T, len: usize) -> &'static [T] {
```

```
    std::slice::from_raw_parts(ptr, len)
}
```

What to document: - Pointer validity (aligned, non-null, within bounds) - Initialization state (is memory initialized?) - Concurrent access (can other threads access this?) - Ownership (who frees this memory?)

3. Use Helper Functions

Extract unsafe operations into well-named, well-tested helper functions.

```
mod unsafe_helpers {
    /// Writes a value to a pointer without dropping the old value.
    ///
    /// # Safety
    /// `ptr` must be valid for writes and properly aligned.
    pub(crate) unsafe fn write_unchecked<T>(ptr: *mut T, value: T) {
        debug_assert!(!ptr.is_null(), "write_unchecked: null pointer");
        std::ptr::write(ptr, value);
    }

    /// Reads a value from a pointer without moving it.
    ///
    /// # Safety
    /// `ptr` must be valid for reads, properly aligned, and point to initialized data.
    pub(crate) unsafe fn read_unchecked<T>(ptr: *const T) -> T {
        debug_assert!(!ptr.is_null(), "read_unchecked: null pointer");
        std::ptr::read(ptr)
    }
}
```

Benefits: - Centralize unsafe operations for easier auditing - Add debug assertions for development builds - Document assumptions once, reference from call sites

4. Use Clippy and Miri

Automated tools catch mistakes humans miss.

```
# Check for undocumented unsafe blocks
cargo clippy -- -W clippy::undocumented_unsafe_blocks

# Detect undefined behavior at runtime
cargo +nightly miri test

# Run with address sanitizer (detects memory errors)
RUSTFLAGS="-Z sanitizer=address" cargo +nightly test

# Run with thread sanitizer (detects data races)
RUSTFLAGS="-Z sanitizer=thread" cargo +nightly test
```

Miri interprets code and detects: - Use of uninitialized memory - Use-after-free - Invalid pointer arithmetic - Data races (in unsafe code) - Violating pointer aliasing rules

Clippy warns about: - Undocumented unsafe blocks - Transmutes that could be replaced with safe alternatives - Missing safety comments

5. Consider Alternatives

Before writing unsafe code, check if a safe solution exists.

```
//=====
// Instead of raw pointers, consider:
//=====

// - std::pin::Pin for self-referential structs
// - std::rc::Rc or std::sync::Arc for shared ownership
// - std::cell::UnsafeCell for interior mutability (still unsafe, but more targeted)
// - std::sync::atomic for lock-free operations
// - ouroboros crate for self-referential structs
// - bytemuck crate for safe transmutes (compile-time checks)
// - zerocopy crate for zero-copy parsing with safety

//=====
// Example: Safe transmute with bytemuck
//=====

use bytemuck::{Pod, Zeroable};

#[derive(Copy, Clone, Pod, Zeroable)]
#[repr(C)]
struct Point {
    x: f32,
    y: f32,
}

fn safe_transmute_example() {
    let bytes: [u8; 8] = [0; 8];
    let point: Point = bytemuck::cast(bytes); // Compile-time verified safe
}
```

Crates that help: - [bytemuck](#): Safe transmutations with compile-time verification - [zerocopy](#): Zero-copy parsing with safety proofs - [ouroboros](#): Self-referential structs without unsafe (macros generate safe wrappers) - [parking_lot](#): Better locks with less overhead than [std::sync](#)

Summary

Unsafe Rust is not about being reckless—it's about taking responsibility for safety properties the compiler cannot verify.

Key principles: 1. **Minimize scope:** Keep unsafe blocks small and localized 2. **Build safe abstractions:** Encapsulate unsafe code behind safe APIs 3. **Document invariants:** Explain what must

be true for correctness 4. **Test rigorously**: Use Miri, sanitizers, and stress tests 5. **PREFER ALTERNATIVES**: Use safe solutions when they exist

When to use unsafe: - Implementing fundamental abstractions (collections, concurrency primitives) - FFI to interact with C libraries - Performance optimizations where safe code can't match (after profiling!) - Hardware-level programming (embedded systems, drivers)

When to avoid unsafe: - To bypass the borrow checker in application code (redesign instead) - For micro-optimizations without measurements - When safe abstractions exist (use them!) - If you can't explain why it's safe (it probably isn't)

The standard library is proof this works: millions of lines of safe Rust code rely on carefully crafted unsafe foundations. Your unsafe code can achieve the same reliability with discipline and care.

Synchronous I/O

This chapter covers Rust's synchronous I/O patterns—blocking operations that pause threads until complete. Unlike async I/O (the next chapter), synchronous I/O is simpler, easier to debug, and sufficient for most programs. CLI tools, build scripts, and even high-performance servers with thread pools rely on these patterns.

Pattern 1: Basic File Operations

Problem: Files are the main way programs persist data—configuration, logs, cached results, user documents—but handling them correctly can be tricky. You need to decide how to read the file efficiently, handle different formats, and deal with errors safely.

Solution: Do you read the entire file into memory (`fs::read_to_string`) for simplicity, or process it line by line (`BufReader::lines`) to handle files larger than RAM? The right choice depends on your constraints.

Why It Matters: Choosing the wrong approach can cause: Memory exhaustion if a large file is loaded at once. Slow performance if line-by-line reading is overused for small files.

Use Cases: Reading JSON file, Processing logs or CSV files , CLI tools or servers that need predictable file I/O behavior.

Example: Basic file reading

The simplest case: read a small file entirely into memory.

```
use std::fs::File;
use std::io::{self, Read};
```

Example: Read entire file to string (UTF-8)

This example walks through how to read entire file to string (utf-8).

```
fn read_to_string(path: &str) -> io::Result<String> {
    std::fs::read_to_string(path)
    // Allocates a String big enough for the entire file
    // Returns Err if file doesn't exist, isn't readable, or isn't valid UTF-8
}
```

Example: Read entire file to bytes (binary)

This example walks through how to read entire file to bytes (binary).

```
fn read_to_bytes(path: &str) -> io::Result<Vec<u8>> {
    std::fs::read(path)
    // Allocates a Vec<u8> and reads all bytes
    // Returns Err if file doesn't exist or isn't readable
}
```

Example: Manual reading with buffer control

This example walks through manual reading with buffer control.

```
fn read_with_buffer(path: &str) -> io::Result<String> {
    let mut file = File::open(path)?;
    let mut contents = String::new();

    // read_to_string reads until EOF, automatically resizing the String
    file.read_to_string(&mut contents)?;
    Ok(contents)
}
```

Example: Read exact number of bytes

This example walks through how to read exact number of bytes.

```
fn read_exact_bytes(path: &str, n: usize) -> io::Result<Vec<u8>> {
    let mut file = File::open(path)?;
    let mut buffer = vec![0; n];

    // read_exact returns Err if fewer than n bytes are available
    // This guarantees you get all n bytes or an error--no partial reads
    file.read_exact(&mut buffer)?;
    Ok(buffer)
}
```

When each pattern fits:

- `fs::read_to_string()`: Config files < 10 MB, HTML templates, small data files
- `fs::read()`: Binary files you'll process in-memory (images, compressed data)
- Manual `File::open() + read_to_string()`: Same as above but when you need the `File` handle for metadata
- `read_exact()`: Binary protocols with fixed-size headers, database page reads

The UTF-8 guarantee: `read_to_string` validates UTF-8. If your file contains invalid Unicode, it returns `Err`. Use `read()` for binary data or `String::from_utf8_lossy()` if you want replacement characters instead of errors.

Example: Basic File Writing

Writing is simpler than reading because you control the data format. The main decision: overwrite, append, or create new?

```
use std::fs::File;
use std::io::{self, Write};
```

Example: Write string to file (overwrites existing)

This example walks through how to write string to file (overwrites existing).

```
fn write_string(path: &str, content: &str) -> io::Result<()> {
    std::fs::write(path, content)
    // Creates file if it doesn't exist
    // Truncates (erases) existing content
    // Writes all content in one operation
}
```

Example: Write bytes to file

This example walks through how to write bytes to file.

```
fn write_bytes(path: &str, content: &[u8]) -> io::Result<()> {
    std::fs::write(path, content)
}
```

Example: Manual writing with file handle

This example walks through manual writing with file handle.

```
fn write_with_handle(path: &str, content: &str) -> io::Result<()> {
    let mut file = File::create(path)?;

    // write_all ensures all bytes are written or returns Err
    // Partial writes are retried automatically
    file.write_all(content.as_bytes())?;
    Ok(())
}
```

Example: Append to file (preserves existing content)

This example walks through how to append to file (preserves existing content).

```

fn append_to_file(path: &str, content: &str) -> io::Result<()> {
    use std::fs::OpenOptions;

    let mut file = OpenOptions::new()
        .append(true)      // Open in append mode
        .create(true)      // Create if doesn't exist
        .open(path)?;

    writeln!(file, "{}", content)?; // Adds newline automatically
    Ok(())
}

```

Critical distinction: `File::create()` truncates (erases) existing files. If you mean to append, use `OpenOptions`. Many bugs come from accidentally truncating when you meant to append.

Automatic flushing: When a `File` is dropped, Rust automatically flushes buffered writes and closes the file. You usually don't need explicit `flush()` unless you're coordinating with external processes that need immediate visibility.

Example: File Opening Options

`OpenOptions` gives fine-grained control over how files are opened—essential for implementing robust file-based protocols or handling concurrent access.

```

use std::fs::OpenOptions;
use std::io;

fn advanced_file_opening() -> io::Result<()> {
    // Read-only mode (default for File::open)
    // Fails if file doesn't exist
    let file = OpenOptions::new()
        .read(true)
        .open("data.txt")?;

    // Write-only mode, create if doesn't exist
    // This is what File::create() does internally
    let file = OpenOptions::new()
        .write(true)
        .create(true)      // Create if missing
        .open("output.txt")?;

    // Append mode (write to end, never truncate)
    // Essential for log files
    let file = OpenOptions::new()
        .append(true)
        .open("log.txt")?;

    // Truncate existing file to zero length
    // Dangerous: erases all existing content!
    let file = OpenOptions::new()
        .write(true)

```

```

    .truncate(true)
    .open("temp.txt")?;

    // Create new file, fail if it already exists
    // Use this to avoid overwriting important files
    let file = OpenOptions::new()
        .write(true)
        .create_new(true)    // Fail if exists (atomic check-and-create)
        .open("unique.txt")?;

    // Read and write mode (for in-place modification)
    // Allows seeking and both reading and writing
    let file = OpenOptions::new()
        .read(true)
        .write(true)
        .open("data.bin")?;

    // Custom permissions (Unix only)
    // Set file mode bits (rwxrwxrwx)
#[cfg(unix)]
{
    use std::os::unix::fs::OpenOptionsExt;
    let file = OpenOptions::new()
        .write(true)
        .create(true)
        .mode(0o644)          // rw-r--r-- (owner can write, others can read)
        .open("secure.txt")?;

}

Ok(())
}

```

Common patterns: - **Append-only log:** `.append(true).create(true)` — Never loses data, safe for multiple writers - **Exclusive creation:** `.write(true).create_new(true)` — Atomic: either you create it or fail - **Read-modify-write:** `.read(true).write(true)` — Allows seeking to update parts of file

Concurrency gotcha: Opening a file for writing doesn't lock it on most platforms. Multiple processes can open the same file simultaneously and corrupt it. Use file locking (platform-specific) or atomic file replacement (write to temp, then rename) for safe concurrent access.

Pattern 2: Buffered Reading and Writing

Problem: Reading or writing files byte-by-byte makes a system call per byte—catastrophically slow with $O(N)$ syscalls for N bytes. Processing a 100 MB file unbuffered can take minutes.

Solution: Wrap File handles in BufReader/BufWriter which maintain internal buffers (default 8 KB). BufReader amortizes reads: fills buffer with one syscall, serves subsequent reads from memory.

Why It Matters: Buffering provides 1000x speedup—unbuffered 100 MB file takes minutes, buffered takes milliseconds. Syscall overhead dominates unbuffered I/O.

Use Cases: Log file parsing (line-by-line), CSV processing (buffered reading), config file loading, generating reports (buffered writing), any text-oriented file I/O, binary protocol parsing with custom delimiters.

Example: Buffered Line-by-Line Reading

Process large log files or text data that doesn't fit in memory. Need memory-efficient streaming. Want to skip comments or filter lines.

```
use std::fs::File;
use std::io::{self, BufRead, BufReader};
```

Example: Process large files line by line (memory-efficient)

This example walks through process large files line by line (memory-efficient).

```
fn process_large_file(path: &str) -> io::Result<()> {
    let file = File::open(path)?;
    let reader = BufReader::new(file); // 8KB buffer by default

    for (index, line) in reader.lines().enumerate() {
        let line = line?; // Handle I/O errors per line

        // Process each line
        if line.starts_with('#') {
            continue; // Skip comments
        }

        println!("Line {}: {}", index + 1, line);
    }

    Ok(())
}
```

Example: Filter lines (e.g., only errors)

This example walks through how to filter lines (e.g., only errors).

```
fn process_errors_only(path: &str) -> io::Result<Vec<String>> {
    let file = File::open(path)?;
    let reader = BufReader::new(file);

    reader.lines()
        .filter_map(|line| line.ok()) // Skip I/O errors
        .filter(|line| line.contains("ERROR"))
        .collect()
}
```

Key Benefits: - Memory usage: O(1) per line, not O(file size) - Scales to files larger than RAM - Easy filtering and transformation - 1000x faster than byte-by-byte reads

Example: Buffered Writing for Performance

Writing many small chunks (like log entries or CSV rows) makes a syscall per write. Generating large output files with unbuffered writes is slow.

```
use std::fs::File;
use std::io::{self, BufWriter, Write};
```

Example: Buffered writing (essential for performance)

This example walks through buffered writing (essential for performance).

```
fn buffered_write(path: &str, lines: &[&str]) -> io::Result<()> {
    let file = File::create(path)?;
    let mut writer = BufWriter::new(file); // 8 KB buffer by default

    for line in lines {
        writeln!(writer, "{}", line)?; // Writes to buffer
    }

    writer.flush()?;
    Ok(())
}
```

Example: Append to log file (preserves existing)

This example walks through how to append to log file (preserves existing).

```
fn append_log(path: &str, message: &str) -> io::Result<()> {
    use std::fs::OpenOptions;

    let file = OpenOptions::new()
        .append(true)
        .create(true)
        .open(path)?;

    let mut writer = BufWriter::new(file);
    writeln!(writer, "{}", message)?;
    writer.flush()?;
    Ok(())
}
```

Key Benefits: - Batches many writes into few syscalls - 50-100x faster for bulk writes - Automatic flush on drop (but explicit flush safer) - Essential for generating large output files

Pattern 3: Standard Streams

Problem: Programs need terminal I/O for user interaction. Shell pipelines break if diagnostics go to stdout instead of stderr.

Solution: Use `io::stdin()` for input, `io::stdout()` for data output, `io::stderr()` for errors/diagnostics. Call `flush()` after `print!()` before reading input.

Why It Matters: Correct stream separation enables Unix pipelines (`program | grep`). Flushing prevents UX bugs where prompts appear after input.

Use Cases: CLI tools (interactive prompts, menus), Unix filters (`cat|grep|wc`), progress indicators (stderr while stdout pipes data), logging, command-line argument parsing.

Example: Interactive Prompts with `stdin`

Read user input with prompts. Without flushing, prompts don't appear before input.

```
use std::io::{self, Write};
```

Example: Read with prompt

This example walks through how to read with prompt.

```
fn prompt(message: &str) -> io::Result<String> {
    print!("{} ", message);
    io::stdout().flush()?;
    // CRITICAL: flush before reading

    let mut input = String::new();
    io::stdin().read_line(&mut input)?;
    Ok(input.trim().to_string())
}
```

Example: Interactive menu

This example walks through interactive menu.

```
fn interactive_menu() -> io::Result<()> {
    loop {
        println!("\n==== Menu ====");
        println!("1. Process data");
        println!("2. View stats");
        println!("3. Exit");

        let choice = prompt("Enter choice: ")?;

        match choice.as_str() {
            "1" => println!("Processing..."),
            "2" => println!("Stats: ..."),
            "3" => break,
        }
    }
}
```

```

        _ => eprintln!("Invalid choice"), // Error to stderr!
    }
}
Ok(())
}

```

Key Benefits: - Flush before reading prevents prompt bugs - Use stdin.lock() for efficient multi-line reads - EOF (Ctrl+D/Ctrl+Z) handled gracefully - Errors go to stderr, output to stdout

Pattern 4: Memory-Mapped I/O

Problem: Random access with read() + seek() is O(N) per access. Large files don't fit in RAM.

Solution: Use memmap2 crate to treat files as byte slices. OS handles paging data in/out.

Why It Matters: Random access becomes memory-speed (hot pages). Databases need O(1) page access, not O(N) seek+read.

Use Cases: Databases (page-based storage), binary search in large files, memory-mapped data structures, shared memory IPC, large read-only assets, sparse file access.

Basic Memory Mapping

Rust doesn't include mmap in the standard library (it's unsafe and platform-specific). The [memmap2](#) crate provides a safe abstraction.

```

// Note: Add to Cargo.toml:
// [dependencies]
// memmap2 = "0.9"

```

Example: This is a conceptual example showing the API

This example walks through this is a conceptual example showing the api.

```

#[cfg(feature = "memmap_example")]
mod memmap_examples {
    use memmap2::{Mmap, MmapMut, MmapOptions};
    use std::fs::File;
    use std::io::{self, Write};

    // Read-only memory map
    // Use this for: Large read-only data files, databases
    fn mmap_read(path: &str) -> io::Result<()> {
        let file = File::open(path)?;
        let mmap = unsafe { Mmap::map(&file)? };

        // Access memory like a byte slice
        let data: &[u8] = &mmap[..];
        println!("First 10 bytes: {:?}", &data[..10.min(data.len())]);
    }
}

```

```

    // mmap is unmapped when dropped
    Ok(())
}

// Mutable memory map
// Use this for: In-place file modification, persistent data structures
fn mmap_write(path: &str) -> io::Result<()> {
    let file = File::options()
        .read(true)
        .write(true)
        .create(true)
        .open(path)?;

    // Set file size before mapping
    file.set_len(1024)?;

    let mut mmap = unsafe { MmapMut::map_mut(&file)? };

    // Write data directly to memory (actually the file)
    mmap[0..5].copy_from_slice(b"Hello");

    // Flush to ensure writes reach disk
    mmap.flush()?;
}

Ok(())
}

// Anonymous memory map (not backed by file)
// Use this for: Shared memory IPC, large temporary buffers
fn mmap_anonymous() -> io::Result<()> {
    let mut mmap = MmapMut::map_anon(1024)?;

    mmap[0..5].copy_from_slice(b"Hello");

    println!("Data: {:?}", &mmap[0..5]);

    Ok(())
}

// Large file processing with mmap
// Use this when: File is too large for RAM but you need random access
fn process_large_file_mmap(path: &str) -> io::Result<u64> {
    let file = File::open(path)?;
    let mmap = unsafe { Mmap::map(&file)? };

    // Count newlines efficiently (CPU-bound, not I/O-bound)
    let count = mmap.iter().filter(|&&b| b == b'\n').count();

    Ok(count)
}
}

```

Why unsafe: The OS can change mapped memory at any time (e.g., if another process modifies the file). Rust can't guarantee your references remain valid. The [memmap2](#) crate encapsulates this unsafety.

Performance characteristics: - **Cold access:** First access to a page causes page fault (OS loads page from disk). Slower than buffered read. - **Hot access:** Subsequent access to same page is pure memory speed. Faster than buffered read. - **Random access:** Mmap excels. Buffered I/O requires seeks.

Gotcha: Mmap doesn't necessarily improve performance. For sequential reads, [BufReader](#) is simpler and often faster. Measure first.

Pattern 5: Directory Traversal

Problem: Need to walk file trees to find files, calculate sizes, or batch process. Simple recursion hits symlink loops.

Solution: Use `fs::read_dir()` for single-level listing. Implement recursive walk with visited path tracking (or use `walkdir` crate).

Why It Matters: Build systems scan thousands of files to find sources. Backup tools walk entire disks.

Use Cases: Build systems (find .rs files), file search tools (find by name/pattern), disk usage analyzers, backup tools, batch file operations (chmod/chown recursively).

Example: Basic Directory Operations

```
use std::fs;
use std::io;
use std::path::Path;
```

Example: Create directory

This example walks through how to create directory.

```
fn create_directory(path: &str) -> io::Result<()> {
    fs::create_dir(path)
    // Fails if parent doesn't exist
    // Fails if directory already exists
}
```

Example: Create directory and all parent directories

This example walks through how to create directory and all parent directories.

```
// Like mkdir -p in Unix
fn create_directory_all(path: &str) -> io::Result<()> {
    fs::create_dir_all(path)
    // Creates parent directories as needed
```

```
// Succeeds if directory already exists
}
```

Example: Remove empty directory

This example walks through remove empty directory.

```
fn remove_directory(path: &str) -> io::Result<()> {
    fs::remove_dir(path)
    // Fails if directory is not empty
}
```

Example: Remove directory and all contents (dangerous!)

This example walks through remove directory and all contents (dangerous!).

```
fn remove_directory_all(path: &str) -> io::Result<()> {
    fs::remove_dir_all(path)
    // Recursively deletes everything
    // Like rm -rf in Unix
}
```

Example: Check if path exists

This example walks through check if path exists.

```
fn path_exists(path: &str) -> bool {
    Path::new(path).exists()
    // Returns false for broken symlinks
}
```

Example: Check if path is directory

This example walks through check if path is directory.

```
fn is_directory(path: &str) -> bool {
    Path::new(path).is_dir()
    // Follows symlinks
}
```

Safety note: `remove_dir_all` is dangerous. It's equivalent to `rm -rf`. There's no trash bin, no undo. Many programs ask for confirmation before using this.

Example: Reading Directory Contents

Listing a directory is simple, but the iterator-based API requires careful error handling.

```
use std::fs;
use std::io;
use std::path::PathBuf;
```

Example: List directory contents

This example walks through list directory contents.

```
fn list_directory(path: &str) -> io::Result<Vec<PathBuf>> {
    let mut entries = Vec::new();

    for entry in fs::read_dir(path)? {
        let entry = entry?; // Each entry can fail
        entries.push(entry.path());
    }

    Ok(entries)
}
```

Example: List only files (skip directories)

This example walks through list only files (skip directories).

```
fn list_files_only(path: &str) -> io::Result<Vec<PathBuf>> {
    let mut files = Vec::new();

    for entry in fs::read_dir(path)? {
        let entry = entry?;
        if entry.file_type()?.is_file() {
            files.push(entry.path());
        }
    }

    Ok(files)
}
```

Example: List files with specific extension

This example walks through list files with specific extension.

```
// Use this for: Finding all .rs files, .txt files, etc.
fn list_by_extension(path: &str, ext: &str) -> io::Result<Vec<PathBuf>> {
    let mut files = Vec::new();

    for entry in fs::read_dir(path)? {
        let entry = entry?;
        let path = entry.path();

        if path.extension().and_then(|s| s.to_str()) == Some(ext) {
```

```
        files.push(path);
    }

Ok(files)
}
```

Example: Get directory entries with metadata

This example walks through get directory entries with metadata.

```
// Use this for: Sorting by size, filtering by date, etc.
fn list_with_metadata(path: &str) -> io::Result<Vec<(PathBuf, fs::Metadata)>> {
    let mut entries = Vec::new();

    for entry in fs::read_dir(path)? {
        let entry = entry?;
        let metadata = entry.metadata()?;
        entries.push((entry.path(), metadata));
    }

    Ok(entries)
}
```

Error handling: `read_dir()` can fail (directory doesn't exist, no permission). Each call to `entry?` can also fail (permission denied on individual files). Handle both.

Example: Recursive Directory Traversal

Walking entire directory trees is common but requires careful handling of errors and cycles (symlink loops).

```
use std::fs;
use std::io;
use std::path::{Path, PathBuf};
```

Example: Recursive file listing

This example walks through recursive file listing.

```
// Classic depth-first search pattern
fn walk_directory(path: &Path, files: &mut Vec<PathBuf>) -> io::Result<()> {
    if path.is_dir() {
        for entry in fs::read_dir(path)? {
            let entry = entry?;
            let path = entry.path();

            if path.is_dir() {
                walk_directory(&path, files)?; // Recurse
            } else {

```

```

        files.push(path);
    }
}
Ok(())
}
```

Example: Get all files recursively

This example walks through get all files recursively.

```

fn get_all_files(path: &str) -> io::Result<Vec<PathBuf>> {
    let mut files = Vec::new();
    walk_directory(Path::new(path), &mut files)?;
    Ok(files)
}
```

Example: Recursive directory tree printer (visual tree)

This example walks through recursive directory tree printer (visual tree).

```

// Produces output like the `tree` command
fn print_tree(path: &Path, prefix: &str) -> io::Result<()> {
    let entries = fs::read_dir(path)?;
    let mut entries: Vec<_> = entries.collect::<Result<_, _>>()?;
    entries.sort_by_key(|e| e.path());

    for (i, entry) in entries.iter().enumerate() {
        let is_last = i == entries.len() - 1;
        let connector = if is_last { "└ " } else { "├ " };
        let extension = if is_last { "    " } else { "│   " };

        println!("{}{}{}", prefix, connector, entry.file_name().to_string_lossy());

        if entry.file_type()?.is_dir() {
            let new_prefix = format!("{}{}", prefix, extension);
            print_tree(&entry.path(), &new_prefix)?;
        }
    }
}

Ok(())
}
```

Example: Find files matching pattern (like find command)

This example walks through find files matching pattern (like find command).

```

fn find_files(root: &Path, pattern: &str) -> io::Result<Vec<PathBuf>> {
    let mut matches = Vec::new();
```

```

fn search(path: &Path, pattern: &str, matches: &mut Vec<PathBuf>) -> io::Result<()> {
    for entry in fs::read_dir(path)? {
        let entry = entry?;
        let path = entry.path();

        if path.is_dir() {
            search(&path, pattern, matches)?;
        } else if path.file_name()
            .and_then(|n| n.to_str())
            .map(|n| n.contains(pattern))
            .unwrap_or(false)
        {
            matches.push(path);
        }
    }
    Ok(())
}

search(root, pattern, &mut matches)?;
Ok(matches)
}

```

Symlink loops: This code doesn't detect symlink cycles. If `/a/b` symlinks to `/a`, you'll recurse forever. Production code should track visited inodes (Unix) or use a depth limit.

Performance: For very large directories (millions of files), consider parallel traversal or using OS-specific optimizations (like Linux's `readdir64`).

Pattern 6: Process Spawning and Piping

Problem: Need to run external commands and capture output. Want to chain processes like Unix pipelines.

Solution: Use `Command::new()` with `.output()` (captures all), `.status()` (inherits streams), or `.spawn()` (returns immediately). Set `Stdio::piped()` to capture output.

Why It Matters: Integration with system tools essential for build scripts, testing, automation. Improper stream handling causes deadlocks (child blocks on full pipe, parent blocks reading).

Use Cases: Build scripts (invoke compilers), test runners (execute programs and check output), automation tools, implementing Unix pipelines (`cat|grep|wc`), subprocess orchestration.

Example: Basic Process Execution

```

use std::process::{Command, Stdio};
use std::io::{self, Write};

```

Example: Run command and capture output

This example walks through run command and capture output.

```

fn run_command() -> io::Result<()> {
    let output = Command::new("ls")
        .arg("-la")
        .output()?;
    // Waits for completion, captures all output

    println!("Status: {}", output.status);
    println!("Stdout: {}", String::from_utf8_lossy(&output.stdout));
    println!("Stderr: {}", String::from_utf8_lossy(&output.stderr));

    Ok(())
}

```

Example: Check if command succeeded

This example walks through check if command succeeded.

```

fn run_command_check() -> io::Result<()> {
    let status = Command::new("cargo")
        .arg("build")
        .status()?;
    // Inherits stdin/stdout/stderr, just waits for completion

    if status.success() {
        println!("Build succeeded!");
    } else {
        println!("Build failed with: {}", status);
    }

    Ok(())
}

```

Example: Run with environment variables

This example walks through run with environment variables.

```

fn run_with_env() -> io::Result<()> {
    let output = Command::new("printenv")
        .env("MY_VAR", "my_value")
        .env("ANOTHER_VAR", "another_value")
        .output?;

    println!("{}", String::from_utf8_lossy(&output.stdout));
    Ok(())
}

```

Example: Run in specific directory

This example walks through run in specific directory.

```
fn run_in_directory() -> io::Result<()> {
    let output = Command::new("pwd")
        .current_dir("/tmp")
        .output()?;
    println!("Working directory: {}", String::from_utf8_lossy(&output.stdout));
    Ok(())
}
```

API choices: - `.output()`: Captures stdout/stderr, waits for exit. Use for short commands.
- `.status()`: Inherits streams, just returns exit code. Use for interactive commands.
- `.spawn()`: Returns immediately with `Child` handle. Use for async or long-running processes.

Example: Streaming Output

Capturing all output before processing isn't always feasible. Long-running processes need real-time feedback.

```
use std::process::{Command, Stdio};
use std::io::{self, BufRead, BufReader};
```

Example: Stream stdout in real-time

This example walks through stream stdout in real-time.

```
fn stream_output() -> io::Result<()> {
    let mut child = Command::new("ping")
        .arg("-c")
        .arg("5")
        .arg("8.8.8.8")
        .stdout(Stdio::piped())
        .spawn()?;
    let stdout = child.stdout.take().unwrap();
    let reader = BufReader::new(stdout);

    for line in reader.lines() {
        println!("Output: {}", line?);
    }

    let status = child.wait()?;
    println!("Exit status: {}", status);
    Ok(())
}
```

Example: Capture both stdout and stderr separately

This example walks through capture both stdout and stderr separately.

```

// Requires threading to avoid deadlock
fn capture_both_streams() -> io::Result<()> {
    let mut child = Command::new("cargo")
        .arg("build")
        .stdout(Stdio::piped())
        .stderr(Stdio::piped())
        .spawn()?;
    let stdout = child.stdout.take().unwrap();
    let stderr = child.stderr.take().unwrap();

    // Read stdout in one thread
    let stdout_thread = std::thread::spawn(move || {
        let reader = BufReader::new(stdout);
        for line in reader.lines() {
            if let Ok(line) = line {
                println!("[OUT] {}", line);
            }
        }
    });
    // Read stderr in another thread
    let stderr_thread = std::thread::spawn(move || {
        let reader = BufReader::new(stderr);
        for line in reader.lines() {
            if let Ok(line) = line {
                eprintln!("[ERR] {}", line);
            }
        }
    });

    stdout_thread.join().unwrap();
    stderr_thread.join().unwrap();

    let status = child.wait()?;
    println!("Process exited with: {}", status);
    Ok(())
}

```

Deadlock warning: If you read stdout while the child is blocked writing to stderr (and vice versa), you deadlock. Use threads or async I/O to read both concurrently.

Example: Piping Between Processes

Unix pipelines (`cat file | grep pattern | wc -l`) chain processes, streaming data without intermediate files.

```

use std::process::{Command, Stdio};
use std::io::{self, Write};

```

Example: Pipe output from one command to another (ls | grep txt)

This example walks through pipe output from one command to another (ls | grep txt).

```
fn pipe_commands() -> io::Result<()> {
    let ls = Command::new("ls")
        .stdout(Stdio::piped())
        .spawn()?;
    let grep = Command::new("grep")
        .arg("txt")
        .stdin(Stdio::from(ls.stdout.unwrap()))
        .stdout(Stdio::piped())
        .spawn()?;
    let output = grep.wait_with_output()?;
    println!("{}: {}", String::from_utf8_lossy(&output.stdout));
    Ok(())
}
```

Example: Complex pipeline: cat file | grep pattern | wc -l

This example walks through complex pipeline: cat file | grep pattern | wc -l.

```
fn complex_pipeline(file: &str, pattern: &str) -> io::Result<()> {
    let cat = Command::new("cat")
        .arg(file)
        .stdout(Stdio::piped())
        .spawn()?;
    let grep = Command::new("grep")
        .arg(pattern)
        .stdin(Stdio::from(cat.stdout.unwrap()))
        .stdout(Stdio::piped())
        .spawn()?;
    let wc = Command::new("wc")
        .arg("-l")
        .stdin(Stdio::from(grep.stdout.unwrap()))
        .stdout(Stdio::piped())
        .spawn()?;
    let output = wc.wait_with_output()?;
    println!("Lines matching '{}': {}", pattern,
        String::from_utf8_lossy(&output.stdout).trim());
    Ok(())
}
```

How piping works: `Stdio::from(child.stdout.unwrap())` passes the child's stdout as stdin to the next process. The OS manages the buffer between processes.

Summary

This chapter covered synchronous I/O patterns:

1. **Buffered Reading/Writing:** BufReader/BufWriter for 1000x speedup via syscall amortization
2. **Standard Streams:** stdin/stdout/stderr separation, locking, flushing for correct CLI behavior
3. **Memory-Mapped I/O:** Zero-copy random access, database-style page reads
4. **Directory Traversal:** Recursive walking, filtering by pattern, avoiding symlink cycles
5. **Process Spawning:** Running commands, capturing output, Unix pipelines

Key Takeaways: - Always buffer I/O—unbuffered is 1000x slower - Separate stdout (data) from stderr (diagnostics) - Use mmap for random access, buffered I/O for sequential - Track inodes to prevent infinite recursion in directory walks - Use threads when reading multiple process streams

Performance Guidelines: - Buffering: $O(N)$ syscalls $\rightarrow O(N/8192)$ syscalls - Stream locking: 50x speedup for bulk writes - Mmap: $O(1)$ random access vs $O(N)$ seek+read - Directory walk: $O(N)$ with proper caching

Async I/O Patterns

This chapter explores async I/O patterns using Tokio. Handling thousands of concurrent connections, buffered streams, backpressure, connection pooling, and timeouts. Async I/O allows one thread to manage thousands of operations by yielding when blocked, solving the C10K problem.

Pattern 1: Tokio File and Network I/O

Problem: Synchronous I/O blocks threads waiting for data. One thread per connection doesn't scale—10K connections need 20GB RAM (2MB stack each).

Solution: Use Tokio async I/O where operations return futures you `.await`. TcpListener/TcpStream for network accept/read/write.

Why It Matters: Single thread handles 10K connections in 10MB memory (vs 20GB threaded). Solves C10K problem—web servers serve 10K+ concurrent users.

Use Cases: Web servers (HTTP, HTTPS), API gateways, chat servers, WebSocket servers, game servers, HTTP clients (concurrent requests), file servers (mixing file and network I/O), concurrent file processing, real-time data pipelines.

Example: Async File Operations

Read/write files without blocking async runtime. Concurrent file operations should parallelize.

```
// Note: Add to Cargo.toml:  
// [dependencies]  
// tokio = { version = "1", features = ["full"] }
```

```
use tokio::fs::{File, OpenOptions};  
use tokio::io::{self, AsyncReadExt, AsyncWriteExt};
```

Example: Read entire file into a String

This example walks through how to read entire file into a string.

```
async fn read_file(path: &str) -> io::Result<String> {  
    tokio::fs::read_to_string(path).await  
    // Convenience method: allocates a String, reads entire file  
    // Returns Err if file missing, unreadable, or contains invalid UTF-8  
}
```

Example: Read entire file into Vec

This example walks through how to read entire file into vec.

```
async fn read_bytes(path: &str) -> io::Result<Vec<u8>> {  
    tokio::fs::read(path).await  
    // Reads entire file into memory  
    // More efficient than read_to_string for binary data  
}
```

Example: Write string to file (overwrites existing content)

This example walks through how to write string to file (overwrites existing content).

```
async fn write_file(path: &str, content: &str) -> io::Result<()> {  
    tokio::fs::write(path, content).await  
    // Convenience method: creates file, writes content, closes file  
    // Overwrites existing file! Use append_to_file if you want to append  
}
```

Example: Manual read with buffer control

This example walks through manual read with buffer control.

```
async fn read_with_buffer(path: &str) -> io::Result<Vec<u8>> {  
    let mut file = File::open(path).await?;  
    let mut buffer = Vec::new();  
  
    // read_to_end() allocates as needed while reading  
    file.read_to_end(&mut buffer).await?  
    Ok(buffer)  
}
```

Example: Manual write with explicit handle

This example walks through manual write with explicit handle.

```
async fn write_with_handle(path: &str, data: &[u8]) -> io::Result<()> {
    let mut file = File::create(path).await?;

    // write_all() ensures all bytes are written (loops if needed)
    file.write_all(data).await?;

    // flush() ensures buffered data reaches the OS
    // (Tokio buffers writes internally for efficiency)
    file.flush().await?;
    Ok(())
}
```

Example: Append to existing file (or create if missing)

This example walks through how to append to existing file (or create if missing).

```
async fn append_to_file(path: &str, content: &str) -> io::Result<()> {
    let mut file = OpenOptions::new()
        .append(true) // Append mode: writes go to end of file
        .create(true) // Create file if it doesn't exist
        .open(path)
        .await?;

    file.write_all(content.as_bytes()).await?;
    file.write_all(b"\n").await?; // Add newline separator
    Ok(())
}
```

Example: Copy file asynchronously

This example walks through copy file asynchronously.

```
async fn copy_file(src: &str, dst: &str) -> io::Result<u64> {
    tokio::fs::copy(src, dst).await
    // Efficiently copies src to dst using OS-level optimizations when possible
}
```

Example: Example usage

This example walks through example usage.

```
#[tokio::main]
async fn main() -> io::Result<()> {
    // Read file asynchronously
    let content = read_file("example.txt").await?;
```

```
    println!("File content: {}", content);

    // Write file asynchronously
    write_file("output.txt", "Hello, async!").await?;

    Ok(())
}
```

Example: Async Line Reading

Reading line-by-line is essential for processing log files, CSV data, and other line-oriented formats. The async version uses `BufReader` to buffer reads efficiently, minimizing system calls.

Why buffering matters: Unbuffered reads perform one system call per byte or small chunk, which is catastrophically slow. A `BufReader` reads large chunks (8KB by default) into an internal buffer, then serves bytes from that buffer. This reduces system calls by 100-1000x, making line-by-line reading practical.

```
use tokio::fs::File;
use tokio::io::{self, AsyncBufReadExt, BufReader};
```

Example: Read all lines into memory

This example walks through how to read all lines into memory.

```
async fn read_lines(path: &str) -> io::Result<Vec<String>> {
    let file = File::open(path).await?;
    let reader = BufReader::new(file);
    let mut lines = Vec::new();

    let mut line_stream = reader.lines();
    // lines() returns a stream that yields one line at a time
    // Lines are split on \n or \r\n, with the newline removed

    while let Some(line) = line_stream.next_line().await? {
        lines.push(line);
    }

    Ok(lines)
}
```

Example: Process large file line by line without loading into memory

This example walks through process large file line by line without loading into memory.

```
async fn process_large_file(path: &str) -> io::Result<()> {
    let file = File::open(path).await?;
    let reader = BufReader::new(file);
    let mut lines = reader.lines();
```

```

let mut count = 0;
while let Some(line) = lines.next_line().await? {
    if !line.starts_with('#') {
        // Process non-comment lines
        // Each line is processed and dropped before reading the next
        count += 1;
    }
}

println!("Processed {} lines", count);
Ok(())
}

```

Example: Read first N lines (useful for previews or headers)

This example walks through how to read first n lines (useful for previews or headers).

```

async fn read_first_n_lines(path: &str, n: usize) -> io::Result<Vec<String>> {
    let file = File::open(path).await?;
    let reader = BufReader::new(file);
    let mut lines = reader.lines();
    let mut result = Vec::new();

    for _ in 0..n {
        if let Some(line) = lines.next_line().await? {
            result.push(line);
        } else {
            break; // File has fewer than n lines
        }
    }

    Ok(result)
}

```

Example: TCP Network I/O

TCP (Transmission Control Protocol) provides reliable, ordered, connection-oriented communication. Tokio's TCP implementation is truly async—it uses the OS's non-blocking I/O facilities (epoll on Linux, kqueue on BSD/macOS, IOCP on Windows) to wait for data without blocking threads.

The TCP Server Pattern

A typical async TCP server follows this pattern: 1. Bind a `TcpListener` to an address 2. Loop forever, accepting connections with `.accept().await` 3. For each connection, spawn a new task with `tokio::spawn` 4. Each task handles one client independently 5. When a client disconnects, its task completes and is cleaned up

This pattern allows one thread to handle thousands of concurrent connections. The runtime multiplexes all connections on a small thread pool.

```
use tokio::net::{TcpListener, TcpStream};  
use tokio::io::{self, AsyncReadExt, AsyncWriteExt};
```

Example: TCP Echo Server

This example walks through tcp echo server.

```
async fn run_tcp_server(addr: &str) -> io::Result<()> {  
    let listener = TcpListener::bind(addr).await?;  
    println!("Server listening on {}", addr);  
  
    loop {  
        // accept() awaits the next incoming connection  
        // Returns (socket, peer_address)  
        let (socket, addr) = listener.accept().await?;  
        println!("New connection from {}", addr);  
  
        // Spawn a task for each connection  
        // Each task runs independently, allowing concurrent clients  
        tokio::spawn(async move {  
            if let Err(e) = handle_client(socket).await {  
                eprintln!("Error handling client: {}", e);  
            }  
        });  
    }  
    // Note: This loop never exits. In production, you'd add graceful shutdown.  
}
```

Example: Handle a single client connection (echo protocol)

This example walks through how to handle a single client connection (echo protocol).

```
async fn handle_client(mut socket: TcpStream) -> io::Result<()> {  
    let mut buffer = [0; 1024];  
  
    loop {  
        // read() awaits data from the client  
        // Returns number of bytes read, or 0 on EOF (client disconnected)  
        let n = socket.read(&mut buffer).await?;  
  
        if n == 0 {  
            // Client closed the connection gracefully  
            return Ok(());  
        }  
  
        // Echo the data back to the client  
        // write_all() ensures all bytes are sent (loops if the write is partial)  
        socket.write_all(&buffer[..n]).await?;
```

```
    }  
}
```

Example: TCP Client

This example walks through tcp client.

```
async fn tcp_client(addr: &str, message: &str) -> io::Result<String> {  
    // connect() performs DNS resolution (if needed) and TCP handshake  
    let mut stream = TcpStream::connect(addr).await?;  
  
    // Send message  
    stream.write_all(message.as_bytes()).await?;  
  
    // Read response  
    let mut buffer = Vec::new();  
    stream.read_to_end(&mut buffer).await?;  
    // Note: read_to_end() reads until EOF, which means the server  
    // must close the connection after responding  
  
    Ok(String::from_utf8_lossy(&buffer).to_string())  
}
```

Example: HTTP-like request handling (simplified)

This example walks through http-like request handling (simplified).

```
async fn http_handler(mut socket: TcpStream) -> io::Result<()> {  
    let mut buffer = [0; 4096];  
  
    // Read the HTTP request  
    let n = socket.read(&mut buffer).await?;  
  
    let request = String::from_utf8_lossy(&buffer[..n]);  
    println!("Request: {}", request);  
  
    // Send HTTP response  
    // In production, you'd parse the request and route to handlers  
    let response = "HTTP/1.1 200 OK\r\n"  
        Content-Type: text/plain\r\n"  
        Content-Length: 13\r\n"  
        \r\n"  
        Hello, World!";  
  
    socket.write_all(response.as_bytes()).await?  
    Ok(())  
}
```

Example: UDP Network I/O

UDP (User Datagram Protocol) is connectionless and unreliable—packets may arrive out of order, be duplicated, or be lost entirely. But UDP has advantages: lower latency (no connection setup), simpler protocol, and support for broadcast/multicast.

When to use UDP: - Low-latency gaming or real-time video where occasional packet loss is acceptable
- DNS queries (simple request-response where retries are easy)
- Service discovery and heartbeat protocols
- Metrics and logging where some data loss is tolerable

When to use TCP instead: - When you need guaranteed delivery and ordering
- When you're transferring bulk data (files, database replication)
- When you need flow control and congestion management

```
use tokio::net::UdpSocket;
use tokio::io;
```

Example: UDP Echo Server

This example walks through udp echo server.

```
async fn udp_server(addr: &str) -> io::Result<()> {
    let socket = UdpSocket::bind(addr).await?;
    println!("UDP server listening on {}", addr);

    let mut buffer = [0; 1024];

    loop {
        // recv_from() awaits a datagram from any sender
        // Returns (bytes_received, sender_address)
        let (len, addr) = socket.recv_from(&mut buffer).await?;
        println!("Received {} bytes from {}", len, addr);

        // Echo the datagram back to the sender
        // UDP doesn't guarantee delivery, so send_to() might succeed
        // even if the datagram never arrives
        socket.send_to(&buffer[..len], addr).await?;
    }
}
```

Example: UDP Client

This example walks through udp client.

```
async fn udp_client(server_addr: &str, message: &str) -> io::Result<String> {
    // Bind to a random local port (0.0.0.0:0 means "any port")
    let socket = UdpSocket::bind("0.0.0.0:0").await?;

    // Send datagram to server
    socket.send_to(message.as_bytes(), server_addr).await?;
```

```

// Wait for response
let mut buffer = [0; 1024];
let (len, _) = socket.recv_from(&mut buffer).await?;
Ok(String::from_utf8_lossy(&buffer[..len]).to_string())
}

```

Example: Unix Domain Sockets

Unix domain sockets provide inter-process communication (IPC) on the same machine. They're faster than TCP (no network stack overhead) and support passing file descriptors between processes.

When to use Unix sockets:

- Communication between processes on the same machine (Docker daemon, database connections)
- When you need higher performance than TCP for local IPC
- When you want filesystem-based access control (socket file permissions)

```

#[cfg(unix)]
mod unix_sockets {
    use tokio::net::{UnixListener, UnixStream};
    use tokio::io::{self, AsyncReadExt, AsyncWriteExt};

    pub async fn unix_server(path: &str) -> io::Result<()> {
        // Remove old socket file if exists (bind fails if file exists)
        let _ = std::fs::remove_file(path);

        let listener = UnixListener::bind(path)?;
        println!("Unix socket server listening on {}", path);

        loop {
            let (mut socket, _) = listener.accept().await?;

            // Spawn task for each connection (same pattern as TCP)
            tokio::spawn(async move {
                let mut buffer = [0; 1024];

                while let Ok(n) = socket.read(&mut buffer).await {
                    if n == 0 {
                        break;
                    }
                    let _ = socket.write_all(&buffer[..n]).await;
                }
            });
        }
    }

    pub async fn unix_client(path: &str, message: &str) -> io::Result<String> {
        let mut stream = UnixStream::connect(path).await?;

        stream.write_all(message.as_bytes()).await?;

        let mut buffer = Vec::new();

```

```
    stream.read_to_end(&mut buffer).await?;
    Ok(String::from_utf8_lossy(&buffer).to_string())
}
}
```

Pattern 2: Buffered Async Streams

Problem: Byte-by-byte async reads/writes trigger many syscalls—catastrophically inefficient. Need to batch I/O operations.

Solution: Use AsyncBufReadExt trait with read_line(), lines() for text. BufReader wraps async readers (File, TcpStream) with 8KB default buffer.

Why It Matters: Buffering reduces syscalls by 100x (byte-by-byte → chunked). Reading 1MB file: unbuffered = 1M syscalls, buffered = ~128 syscalls.

Use Cases: Line-based protocols (HTTP headers, SMTP, Redis protocol), chat protocols (newline-delimited), log streaming, CSV/JSON over network, codec-based protocols (protobuf, MessagePack), WebSocket framing, custom protocol parsers.

Example: Using BufReader and BufWriter

Minimize syscalls for async read/write operations with batching.

```
use tokio::fs::File;
use tokio::io::{self, AsyncBufReadExt, AsyncWriteExt, BufReader, BufWriter};
```

Example: Buffered reading with custom buffer size

This example walks through buffered reading with custom buffer size.

```
async fn buffered_read(path: &str) -> io::Result<()> {
    let file = File::open(path).await?;

    // Create BufReader with 8KB buffer (adjust based on your workload)
    let reader = BufReader::with_capacity(8192, file);
    let mut lines = reader.lines();

    while let Some(line) = lines.next_line().await? {
        println!("{}", line);
        // Processing each line is fast because BufReader minimizes system calls
    }

    Ok(())
}
```

Example: Buffered writing with custom buffer size

This example walks through buffered writing with custom buffer size.

```
async fn buffered_write(path: &str, lines: &[&str]) -> io::Result<()> {
    let file = File::create(path).await?;

    // BufWriter accumulates writes, flushes when buffer is full
    let mut writer = BufWriter::with_capacity(8192, file);

    for line in lines {
        // These writes don't immediately hit disk—they go to the buffer
        writer.write_all(line.as_bytes()).await?;
        writer.write_all(b"\n").await?;
    }

    // flush() ensures all buffered data is written to disk
    // Without this, some data might remain in the buffer!
    writer.flush().await?;
    Ok(())
}
```

Example: Copy with buffering

This example walks through copy with buffering.

```
async fn buffered_copy(src: &str, dst: &str) -> io::Result<u64> {
    let src_file = File::open(src).await?;
    let dst_file = File::create(dst).await?;

    let mut reader = BufReader::new(src_file);
    let mut writer = BufWriter::new(dst_file);

    // copy() efficiently transfers data, using the buffers to minimize system calls
    tokio::io::copy(&mut reader, &mut writer).await
}
```

Example: Stream Processing with AsyncRead/AsyncWrite

Implementing custom [AsyncRead](#) or [AsyncWrite](#) allows you to transform data as it's read or written. This is useful for encryption, compression, encoding, or protocol framing.

When to implement AsyncRead/AsyncWrite: - Building a custom protocol or encoding layer - Adding transparent encryption/compression - Implementing adapters between different I/O types - Creating testable mock I/O objects

```
use tokio::io::{self, AsyncRead, AsyncReadExt, AsyncWrite, AsyncWriteExt};
use std::pin::Pin;
use std::task::{Context, Poll};
```

Example: Custom async reader that uppercases data

This example walks through custom async reader that uppercases data.

```
struct UppercaseReader<R> {
    inner: R,
}

impl<R: AsyncRead + Unpin> AsyncRead for UppercaseReader<R> {
    fn poll_read(
        mut self: Pin<&mut Self>,
        cx: &mut Context<'_>,
        buf: &mut tokio::io::ReadBuf<'_>,
    ) -> Poll<io::Result<()>> {
        // Track how many bytes were in the buffer before reading
        let before_len = buf.filled().len();

        // Delegate to the inner reader
        match Pin::new(&mut self.inner).poll_read(cx, buf) {
            Poll::Ready(Ok(())) => {
                // Uppercase the newly read bytes
                let filled = buf.filled_mut();
                for byte in &mut filled[before_len..] {
                    if byte.is_ascii_lowercase() {
                        *byte = byte.to_ascii_uppercase();
                    }
                }
                Poll::Ready(Ok(()))
            }
            other => other,
        }
    }
}
```

Example: Usage example

This example walks through usage example.

```
async fn use_uppercase_reader() -> io::Result<()> {
    use tokio::fs::File;

    let file = File::open("input.txt").await?;
    let mut reader = UppercaseReader { inner: file };

    let mut buffer = String::new();
    reader.read_to_string(&mut buffer).await?;
    println!("Uppercased: {}", buffer);

    Ok(())
}
```

Example: Stream Splitting and Framing

Many protocols benefit from splitting a stream into independent read and write halves. This allows one task to read while another writes, or enables implementing full-duplex protocols where requests and responses flow concurrently.

```
use tokio::net::TcpStream;
use tokio::io::{self, AsyncRead, AsyncWrite, AsyncWriteExt, BufReader};
```

Example: Split stream into read and write halves

This example walks through split stream into read and write halves.

```
async fn split_stream_example() -> io::Result<()> {
    let stream = TcpStream::connect("127.0.0.1:8080").await?;

    // into_split() divides the stream into independent halves
    // The read half can be used in one task, write half in another
    let (read_half, write_half) = stream.into_split();

    // Spawn reader task
    let reader_task = tokio::spawn(async move {
        let mut reader = BufReader::new(read_half);
        let mut lines = reader.lines();

        while let Some(line) = lines.next_line().await? {
            println!("Received: {}", line);
        }
    });

    Ok::<_, io::Error>(())
};

// Spawn writer task
let writer_task = tokio::spawn(async move {
    let mut writer = write_half;

    for i in 0..10 {
        writer.write_all(format!("Message {}\n", i).as_bytes()).await?;
    }

    Ok::<_, io::Error>(())
});

// Wait for both tasks to complete
reader_task.await??;
writer_task.await??;

Ok(())
}
```

Example: Using `tokio_util::codec` for Framing

Framing solves a fundamental problem in network protocols: how do you know where one message ends and the next begins? Raw TCP gives you a byte stream with no message boundaries. Codecs provide message framing—they handle splitting the stream into discrete messages and vice versa.

Common framing strategies: - **Line-delimited**: Messages separated by newlines (simple, human-readable) - **Length-prefixed**: Each message starts with its length (efficient, binary-safe) - **Fixed-size**: All messages are the same size (simple but inflexible) - **Delimiter-based**: Messages separated by a special byte sequence (like HTTP headers separated by `\r\n\r\n`)

```
// Add to Cargo.toml:  
// tokio-util = { version = "0.7", features = ["codec"] }  
  
use tokio::net::TcpStream;  
use tokio_util::codec::{Framed, LinesCodec};  
use futures::{SinkExt, StreamExt};
```

Example: Line-delimited codec

This example walks through line-delimited codec.

```
async fn framed_lines() -> io::Result<()> {  
    let stream = TcpStream::connect("127.0.0.1:8080").await?;  
  
    // Framed wraps a stream and codec, providing a Stream of messages  
    // and a Sink for sending messages  
    let mut framed = Framed::new(stream, LinesCodec::new());  
  
    // Send lines (Sink interface)  
    framed.send("Hello, World!".to_string()).await?;  
    framed.send("Another line".to_string()).await?;  
  
    // Receive lines (Stream interface)  
    while let Some(result) = framed.next().await {  
        match result {  
            Ok(line) => println!("Received: {}", line),  
            Err(e) => eprintln!("Error: {}", e),  
        }  
    }  
    Ok(())  
}
```

Example: Custom codec for length-prefixed messages

This example walks through custom codec for length-prefixed messages.

```
use bytes::{Buf, BufMut, BytesMut};  
use tokio_util::codec::{Decoder, Encoder};
```

```

struct LengthPrefixedCodec;

impl Decoder for LengthPrefixedCodec {
    type Item = Vec<u8>;
    type Error = io::Error;

    fn decode(&mut self, src: &mut BytesMut) -> Result<Option<Self::Item>, Self::Error> {
        // Need at least 4 bytes for the length prefix
        if src.len() < 4 {
            return Ok(None); // Need more data
        }

        // Read the length prefix (big-endian u32)
        let mut length_bytes = [0u8; 4];
        length_bytes.copy_from_slice(&src[..4]);
        let length = u32::from_be_bytes(length_bytes) as usize;

        // Check if we have the complete message
        if src.len() < 4 + length {
            return Ok(None); // Need more data
        }

        // We have a complete message—extract it
        src.advance(4); // Skip the length prefix
        let data = src.split_to(length).to_vec();
        Ok(Some(data))
    }
}

impl Encoder<Vec<u8>> for LengthPrefixedCodec {
    type Error = io::Error;

    fn encode(&mut self, item: Vec<u8>, dst: &mut BytesMut) -> Result<(), Self::Error> {
        // Write length prefix
        let length = item.len() as u32;
        dst.put_u32(length);

        // Write message data
        dst.put_slice(&item);
        Ok(())
    }
}

async fn length_prefixed_example() -> io::Result<()> {
    let stream = TcpStream::connect("127.0.0.1:8080").await?;
    let mut framed = Framed::new(stream, LengthPrefixedCodec);

    // Send message (automatically prefixed with length)
    framed.send(b"Hello, World!".to_vec()).await?;

    // Receive message (automatically unpacked from length-prefixed format)
    if let Some(result) = framed.next().await {

```

```

        let data = result?;
        println!("Received {} bytes", data.len());
    }

    Ok(())
}

```

Pattern 3: Backpressure Handling

Problem: Fast producer overwhelms slow consumer causing unbounded memory growth. Web scraper fetches faster than parser processes—queue grows until OOM.

Solution: Use bounded mpsc channels with capacity limit—send() blocks when full. Semaphore limits concurrent operations (e.g., max 10 concurrent requests).

Why It Matters: Prevents OOM from unbounded queues—production systems must bound memory. Fast network source won't overwhelm slow disk sink.

Use Cases: Producer-consumer pipelines (network → processing → disk), streaming aggregation (sensor data, logs), rate-limited HTTP clients (respect API limits), connection pools (bound concurrent connections), download managers (limit concurrent downloads), WebSocket servers (per-client backpressure), data pipelines (ETL systems).

Example: Manual Backpressure with Bounded Channels

Control flow between producer and consumer to prevent memory exhaustion.

```

use tokio::sync::mpsc;
use tokio::time::{sleep, Duration};

```

Example: Producer with backpressure

This example walks through producer with backpressure.

```

async fn producer_with_backpressure(tx: mpsc::Sender<i32>) {
    for i in 0..100 {
        // send() provides backpressure—blocks when channel is full
        // This ensures we don't overwhelm the consumer
        if let Err(_) = tx.send(i).await {
            println!("Receiver dropped");
            break;
        }
        println!("Sent: {}", i);
    }
}

```

Example: Consumer (intentionally slow to demonstrate backpressure)

This example walks through consumer (intentionally slow to demonstrate backpressure).

```
async fn consumer(mut rx: mpsc::Receiver<i32>) {
    while let Some(value) = rx.recv().await {
        println!("Processing: {}", value);
        // Simulate slow processing
        sleep(Duration::from_millis(100)).await;
    }
}
```

Example: Usage with bounded channel

This example walks through usage with bounded channel.

```
async fn backpressure_example() {
    // Bounded channel with capacity 10
    // Producer can get ahead by 10 items, then must wait
    let (tx, rx) = mpsc::channel(10);

    let producer = tokio::spawn(producer_with_backpressure(tx));
    let consumer = tokio::spawn(consumer(rx));

    let _ = tokio::join!(producer, consumer);
}
```

Example: Stream Backpressure with Buffering

The `buffered()` combinator limits the number of futures executing concurrently, providing backpressure for stream processing.

```
use futures::stream::{self, StreamExt};
use tokio::time::{sleep, Duration};

async fn stream_backpressure() {
    let stream = stream::iter(0..100)
        .map(|i| async move {
            println!("Generating: {}", i);
            i
        })
        // buffered(5) means at most 5 futures run concurrently
        // This provides backpressure: we won't start future #6 until one completes
        .buffered(5)
        .for_each(|value| async move {
            println!("Processing: {}", value);
            sleep(Duration::from_millis(100)).await;
        })
}
```

```
    .await;
}
```

Example: Rate Limiting

Rate limiting controls the rate of operations to avoid overwhelming downstream systems or respecting API rate limits.

- When to use rate limiting:**
 - Calling third-party APIs with rate limits (e.g., “100 requests per minute”)
 - Protecting downstream services from being overwhelmed
 - Implementing fair scheduling among multiple clients
 - Smoothing bursty traffic

```
use tokio::time::{sleep, Duration, Instant};
```

Example: Simple rate limiter (token bucket algorithm)

This example walks through simple rate limiter (token bucket algorithm).

```
struct RateLimiter {
    max_per_second: u32,
    last_reset: Instant,
    count: u32,
}

impl RateLimiter {
    fn new(max_per_second: u32) -> Self {
        RateLimiter {
            max_per_second,
            last_reset: Instant::now(),
            count: 0,
        }
    }

    async fn acquire(&mut self) {
        let now = Instant::now();
        let elapsed = now.duration_since(self.last_reset);

        // Reset counter every second
        if elapsed >= Duration::from_secs(1) {
            self.last_reset = now;
            self.count = 0;
        }

        // If we've hit the rate limit, wait until next second
        if self.count >= self.max_per_second {
            let wait_time = Duration::from_secs(1) - elapsed;
            sleep(wait_time).await;
            self.last_reset = Instant::now();
            self.count = 0;
        }
    }
}
```

```

        self.count += 1;
    }

}

async fn rate_limited_requests() {
    let mut limiter = RateLimiter::new(10); // 10 requests per second

    for i in 0..50 {
        // acquire() blocks if we've exceeded the rate limit
        limiter.acquire().await;
        println!("Request {}", i);
        // Make request...
    }
}

```

Example: Semaphore for Concurrency Control

A semaphore limits the number of concurrent operations. This is essential for controlling resource usage (database connections, file handles, HTTP requests).

Semaphore vs. Channel: - Use a **semaphore** when you just need to limit concurrency (e.g., "at most 5 concurrent HTTP requests") - Use a **channel** when you need to pass data between tasks with bounded buffering

```

use tokio::sync::Semaphore;
use std::sync::Arc;

async fn concurrent_with_limit() {
    // Semaphore with 5 permits = max 5 concurrent tasks
    let semaphore = Arc::new(Semaphore::new(5));

    let mut handles = vec![];

    for i in 0..20 {
        // acquire_owned() waits if no permits are available
        let permit = semaphore.clone().acquire_owned().await.unwrap();

        let handle = tokio::spawn(async move {
            println!("Task {} started", i);
            sleep(Duration::from_secs(1)).await;
            println!("Task {} completed", i);
            // Permit is dropped here, releasing it back to the semaphore
            drop(permit);
        });

        handles.push(handle);
    }

    // Wait for all tasks to complete
    for handle in handles {
        handle.await.unwrap();
    }
}

```

```
}
```

Example: Flow Control in Network Servers

Limiting concurrent connections prevents resource exhaustion and ensures fair service under load.

```
use tokio::net::TcpListener;
use tokio::sync::Semaphore;
use std::sync::Arc;
```

Example: TCP server with connection limit

This example walks through tcp server with connection limit.

```
async fn server_with_connection_limit(addr: &str, max_connections: usize) -> io::Result<()> {
    let listener = TcpListener::bind(addr).await?;
    let semaphore = Arc::new(Semaphore::new(max_connections));

    println!("Server listening on {} (max {} connections)", addr, max_connections);

    loop {
        let (socket, addr) = listener.accept().await?;

        // Try to acquire permit (blocks if at connection limit)
        let permit = semaphore.clone().acquire_owned().await.unwrap();

        println!("New connection from {} ({} slots available)",
            addr,
            semaphore.available_permits());

        tokio::spawn(async move {
            // Handle connection
            let _ = handle_client(socket).await;
            // Permit dropped here, freeing a connection slot
            drop(permit);
        });
    }
}
```

Pattern 4: Connection Pooling

Problem: Creating TCP/database connections expensive—100ms+ for TCP handshake + TLS + auth. Can't scale to 1 new connection per request (too slow).

Solution: Use bb8 or deadpool crates for production-ready pools. Configure min/max pool size (e.g., 10-50 connections).

Why It Matters: Reduces latency 100x—reuse (1ms) vs new connection (100ms). Prevents connection exhaustion: pool limits concurrent connections.

Use Cases: Database connection pools (Postgres, MySQL, Redis), HTTP client connection pools (reqwest with pool), gRPC connection pools, connection-limited APIs (respect limits), microservices (service-to-service), connection-expensive protocols (TLS, SSH).

Example: Connection Pool Pattern

Efficiently manage and reuse database or network connections.

```
use tokio::sync::Mutex;
use std::sync::Arc;

struct Connection {
    id: usize,
}

impl Connection {
    async fn new(id: usize) -> io::Result<Self> {
        // Simulate connection setup (DNS, TCP handshake, authentication)
        sleep(Duration::from_millis(100)).await;
        Ok(Connection { id })
    }

    async fn execute(&self, query: &str) -> io::Result<String> {
        println!("Connection {} executing: {}", self.id, query);
        sleep(Duration::from_millis(50)).await;
        Ok(format!("Result from connection {}", self.id))
    }
}

struct SimplePool {
    connections: Arc<Mutex<Vec<Connection>>>,
    max_size: usize,
}

impl SimplePool {
    async fn new(size: usize) -> io::Result<Self> {
        let mut connections = Vec::new();

        // Pre-create all connections
        for i in 0..size {
            connections.push(Connection::new(i).await?);
        }

        Ok(SimplePool {
            connections: Arc::new(Mutex::new(connections)),
            max_size: size,
        })
    }
}
```

```

async fn acquire(&self) -> io::Result<PooledConnection> {
    loop {
        let mut pool = self.connections.lock().await;

        if let Some(conn) = pool.pop() {
            // Got a connection from the pool
            return Ok(PooledConnection {
                conn: Some(conn),
                pool: self.connections.clone(),
            });
        }

        // No connections available—wait and retry
        drop(pool);
        sleep(Duration::from_millis(10)).await;
    }
}

```

Example: RAII wrapper: returns connection to pool on drop

This example walks through raii wrapper: returns connection to pool on drop.

```

struct PooledConnection {
    conn: Option<Connection>,
    pool: Arc<Mutex<Vec<Connection>>>,
}

impl PooledConnection {
    async fn execute(&self, query: &str) -> io::Result<String> {
        self.conn.as_ref().unwrap().execute(query).await
    }
}

impl Drop for PooledConnection {
    fn drop(&mut self) {
        if let Some(conn) = self.conn.take() {
            // Return connection to pool
            let pool = self.pool.clone();
            tokio::spawn(async move {
                pool.lock().await.push(conn);
            });
        }
    }
}

```

Example: Usage

This example walks through usage.

```

async fn use_pool() -> io::Result<()> {
    let pool = SimplePool::new(5).await?;

    let mut handles = vec![];

    // Spawn 20 tasks, but only 5 connections exist
    // Tasks will wait for connections to become available
    for i in 0..20 {
        let pool = pool.clone();
        let handle = tokio::spawn(async move {
            let conn = pool.acquire().await.unwrap();
            let result = conn.execute(&format!("Query {}", i)).await.unwrap();
            println!("{}: {}", i, result);
            // Connection returned to pool when `conn` is dropped
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.await.unwrap();
    }

    Ok(())
}

```

Example: Advanced Pool with deadpool

`deadpool` is a production-ready connection pool library with features like connection recycling, timeouts, and health checks.

```

// Add to Cargo.toml:
// deadpool = "0.10"

use deadpool::managed::{Manager, Pool, RecycleResult};
use async_trait::async_trait;

struct MyConnection {
    id: usize,
}

struct MyManager {
    next_id: Arc<Mutex<usize>>,
}

#[async_trait]
impl Manager for MyManager {
    type Type = MyConnection;
    type Error = io::Error;

    // Called when the pool needs to create a new connection
    async fn create(&self) -> Result<MyConnection, io::Error> {

```

```

let mut id = self.next_id.lock().await;
let conn = MyConnection { id: *id };
*id += 1;
println!("Created connection {}", conn.id);
Ok(conn)
}

// Called when a connection is returned to the pool
// Allows health checks or cleanup before reuse
async fn recycle(&self, conn: &mut MyConnection) -> RecycleResult<io::Error> {
    println!("Recycling connection {}", conn.id);
    // Could perform a health check here (e.g., ping the database)
    Ok(())
}
}

async fn use_deadpool() -> io::Result<()> {
    let manager = MyManager {
        next_id: Arc::new(Mutex::new(0)),
    };

    let pool = Pool::builder(manager)
        .max_size(5) // Max 5 connections
        .build()
        .unwrap();

    let mut handles = vec![];

    for i in 0..20 {
        let pool = pool.clone();
        let handle = tokio::spawn(async move {
            // get() acquires a connection (waits if pool is exhausted)
            let conn = pool.get().await.unwrap();
            println!("Using connection {} for task {}", conn.id, i);
            sleep(Duration::from_millis(100)).await;
            // Connection returned to pool when dropped
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.await.unwrap();
    }
}

Ok(())
}

```

Example: HTTP Client Connection Pool

The `request` HTTP client has built-in connection pooling, which is essential for making many HTTP requests efficiently.

```

// Add to Cargo.toml:
// reqwest = "0.11"

use reqwest::Client;
use std::time::Duration;

async fn http_connection_pool() -> Result<(), Box<dyn std::error::Error>> {
    // reqwest Client has built-in connection pooling
    // Multiple requests to the same host reuse TCP connections
    let client = Client::builder()
        .pool_max_idle_per_host(10) // Keep up to 10 idle connections per host
        .pool_idle_timeout(Duration::from_secs(90)) // Close idle connections after 90s
        .timeout(Duration::from_secs(30)) // Request timeout
        .build()?;
}

let mut handles = vec![];

for i in 0..50 {
    let client = client.clone();
    let handle = tokio::spawn(async move {
        // Requests to the same host reuse connections from the pool
        let response = client
            .get(&format!("https://api.example.com/item/{}", i))
            .send()
            .await;

        match response {
            Ok(resp) => println!("Request {}: Status {}", i, resp.status()),
            Err(e) => eprintln!("Request {} failed: {}", i, e),
        }
    });
    handles.push(handle);
}

for handle in handles {
    handle.await?;
}

Ok(())
}

```

Pattern 5: Timeout and Cancellation

Problem: Async operations can hang forever without time bounds—network request to unresponsive server blocks indefinitely. Need to cancel slow operations to prevent resource leaks.

Solution: Use `tokio::time::timeout()` to wrap futures with duration limit—returns `Err` on timeout. `select!`

Why It Matters: Prevents resource leaks from hung operations—without timeout, stuck connections never close. HTTP requests must timeout (client disappeared, network partition).

Use Cases: HTTP request timeouts (prevent hung requests), database query timeouts (prevent slow queries), graceful shutdown (SIGTERM handling), user cancellation (browser closed, request canceled), health checks (must timeout), circuit breakers (timeout = failure signal), deadline propagation (gRPC deadlines), connection idle timeouts.

Example: Basic Timeout Pattern

Prevent operations from running indefinitely by setting time limits.

```
use tokio::time::{timeout, Duration};

async fn with_timeout() -> Result<(), Box
```

Example: Timeout with Fallback

A common pattern is to try a primary operation with a timeout, falling back to an alternative if it fails.

```
use tokio::time::{timeout, Duration};

async fn timeout_with_fallback() -> String {
    let result = timeout(
        Duration::from_secs(2),
        fetch_data_from_primary(),
    ).await;

    match result {
        Ok(Ok(data)) => data, // Primary succeeded
        _ => {
            // Primary timed out or failed-try fallback
            println!("Primary failed, trying fallback");
            fetch_data_from_fallback().await.unwrap_or_default()
        }
    }
}
```

```

}

async fn fetch_data_from_primary() -> io::Result<String> {
    sleep(Duration::from_secs(5)).await; // Too slow
    Ok("Primary data".to_string())
}

async fn fetch_data_from_fallback() -> io::Result<String> {
    sleep(Duration::from_millis(500)).await; // Fast fallback
    Ok("Fallback data".to_string())
}

```

Example: Cancellation with CancellationToken

CancellationToken provides a mechanism for coordinated cancellation across multiple tasks. When you cancel the token, all tasks listening to it are notified.

When to use CancellationToken: - Implementing graceful shutdown (cancel all background tasks) - Stopping a group of related tasks when one fails - Implementing request cancellation in a server

```

// Add to Cargo.toml:
// tokio-util = "0.7"

use tokio_util::sync::CancellationToken;

async fn cancellable_operation(token: CancellationToken) {
    let mut interval = tokio::time::interval(Duration::from_secs(1));

    loop {
        tokio::select! {
            // Check if cancellation was requested
            _ = token.cancelled() => {
                println!("Operation cancelled");
                break;
            }
            // Do work
            _ = interval.tick() => {
                println!("Working...");
            }
        }
    }
}

async fn cancellation_example() {
    let token = CancellationToken::new();
    let worker_token = token.clone();

    let worker = tokio::spawn(async move {
        cancellable_operation(worker_token).await;
    });

    // Let it run for 5 seconds

```

```

    sleep(Duration::from_secs(5)).await;

    // Cancel the operation
    token.cancel();

    worker.await.unwrap();
}

```

Example: Select for Racing Operations

`tokio::select!` runs multiple futures concurrently and returns as soon as one completes. This is useful for implementing timeouts, racing multiple data sources, or handling multiple events.

```
use tokio::time::{sleep, Duration};
```

Example: Race three operations—return the first to complete

This example walks through race three operations—return the first to complete.

```

async fn race_operations() -> String {
    tokio::select! {
        result = operation_a() => result,
        result = operation_b() => result,
        result = operation_c() => result,
    }
    // The other futures are dropped (canceled) when one completes
}

async fn operation_a() -> String {
    sleep(Duration::from_secs(3)).await;
    "A completed".to_string()
}

async fn operation_b() -> String {
    sleep(Duration::from_secs(1)).await;
    "B completed".to_string() // This completes first
}

async fn operation_c() -> String {
    sleep(Duration::from_secs(2)).await;
    "C completed".to_string()
}

```

Example: Biased select (checks branches in order)

This example walks through biased select (checks branches in order).

```
async fn biased_select() {
    let mut count = 0;
```

```

loop {
    tokio::select! {
        biased; // Check branches in order (not randomly)

        _ = sleep(Duration::from_millis(100)) => {
            count += 1;
            if count >= 10 {
                break;
            }
        }
        _ = async { println!("Other branch") } => {}
    }
}
}

```

Example: Graceful Shutdown

Graceful shutdown ensures that all in-flight requests complete before the server stops, preventing data loss or corrupted state.

```

use tokio::signal;
use tokio::sync::broadcast;

//=====
// TCP server with graceful shutdown
// When Ctrl+C is pressed, the server stops accepting new connections
// but waits for existing connections to finish
//=====

async fn graceful_shutdown_server() -> io::Result<()> {
    let (shutdown_tx, _) = broadcast::channel(1);
    let listener = TcpListener::bind("127.0.0.1:8080").await?;

    println!("Server running. Press Ctrl+C to shutdown.");

    // Spawn shutdown signal handler
    let shutdown_tx_clone = shutdown_tx.clone();
    tokio::spawn(async move {
        signal::ctrl_c().await.expect("Failed to listen for Ctrl+C");
        println!("\nShutdown signal received");
        let _ = shutdown_tx_clone.send(());
    });

    loop {
        let mut shutdown_rx = shutdown_tx.subscribe();

        tokio::select! {
            result = listener.accept() => {
                match result {
                    Ok((socket, addr)) => {
                        println!("New connection from {}", addr);
                        let shutdown_rx = shutdown_tx.subscribe();
                    }
                }
            }
        }
    }
}

```

```

        tokio::spawn(async move {
            let _ = handle_connection_with_shutdown(socket,
shutdown_rx).await;
        });
    }
    Err(e) => eprintln!("Accept error: {}", e),
}
}
_ = shutdown_rx.recv() => {
    println!("Server shutting down");
    break;
}
}
}

println!("Server stopped");
Ok(())
}

async fn handle_connection_with_shutdown(
    mut socket: TcpStream,
    mut shutdown: broadcast::Receiver<()>,
) -> io::Result<()> {
    let mut buffer = [0; 1024];

    loop {
        tokio::select! {
            result = socket.read(&mut buffer) => {
                let n = result?;
                if n == 0 {
                    return Ok(());
                }
                socket.write_all(&buffer[..n]).await?;
            }
            _ = shutdown.recv() => {
                println!("Connection closing due to shutdown");
                return Ok(());
            }
        }
    }
}
}

```

Example: Timeout for Multiple Operations

When running multiple operations concurrently, you can apply a timeout to all of them collectively or to each individually.

```

use tokio::time::{timeout, Duration};
use futures::future::join_all;

```

Example: Collective timeout: all operations must complete within 5 seconds total

This example walks through collective timeout: all operations must complete within 5 seconds total.

```
async fn timeout_multiple() -> Result<Vec<String>, Box<dyn std::error::Error>> {
    let operations = vec![
        async_task(1),
        async_task(2),
        async_task(3),
    ];

    // Timeout applies to the entire join_all
    let result = timeout(
        Duration::from_secs(5),
        join_all(operations),
    ).await?;

    Ok(result.into_iter().map(|r| r.unwrap()).collect())
}

async fn async_task(id: usize) -> io::Result<String> {
    sleep(Duration::from_secs(1)).await;
    Ok(format!("Task {} completed", id))
}
```

Example: Individual timeouts: each operation has its own timeout

This example walks through individual timeouts: each operation has its own timeout.

```
async fn individual_timeouts() -> Vec<Result<String, tokio::time::error::Elapsed>> {
    let operations = vec![
        timeout(Duration::from_secs(1), async_task(1)),
        timeout(Duration::from_secs(2), async_task(2)),
        timeout(Duration::from_secs(3), async_task(3)),
    ];

    join_all(operations).await
        .into_iter()
        .map(|r| r.map(|inner| inner.unwrap()))
        .collect()
}
```

Example: Retry with Timeout

Combining retries with timeouts creates resilient operations that handle transient failures but don't hang forever.

```
use tokio::time::{sleep, timeout, Duration};
```

Example: Generic retry logic with timeout and exponential backoff

This example walks through generic retry logic with timeout and exponential backoff.

```
async fn retry_with_timeout<F, Fut, T>(
    mut operation: F,
    max_retries: usize,
    timeout_duration: Duration,
) -> Result<T, Box<dyn std::error::Error>>
where
    F: FnMut() -> Fut,
    Fut: std::future::Future<Output = io::Result<T>>,
{
    for attempt in 0..max_retries {
        match timeout(timeout_duration, operation()).await {
            Ok(Ok(result)) => return Ok(result), // Success
            Ok(Err(e)) => {
                eprintln!("Attempt {} failed: {}", attempt + 1, e);
            }
            Err(_) => {
                eprintln!("Attempt {} timed out", attempt + 1);
            }
        }
    }

    if attempt < max_retries - 1 {
        // Exponential backoff: wait longer after each failure
        let backoff = Duration::from_millis(100 * 2_u64.pow(attempt as u32));
        sleep(backoff).await;
    }
}

Err("All retry attempts failed".into())
}
```

Example: Usage

This example walks through usage.

```
async fn use_retry() -> Result<(), Box<dyn std::error::Error>> {
    let result = retry_with_timeout(
        || async {
            // Simulated unreliable operation
            if rand::random::f32() < 0.7 {
                Err(io::Error::new(io::ErrorKind::Other, "Random failure"))
            } else {
                Ok("Success!".to_string())
            }
        },
        5, // Max 5 retries
        Duration::from_secs(2), // 2-second timeout per attempt
    ).await?;
```

```

    println!("Result: {}", result);
    Ok(())
}

```

Example: Deadline-based Cancellation

Instead of relative timeouts, you can use absolute deadlines. This is useful when multiple operations share a common deadline.

```

use tokio::time::{sleep, timeout_at, Duration, Instant};

async fn deadline_based_processing() -> io::Result<()> {
    // All tasks must complete by this deadline
    let deadline = Instant::now() + Duration::from_secs(10);

    let tasks = vec![
        process_item(1, deadline),
        process_item(2, deadline),
        process_item(3, deadline),
    ];

    let results = futures::future::join_all(tasks).await;

    for (i, result) in results.iter().enumerate() {
        match result {
            Ok(value) => println!("Task {} succeeded: {}", i, value),
            Err(e) => println!("Task {} failed: {}", i, e),
        }
    }

    Ok(())
}

async fn process_item(id: usize, deadline: Instant) -> io::Result<String> {
    // timeout_at() uses an absolute deadline instead of a duration
    timeout_at(deadline, async move {
        sleep(Duration::from_secs(id as u64 * 2)).await;
        Ok(format!("Item {} processed", id))
    })
    .await
    .map_err(|_| io::Error::new(io::ErrorKind::TimedOut, "Deadline exceeded"))?
}

```

Summary

This chapter covered async I/O patterns using Tokio:

- 1. Tokio File and Network I/O:** Async primitives (TcpListener, tokio::fs), work-stealing scheduler, solves C10K problem

2. **Buffered Async Streams:** AsyncBufReadExt for line-by-line, BufReader/BufWriter, codecs for framing
3. **Backpressure Handling:** Bounded channels, Semaphore, buffer_unordered(n), prevents OOM
4. **Connection Pooling:** bb8/deadpool, reduces latency 100x, manages connection lifecycle
5. **Timeout and Cancellation:** timeout(), select!, CancellationToken, prevents resource leaks

Key Takeaways: - Async I/O enables single thread to handle 10K+ connections - Always use buffering - 100x syscall reduction - Bounded channels provide automatic backpressure - Connection pooling essential for databases/HTTP - Timeouts prevent hung operations—critical for production - Drop cancels futures automatically (structured concurrency)

Performance Guidelines: - Async for I/O-bound, sync for CPU-bound - Use spawn_blocking for blocking operations - Buffer size: 8KB default, adjust for workload - Connection pool: 10-50 connections typical - Always set timeouts on network operations

Production Patterns: - Graceful shutdown with CancellationToken - Per-request timeouts prevent slow requests blocking others - Circuit breakers rely on timeout detection - Backpressure prevents OOM under load - Health checks detect stale pooled connections

Common Pitfalls: - Unbounded channels cause OOM - No timeout = hung connections accumulate - Blocking in async tasks starves others - Missing backpressure = memory exhaustion - Connection pool without health checks keeps stale connections

Serialization Patterns

This chapter covers serialization patterns using serde; converting Rust types to/from JSON, binary formats, config files. Serde provides zero-cost abstraction: types separated from formats, derive macros generate optimal code, switch formats by changing one line.

Pattern 1: Serde Patterns

Problem: Writing manual serialization code for every type and format is tedious. Converting Person to JSON requires writing `to_json()`.

Solution: Derive Serialize and Deserialize traits using `#[derive]` macros. Serde generates format-agnostic serialization code.

Why It Matters: Zero-cost abstraction—compiled code as fast as hand-written. Switch formats by changing `serde_json` to `serde_cbor`—one line change.

Use Cases: REST APIs (JSON request/response), config files (TOML, YAML), RPC between Rust services (bincode—fastest), cross-language messaging (MessagePack, CBOR), database storage (serialize structs to JSONB), caching (bincode for speed), logging (structured logs to JSON).

Example: Basic Derive Pattern

Add serialization to custom types with minimal code.

```
// Add to Cargo.toml:
// [dependencies]
// serde = { version = "1.0", features = ["derive"] }
// serde_json = "1.0"

use serde::{Serialize, Deserialize};
```

Example: Deriving Serialize + Deserialize makes this struct work with any serde format

This example walks through deriving serialize + deserialize makes this struct work with any serde format.

```
#[derive(Serialize, Deserialize, Debug)]
struct Person {
    name: String,
    age: u32,
    email: String,
}

fn basic_serialization() -> Result<(), Box

```

Example: Field Attributes

Field attributes give you fine-grained control over how individual fields are serialized without writing custom code.

Common use cases: - **API compatibility**: Your Rust names don't match the external API (e.g., `user_name` vs `username`) - **Optional fields**: Omit `None` values to reduce payload size - **Sensitive data**: Skip serializing passwords or secrets - **Backward compatibility**: Accept old field names when deserializing - **Flattening**: Merge nested structs into a flat structure

```
use serde::Serialize, Deserialize;

#[derive(Serialize, Deserialize, Debug)]
struct User {
    // Rename: use "username" in JSON, but "name" in Rust code
    #[serde(rename = "username")]
    name: String,

    // Skip serializing if None (reduces JSON size)
    // Field is serialized as "middleName": "value" only when Some
    #[serde(skip_serializing_if = "Option::is_none")]
    middle_name: Option<String>,

    // Provide default value when deserializing if field is missing
    // If JSON doesn't include "age", this becomes 0
    #[serde(default)]
    age: u32,

    // Skip this field entirely (never serialize or deserialize)
    // Useful for runtime-only data or secrets
    #[serde(skip)]
    password_hash: String,

    // Accept multiple names during deserialization
    // Deserializes from "mail", "e-mail", or "email"
    #[serde(alias = "mail", alias = "e-mail")]
    email: String,

    // Flatten: merge nested struct's fields into parent
    // Instead of "metadata": {"created_at": ...}
    // you get "created_at": ... at the top level
    #[serde(flatten)]
    metadata: Metadata,
}

#[derive(Serialize, Deserialize, Debug, Default)]
struct Metadata {
    created_at: Option<String>,
    updated_at: Option<String>,
}

fn field_attributes_example() -> Result<(), Box<dyn std::error::Error>> {
    let user = User {
        name: "Bob".to_string(),
        middle_name: None, // Won't appear in JSON
        age: 25,
        password_hash: "secret".to_string(), // Never serialized
    }
}
```

```

        email: "bob@example.com".to_string(),
        metadata: Metadata {
            created_at: Some("2024-01-01".to_string()),
            updated_at: None, // Won't appear in JSON
        },
    };

    let json = serde_json::to_string_pretty(&user)?;
    println!("{}", json);
    // Output:
    // {
    //     "username": "Bob",
    //     "age": 25,
    //     "email": "bob@example.com",
    //     "created_at": "2024-01-01"

```

Example: }

This example walks through }.

```

// Note: middle_name, password_hash, updated_at are omitted

Ok(())
}

```

Example: Container Attributes

Container attributes apply to the entire struct or enum, affecting how all fields are handled.

Common patterns: - **Case conversion:** Convert Rust's `snake_case` to `camelCase` for JavaScript APIs
- **Strict deserialization:** Reject unknown fields to catch API changes early - **Enum representation:** Control how enum variants are encoded

```
use serde::{Serialize, Deserialize};
```

Example: Rename all fields to camelCase automatically

This example walks through rename all fields to camelcase automatically.

```

// Rust: status_code → JSON: statusCode
#[derive(Serialize, Deserialize)]
#[serde(rename_all = "camelCase")]
struct ApiResponse {
    status_code: u32,           // → statusCode in JSON
    error_message: Option<String>, // → errorMessage in JSON
    response_data: Vec<String>,   // → responseData in JSON
}

//=====
// Deny unknown fields during deserialization

```

```

// Fails if JSON contains fields not defined in the struct
// Use this to detect API version mismatches early
//=====
#[derive(Serialize, Deserialize)]
#[serde(deny_unknown_fields)]
struct StrictConfig {
    host: String,
    port: u16,
}

//=====
// Tagged enum: adds a "type" field to identify the variant
// Useful for discriminated unions in JSON
//=====
#[derive(Serialize, Deserialize, Debug)]
#[serde(tag = "type")]
enum Message {
    Text { content: String },
    Image { url: String, width: u32, height: u32 },
    Video { url: String, duration: u32 },
}
// Serializes as: {"type": "Image", "url": "...", "width": 1920, "height": 1080}

//=====
// Untagged enum: no type field, variant determined by structure
// Serde tries to deserialize as each variant until one succeeds
//=====
#[derive(Serialize, Deserialize, Debug)]
#[serde(untagged)]
enum Value {
    Integer(i64),
    Float(f64),
    String(String),
    Bool(bool),
}
// Serializes as: 42, 3.14, "hello", or true (no type tag)

fn enum_serialization() -> Result<(), Box<dyn std::error::Error>> {
    let message = Message::Image {
        url: "https://example.com/image.jpg".to_string(),
        width: 1920,
        height: 1080,
    };

    let json = serde_json::to_string_pretty(&message)?;
    println!("Tagged enum:\n{}", json);
    // Output:
    // {
    //     "type": "Image",
    //     "url": "https://example.com/image.jpg",
    //     "width": 1920,
    //     "height": 1080
}

```

Example: }

This example walks through }.

```
let value = Value::String("hello".to_string());
let json = serde_json::to_string(&value)?;
println!("Untagged enum: {}", json);
// Output: "hello" (no type information)

Ok(())
}
```

Example: Custom Serialization Functions

Sometimes derive attributes aren't enough—you need to transform data during serialization. Custom functions give you precise control.

Use custom serializers for: - **Type conversion**: Serialize `Duration` as seconds instead of nanos - **Format conversion**: Serialize dates in a specific format - **Validation**: Ensure data meets constraints during deserialization - **Legacy compatibility**: Match quirky formats from old systems

```
use serde::{Serialize, Deserialize, Serializer, Deserializer};
use serde::de::{self, Visitor};
use std::fmt;

#[derive(Serialize, Deserialize, Debug)]
struct Config {
    // Serialize Duration as seconds (u64) instead of nanos
    // Makes JSON more readable: "timeout": 300 instead of "timeout": 300000000000
    #[serde(serialize_with = "serialize_duration", deserialize_with = "deserialize_duration")]
    timeout: std::time::Duration,

    // Custom date format
    #[serde(serialize_with = "serialize_date", deserialize_with = "deserialize_date")]
    created_at: chrono::NaiveDate,
}
```

Example: Convert Duration to seconds for serialization

This example walks through how to convert duration to seconds for serialization.

```
fn serialize_duration<S>(duration: &std::time::Duration, serializer: S) -> Result<S::Ok, S::Error>
where
    S: Serializer,
{
    // Convert to seconds and serialize as a simple number
```

```
    serializer.serialize_u64(duration.as_secs())
}
```

Example: Convert seconds back to Duration during deserialization

This example walks through how to convert seconds back to duration during deserialization.

```
fn deserialize_duration<'de, D>(deserializer: D) -> Result<std::time::Duration, D::Error>
where
    D: Deserializer<'de>,
{
    let secs = u64::deserialize(deserializer)?;
    Ok(std::time::Duration::from_secs(secs))
}

// Using chrono for date handling
use chrono::NaiveDate;
```

Example: Serialize date as “YYYY-MM-DD” string

This example walks through how to serialize date as “yyyy-mm-dd” string.

```
fn serialize_date<S>(date: &NaiveDate, serializer: S) -> Result<S::Ok, S::Error>
where
    S: Serializer,
{
    serializer.serialize_str(&date.format("%Y-%m-%d").to_string())
}
```

Example: Deserialize date from “YYYY-MM-DD” string

This example walks through how to deserialize date from “yyyy-mm-dd” string.

```
fn deserialize_date<'de, D>(deserializer: D) -> Result<NaiveDate, D::Error>
where
    D: Deserializer<'de>,
{
    // Implement Visitor pattern for type-safe deserialization
    struct DateVisitor;

    impl<'de> Visitor<'de> for DateVisitor {
        type Value = NaiveDate;

        fn expecting(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
            formatter.write_str("a date string in YYYY-MM-DD format")
        }

        fn visit_str<E>(self, value: &str) -> Result<NaiveDate, E>
        where
            E: de::Error,
```

```

    {
        // Parse string, convert parse errors to serde errors
        NaiveDate::parse_from_str(value, "%Y-%m-%d")
            .map_err(|e| E::custom(format!("Invalid date: {}", e)))
    }
}

deserializer.deserialize_str(DateVisitor)
}

```

Example: Custom Serialize/Deserialize Implementation

For complete control, implement `Serialize` and `Deserialize` manually. This is necessary for types with complex invariants or non-standard representations.

When to write manual implementations: - Your type has internal invariants that need validation - The default serialization doesn't match your needs - You need to support a legacy format - You want to serialize computed fields or skip internal state

```

use serde::{Serialize, Deserialize, Serializer, Deserializer};
use serde::ser::SerializeStruct;
use serde::de::{self, MapAccess, Visitor};
use std::fmt;

#[derive(Debug)]
struct Point {
    x: f64,
    y: f64,
}

```

Example: Manual Serialize implementation

This example walks through manual serialize implementation.

```

impl Serialize for Point {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: Serializer,
    {
        // Serialize as a struct with 2 fields
        let mut state = serializer.serialize_struct("Point", 2)?;
        state.serialize_field("x", &self.x)?;
        state.serialize_field("y", &self.y)?;
        state.end()
    }
}

```

Example: Manual Deserialize implementation

This example walks through manual deserialize implementation.

```

impl<'de> Deserialize<'de> for Point {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
    where
        D: Deserializer<'de>,
    {
        // Define field identifiers
        enum Field { X, Y }

        // Implement Deserialize for Field enum
        impl<'de> Deserialize<'de> for Field {
            fn deserialize<D>(deserializer: D) -> Result<Field, D::Error>
            where
                D: Deserializer<'de>,
            {
                struct FieldVisitor;

                impl<'de> Visitor<'de> for FieldVisitor {
                    type Value = Field;

                    fn expecting(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
                        formatter.write_str("`x` or `y`")
                    }

                    fn visit_str<E>(self, value: &str) -> Result<Field, E>
                    where
                        E: de::Error,
                    {
                        match value {
                            "x" => Ok(Field::X),
                            "y" => Ok(Field::Y),
                            _ => Err(de::Error::unknown_field(value, FIELDS)),
                        }
                    }
                }

                deserializer.deserialize_identifier(FieldVisitor)
            }
        }
    }

    // Implement visitor for the Point struct
    struct PointVisitor;

    impl<'de> Visitor<'de> for PointVisitor {
        type Value = Point;

        fn expecting(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
            formatter.write_str("struct Point")
        }

        fn visit_map<V>(self, mut map: V) -> Result<Point, V::Error>
        where
            V: MapAccess<'de>,
    }
}

```

```

{
    let mut x = None;
    let mut y = None;

    // Read each field from the map
    while let Some(key) = map.next_key()? {
        match key {
            Field::X => {
                if x.is_some() {
                    return Err(de::Error::duplicate_field("x"));
                }
                x = Some(map.next_value()?);
            }
            Field::Y => {
                if y.is_some() {
                    return Err(de::Error::duplicate_field("y"));
                }
                y = Some(map.next_value()?);
            }
        }
    }

    // Ensure required fields are present
    let x = x.ok_or_else(|| de::Error::missing_field("x"))?;
    let y = y.ok_or_else(|| de::Error::missing_field("y"))?;

    Ok(Point { x, y })
}
}

const FIELDS: [&str] = &["x", "y"];
deserializer.deserialize_struct("Point", FIELDS, PointVisitor)
}
}

```

Example: Serializing with State

Sometimes you need to include computed data or context during serialization. Custom [Serialize](#) implementations make this possible.

```

use serde::{Serialize, Serializer};
use std::collections::HashMap;

struct Database {
    users: HashMap<u64, String>,
}

```

Example: Custom serialization that includes computed data

This example walks through custom serialization that includes computed data.

```
// Adds a "user_count" field that isn't stored in the struct
impl Serialize for Database {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: Serializer,
    {
        use serde::ser::SerializeStruct;

        let mut state = serializer.serialize_struct("Database", 2)?;
        state.serialize_field("users", &self.users)?;
        // Serialize computed field
        state.serialize_field("user_count", &self.users.len())?;
        state.end()
    }
}
```

Example: Wrapper for custom serialization context

This example walks through wrapper for custom serialization context.

```
// Allows passing configuration to serialization logic
struct SerializeWithContext<'a, T> {
    value: &'a T,
    include_metadata: bool,
}

impl<'a, T: Serialize> Serialize for SerializeWithContext<'a, T> {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: Serializer,
    {
        if self.include_metadata {
            // Wrap value with metadata
            use serde::ser::SerializeStruct;
            let mut state = serializer.serialize_struct("WithMetadata", 2)?;
            state.serialize_field("data", self.value)?;
            state.serialize_field("serialized_at", &chrono::Utc::now().to_rfc3339())?;
            state.end()
        } else {
            // Serialize value directly
            self.value.serialize(serializer)
        }
    }
}
```

Pattern 2: Zero-Copy Deserialization

Problem: Deserializing allocates—parsing JSON with “name”:“Alice” allocates String for “Alice”. Processing 100K log lines allocates 100K strings wastefully.

Solution: Use `&str` and `&[u8]` in structs instead of `String` and `Vec`. Add `##[serde(borrow)]` attribute to enable borrowing.

Why It Matters: 10x faster for large inputs—no heap allocation. Constant memory: $O(1)$ vs $O(N)$ for allocating.

Use Cases: Log parsing (borrow from mmap'd file), HTTP request parsing (borrow from socket buffer), streaming data (process without allocating), embedded systems (RAM-constrained), high-throughput parsers (network protocols), zero-allocation servers.

Example: Zero-Copy Borrowing Pattern

Problem: Deserialize without allocating by borrowing from input buffer.

```
use serde::{Deserialize, Serialize};
```

Example: Zero-copy: borrows from the input string

This example walks through zero-copy: borrows from the input string.

```
// Lifetimes ensure the struct can't outlive the input
#[derive(Deserialize, Debug)]
struct BorrowedData<'a> {
    // #[serde(borrow)] tells serde to borrow instead of copying
    #[serde(borrow)]
    name: &'a str,
    #[serde(borrow)]
    description: &'a str,
    count: u32, // Primitive types are always copied
}

fn zero_copy_example() -> Result<(), Box<dyn std::error::Error>> {
    let json = r#"{"name": "Product", "description": "A great product", "count": 42}"#;

    // No string allocation happens here!
    // name and description borrow from the `json` string
    let data: BorrowedData = serde_json::from_str(json)?;

    println!("Name: {}", data.name);
    println!("Description: {}", data.description);
    println!("Count: {}", data.count);

    // data can't outlive json due to lifetime 'a
    // This won't compile: return data;

    Ok(())
}
```

Example: Cow (Clone on Write) for flexible ownership

This example walks through cow (clone on write) for flexible ownership.

```
// Borrows when possible, owns when necessary
#[derive(Deserialize, Serialize, Debug)]
struct FlexibleData<'a> {
    #[serde(borrow)]
    name: std::borrow::Cow<'a, str>,

    #[serde(borrow)]
    tags: std::borrow::Cow<'a, [String]>,
}

fn cow_example() -> Result<(), Box
```

Example: Using Bytes and ByteBuf

Binary data benefits even more from zero-copy deserialization. `serde_bytes` provides specialized handling for byte slices.

```
use serde::{Deserialize, Serialize};
use serde_bytes::{ByteBuf, Bytes};

#[derive(Serialize, Deserialize, Debug)]
struct BinaryData<'a> {
    // Serialize as compact binary array instead of JSON array of numbers
    #[serde(with = "serde_bytes")]
    owned_data: Vec<u8>,

    // Borrow from input buffer (zero-copy)
    #[serde(borrow, with = "serde_bytes")]
    borrowed_data: &'a [u8],
}
```

Example: More efficient for binary data

This example walks through more efficient for binary data.

```
// Without serde_bytes: [1, 2, 3, 4, 5] in JSON (13 bytes)
```

Example: With serde_bytes: "001002003004005" (compact representation)

This example walks through with serde_bytes: "001002003004005" (compact representation).

```
#[derive(Serialize, Deserialize, Debug)]
struct OptimizedBinaryData {
    #[serde(with = "serde_bytes")]
    data: Vec<u8>,
}

fn binary_data_example() -> Result<(), Box<dyn std::error::Error>> {
    let data = OptimizedBinaryData {
        data: vec![1, 2, 3, 4, 5],
    };

    // With serde_bytes, binary data is more efficiently encoded
    let json = serde_json::to_string(&data)?;
    println!("Serialized: {}", json);

    Ok(())
}
```

Example: Zero-Copy with bincode

Bincode is particularly well-suited for zero-copy deserialization because it's a binary format that doesn't need escape sequences or UTF-8 validation.

Example: Add to Cargo.toml:

This example walks through add to cargo.toml::

```
// bincode = "1.3"

use serde::{Deserialize, Serialize};

#[derive(Serialize, Deserialize, Debug)]
struct Record<'a> {
    id: u64,
    #[serde(borrow)]
    name: &'a str,
    #[serde(borrow)]
    data: &'a [u8],
}

fn bincode_zero_copy() -> Result<(), Box<dyn std::error::Error>> {
    let record = Record {
        id: 123,
        name: "Test",
    }
```

```

        data: &[1, 2, 3, 4, 5],
    };

    // Serialize to bytes (very compact)
    let encoded = bincode::serialize(&record)?;

    // Zero-copy deserialization: borrows from `encoded` buffer
    let decoded: Record = bincode::deserialize(&encoded)?;

    println!("Decoded: {:?}", decoded);

    Ok(())
}

```

Example: Custom Zero-Copy Deserializer

For advanced cases, implement custom deserializers that borrow from the input.

```

use serde::de::{self, Deserializer, Visitor};
use std::fmt;

```

Example: Custom deserializer for borrowed slices

This example walks through custom deserializer for borrowed slices.

```

fn deserialize_borrowed_slice<'de, D>(deserializer: D) -> Result<&'de [u8], D::Error>
where
    D: Deserializer<'de>,
{
    struct BorrowedSliceVisitor;

    impl<'de> Visitor<'de> for BorrowedSliceVisitor {
        type Value = &'de [u8];

        fn expecting(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
            formatter.write_str("a borrowed byte slice")
        }

        // This borrows directly from the input buffer
        fn visit_borrowed_bytes<E>(self, v: &'de [u8]) -> Result<Self::Value, E>
        where
            E: de::Error,
        {
            Ok(v)
        }
    }

    deserializer.deserialize_bytes(BorrowedSliceVisitor)
}

#[derive(Deserialize)]

```

```
struct CustomBorrowed<'a> {
    #[serde(deserialize_with = "deserialize_borrowed_slice")]
    data: &'a [u8],
}
```

Pattern 3: Schema Evolution

Problem: API changes break clients—adding “phone” field to User fails deserialization. Renaming “username” to “user_name” breaks all existing JSON.

Solution: Use `#[serde(default)]` for new optional fields—deserializes missing as `default()`. Use `#[serde(rename = "old_name")]` to keep wire format when refactoring.

Why It Matters: Enables gradual rollout—old clients work with new servers during migration. Adding fields backward compatible: v1 clients ignore new fields, v2 clients get defaults.

Use Cases: Versioned REST APIs (v1→v2 migration), database schema migrations (add columns without breaking old code), config file evolution (new options without breaking existing configs), backward-compatible protocols, gradual service updates (rolling deployment), refactoring without API breaks.

Example: Optional Field Pattern

Add new fields without breaking existing serialized data.

```
use serde::{Deserialize, Serialize};
```

Example: Version 1: Original schema

This example walks through version 1: original schema.

```
#[derive(Serialize, Deserialize, Debug)]
struct ConfigV1 {
    host: String,
    port: u16,
}
```

Example: Version 2: Add optional field with default

This example walks through version 2: add optional field with default.

```
// Can deserialize V1 data without errors
#[derive(Serialize, Deserialize, Debug)]
struct ConfigV2 {
    host: String,
    port: u16,

    // New field - defaults to None if missing
}
```

```
// Old JSON without "timeout" → timeout: None
#[serde(default)]
timeout: Option<u32>,
}
```

Example: Version 3: Required field with default

This example walks through version 3: required field with default.

```
#[derive(Serialize, Deserialize, Debug)]
struct ConfigV3 {
    host: String,
    port: u16,

    #[serde(default)]
    timeout: Option<u32>,

    // Defaults to 10 if missing
    // Allows reading old configs without this field
    #[serde(default = "default_max_connections")]
    max_connections: u32,
}

fn default_max_connections() -> u32 {
    10
}

fn schema_evolution_example() -> Result<(), Box<dyn std::error::Error>> {
    // Old JSON (v1) can be deserialized into new struct (v3)
    let old_json = r#"{"host": "localhost", "port": 8080}"#;
    let config: ConfigV3 = serde_json::from_str(old_json)?;

    println!("Config: {:?}", config);
    println!("Max connections (defaulted): {}", config.max_connections);
    // Output: max_connections: 10 (from default function)

    Ok(())
}
```

Example: Versioned Enums

```
use serde::{Deserialize, Serialize};
```

Example: Tag-based versioning: each variant is a schema version

This example walks through tag-based versioning: each variant is a schema version.

```
// The "version" field discriminates between versions
#[derive(Serialize, Deserialize, Debug)]
#[serde(tag = "version")]
```

```

enum VersionedMessage {
    #[serde(rename = "1")]
    V1 { content: String },

    #[serde(rename = "2")]
    V2 {
        content: String,
        timestamp: u64,
    },
}

#[serde(rename = "3")]
V3 {
    content: String,
    timestamp: u64,
    metadata: std::collections::HashMap<String, String>,
},
}

impl VersionedMessage {
    // Migrate any version to the latest
    fn to_latest(self) -> MessageV3 {
        match self {
            VersionedMessage::V1 { content } => MessageV3 {
                content,
                timestamp: 0, // Default for old messages
                metadata: Default::default(),
            },
            VersionedMessage::V2 { content, timestamp } => MessageV3 {
                content,
                timestamp,
                metadata: Default::default(),
            },
            VersionedMessage::V3 { content, timestamp, metadata } => MessageV3 {
                content,
                timestamp,
                metadata,
            },
        }
    }
}

#[derive(Debug)]
struct MessageV3 {
    content: String,
    timestamp: u64,
    metadata: std::collections::HashMap<String, String>,
}

```

Example: Handling Renamed Fields

```

use serde::{Deserialize, Serialize};

```

```

#[derive(Serialize, Deserialize, Debug)]
struct UserProfile {
    // Accept both old and new names during deserialization
    // Deserializes from "user_name", "userName", or "name"
    #[serde(alias = "user_name", alias = "userName")]
    name: String,

    // Serialize as "emailAddress", accept "email" or "emailAddress" when deserializing
    // This allows gradual migration: old clients use "email", new ones use "emailAddress"
    #[serde(rename = "emailAddress", alias = "email")]
    email_address: String,
}

fn renamed_fields_example() -> Result<(), Box<dyn std::error::Error>> {
    // Old format (uses old field names)
    let old_json = r#"{"user_name": "Alice", "email": "alice@example.com"}"#;
    let profile: UserProfile = serde_json::from_str(old_json)?;

    // New format (uses new field names)
    let new_json = serde_json::to_string_pretty(&profile)?;
    println!("New format:\n{}", new_json);
    // Output uses renamed fields:
    // {
    //   "name": "Alice",
    //   "emailAddress": "alice@example.com"
}

```

Example: }

This example walks through }.

```

Ok(())
}

```

Example: Custom Migration Logic

For complex migrations, implement custom deserialization logic.

```

use serde::{Deserialize, Deserializer};
use serde::de::{self, MapAccess, Visitor};
use std::fmt;

#[derive(Debug)]
struct MigratableConfig {
    host: String,
    port: u16,
    connection_string: String,
}

impl<'de> Deserialize<'de> for MigratableConfig {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
}

```

```

where
    D: Deserializer<'de>,
{
    #[derive(Deserialize)]
    #[serde(field_identifier, rename_all = "snake_case")]
    enum Field { Host, Port, ConnectionString }

    struct ConfigVisitor;

    impl<'de> Visitor<'de> for ConfigVisitor {
        type Value = MigratableConfig;

        fn expecting(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
            formatter.write_str("struct MigratableConfig")
        }

        fn visit_map<V>(self, mut map: V) -> Result<MigratableConfig, V::Error>
        where
            V: MapAccess<'de>,
        {
            let mut host = None;
            let mut port = None;
            let mut connection_string = None;

            while let Some(key) = map.next_key()? {
                match key {
                    Field::Host => host = Some(map.next_value()?),  

                    Field::Port => port = Some(map.next_value()?),  

                    Field::ConnectionString => connection_string =  

                        Some(map.next_value()?),  

                }
            }

            // Migration logic: build connection_string from host and port if missing
            // This allows old configs (with host+port) to work with new code (expects
            connection_string)
            let connection_string = if let Some(cs) = connection_string {
                cs
            } else {
                let host = host.ok_or_else(|| de::Error::missing_field("host"))?;
                let port = port.ok_or_else(|| de::Error::missing_field("port"))?;
                format!("{}:{}", host, port)
            };

            let host = host.ok_or_else(|| de::Error::missing_field("host"))?;
            let port = port.ok_or_else(|| de::Error::missing_field("port"))?;

            Ok(MigratableConfig {
                host,
                port,
                connection_string,
            })
        }
    }
}

```

```

    }

    deserializer.deserialize_struct(
        "MigratableConfig",
        &["host", "port", "connection_string"],
        ConfigVisitor,
    )
}

```

Pattern 4: Binary vs Text Formats

Problem: JSON human-readable but large and slow—100KB JSON → 40KB binary. Need cross-language format (bincode Rust-only).

Solution: Use JSON for APIs and debugging (human-readable, universal). Use bincode for Rust-to-Rust IPC (smallest, fastest—10x faster than JSON).

Why It Matters: JSON 2-5x larger than binary (100KB → 40KB MessagePack). Bincode 10x faster parse than JSON for Rust types.

Use Cases: JSON (REST APIs, web configs, debugging), bincode (Rust microservice IPC, caching, session storage), MessagePack (cross-language RPC, binary APIs), CBOR (IoT protocols, embedded systems), TOML (application configs), YAML (complex configs like Kubernetes), Protocol Buffers (Google services, strict schemas).

Example: Format Comparison Pattern

Choose optimal serialization format for use case.

| Format | Size | Speed | Human-readable | Interop | Self-describing |
|-------------|--------|--------|----------------|-----------|-----------------|
| JSON | Large | Slow | Yes | Excellent | Yes |
| Bincode | Tiny | Fast | No | Rust-only | No |
| MessagePack | Small | Fast | No | Excellent | Yes |
| CBOR | Small | Fast | No | Good | Yes |
| YAML | Large | Slow | Yes | Good | Yes |
| TOML | Medium | Medium | Yes | Good | Yes |

Example: JSON Pattern

Need human-readable format for APIs and configs.

Use JSON when: - Building web APIs (de facto standard) - Storing human-editable config files - Debugging (can inspect payloads easily) - Interoperating with JavaScript/browsers

```

use serde::{Deserialize, Serialize};

#[derive(Serialize, Deserialize, Debug)]

```

```

struct Product {
    id: u64,
    name: String,
    price: f64,
    in_stock: bool,
}

fn json_format() -> Result<(), Box<dyn std::error::Error>> {
    let product = Product {
        id: 12345,
        name: "Widget".to_string(),
        price: 29.99,
        in_stock: true,
    };

    // Serialize to JSON with pretty formatting
    let json = serde_json::to_string_pretty(&product)?;
    println!("JSON ({} bytes):\n{}", json.len(), json);
    // Output (~80 bytes with whitespace):
    // {
    //     "id": 12345,
    //     "name": "Widget",
    //     "price": 29.99,
    //     "in_stock": true
}

```

Example: }

This example walks through }.

```

// Deserialize back to Rust struct
let deserialized: Product = serde_json::from_str(&json)?;
println!("Deserialized: {:?}", deserialized);

Ok(())
}

```

Example: Bincode (Binary Format)

Bincode is the most compact binary format for Rust-to-Rust communication. Not self-describing—you must know the exact type to deserialize.

Use Bincode when: - Communicating between Rust services - Storing data where you control both writer and reader - Maximum performance is critical - Size matters (smallest format)

```

use serde::{Deserialize, Serialize};

#[derive(Serialize, Deserialize, Debug)]
struct Product {
    id: u64,
    name: String,
}

```

```

    price: f64,
    in_stock: bool,
}

fn bincode_format() -> Result<(), Box<dyn std::error::Error>> {
    let product = Product {
        id: 12345,
        name: "Widget".to_string(),
        price: 29.99,
        in_stock: true,
    };

    // Serialize to compact binary
    let encoded = bincode::serialize(&product)?;
    println!("Bincode ({} bytes): {:?}", encoded.len(), encoded);
    // Output: ~30 bytes (vs ~80 bytes JSON)

    // Deserialize from binary
    let decoded: Product = bincode::deserialize(&encoded)?;
    println!("Deserialized: {:?}", decoded);

    Ok(())
}

```

Example: MessagePack (Binary Format)

MessagePack is a binary format with broad language support. Good balance between size, speed, and interoperability.

Use MessagePack when: - Building cross-language binary protocols - Need compact format with better interop than Bincode - Real-time systems (gaming, IoT)

Example: Add to Cargo.toml:

This example walks through add to cargo.toml::

```

// rmp-serde = "1.1"

use serde::{Deserialize, Serialize};

#[derive(Serialize, Deserialize, Debug)]
struct Product {
    id: u64,
    name: String,
    price: f64,
    in_stock: bool,
}

fn messagepack_format() -> Result<(), Box<dyn std::error::Error>> {
    let product = Product {
        id: 12345,
        name: "Widget".to_string(),

```

```

        price: 29.99,
        in_stock: true,
    };

    // Serialize to MessagePack
    let encoded = rmp_serde::to_vec(&product)?;
    println!("MessagePack ({} bytes): {:?}", encoded.len(), encoded);
    // Output: ~35 bytes (compact, self-describing)

    // Deserialize
    let decoded: Product = rmp_serde::from_slice(&encoded)?;
    println!("Deserialized: {:?}", decoded);

    Ok(())
}

```

Example: CBOR (Binary Format)

CBOR (Concise Binary Object Representation) is similar to MessagePack but with more features (tags, indefinite-length encoding). Used in IoT and embedded systems.

Example: Add to Cargo.toml:

This example walks through add to cargo.toml::

```

// serde_cbor = "0.11"

use serde::{Deserialize, Serialize};

#[derive(Serialize, Deserialize, Debug)]
struct Product {
    id: u64,
    name: String,
    price: f64,
    in_stock: bool,
}

fn cbor_format() -> Result<(), Box<dyn std::error::Error>> {
    let product = Product {
        id: 12345,
        name: "Widget".to_string(),
        price: 29.99,
        in_stock: true,
    };

    // Serialize to CBOR
    let encoded = serde_cbor::to_vec(&product)?;
    println!("CBOR ({} bytes): {:?}", encoded.len(), encoded);

    // Deserialize
    let decoded: Product = serde_cbor::from_slice(&encoded)?;
    println!("Deserialized: {:?}", decoded);
}

```

```
    Ok(())
}
```

Example: YAML (Text Format)

YAML is very human-readable with minimal syntax. Great for config files, but the complex spec makes parsing slow and error-prone.

Example: Add to Cargo.toml:

This example walks through add to cargo.toml:..

```
// serde_yaml = "0.9"

use serde::{Deserialize, Serialize};

#[derive(Deserialize, Serialize, Debug)]
struct Product {
    id: u64,
    name: String,
    price: f64,
    in_stock: bool,
}

fn yaml_format() -> Result<(), Box<dyn std::error::Error>> {
    let product = Product {
        id: 12345,
        name: "Widget".to_string(),
        price: 29.99,
        in_stock: true,
    };

    // Serialize to YAML
    let yaml = serde_yaml::to_string(&product)?;
    println!("YAML ({} bytes):\n{}", yaml.len(), yaml);
    // Output:
    // id: 12345
    // name: Widget
    // price: 29.99
    // in_stock: true

    // Deserialize
    let deserialized: Product = serde_yaml::from_str(&yaml)?;
    println!("Deserialized: {:?}", deserialized);

    Ok(())
}
```

Example: TOML (Text Format)

TOML is designed for config files. Minimal, unambiguous syntax. Limited nesting makes it unsuitable for complex data structures.

Example: Add to Cargo.toml:

This example walks through add to cargo.toml::

```
// toml = "0.8"

use serde::{Deserialize, Serialize};

#[derive(Serialize, Deserialize, Debug)]
struct Config {
    database: DatabaseConfig,
    server: ServerConfig,
}

#[derive(Serialize, Deserialize, Debug)]
struct DatabaseConfig {
    host: String,
    port: u16,
    username: String,
}

#[derive(Serialize, Deserialize, Debug)]
struct ServerConfig {
    host: String,
    port: u16,
    workers: u32,
}

fn toml_format() -> Result<(), Box<dyn std::error::Error>> {
    let config = Config {
        database: DatabaseConfig {
            host: "localhost".to_string(),
            port: 5432,
            username: "admin".to_string(),
        },
        server: ServerConfig {
            host: "0.0.0.0".to_string(),
            port: 8080,
            workers: 4,
        },
    };
}

// Serialize to TOML
let toml = toml::to_string_pretty(&config)?;
println!("TOML ({} bytes):\n{}", toml.len(), toml);
// Output:
// [database]
```

```

// host = "localhost"
// port = 5432
// username = "admin"
//
// [server]
// host = "0.0.0.0"
// port = 8080
// workers = 4

// Deserialize
let deserialized: Config = toml::from_str(&toml)?;
println!("Deserialized: {:?}", deserialized);

Ok(())
}

```

Example: Format Comparison

Benchmark different formats to see the size difference:

```

use serde::{Deserialize, Serialize};

#[derive(Serialize, Deserialize, Debug, Clone)]
struct BenchmarkData {
    id: u64,
    name: String,
    values: Vec<f64>,
    metadata: std::collections::HashMap<String, String>,
}

fn format_comparison() -> Result<(), Box<dyn std::error::Error>> {
    let data = BenchmarkData {
        id: 12345,
        name: "Test Data".to_string(),
        values: vec![1.0, 2.0, 3.0, 4.0, 5.0],
        metadata: {
            let mut map = std::collections::HashMap::new();
            map.insert("key1".to_string(), "value1".to_string());
            map.insert("key2".to_string(), "value2".to_string());
            map
        },
    };

    // Compare sizes across formats
    let json = serde_json::to_string(&data)?;
    println!("JSON: {} bytes", json.len());

    let bincode = bincode::serialize(&data)?;
    println!("Bincode: {} bytes", bincode.len());

    let msgpack = rmp_serde::to_vec(&data)?;
    println!("MessagePack: {} bytes", msgpack.len());
}

```

```

let cbor = serde_cbor::to_vec(&data)?;
println!("CBOR: {} bytes", cbor.len());

println!("Binary formats are typically 30-50% smaller than JSON");
// Typical output:
// JSON: 120 bytes
// Bincode: 65 bytes (45% smaller)
// MessagePack: 75 bytes (38% smaller)
// CBOR: 78 bytes (35% smaller)

Ok(())
}

```

Pattern 5: Streaming Serialization

Problem: Serializing GB dataset exhausts memory—loading 10GB JSON into RAM fails. Can't process files larger than RAM.

Solution: Use streaming APIs—serialize/deserialize incrementally.

serde_json::Deserializer::from_reader with streaming_iterator pulls one item at time.

Why It Matters: O(1) memory vs O(N) for full load—process 10GB file in 10MB RAM. Enables processing files larger than RAM (logs, DB exports).

Use Cases: Large file processing (GB log files, database dumps), streaming APIs (server-sent events, WebSocket messages), incremental parsing (start processing before download completes), log aggregation (process logs as they arrive), ETL pipelines (transform data in stream), real-time analytics (process events as they occur).

Example: Streaming JSON Pattern

Process large JSON arrays without loading entire array into memory.

```

use serde::Serialize;
use std::io::{self, Write};

#[derive(Serialize)]
struct Record {
    id: u64,
    name: String,
    value: f64,
}

```

Example: Stream records as a JSON array without building the entire array in memory

This example walks through stream records as a json array without building the entire array in memory.

```

fn stream_json_array<W: Write>(mut writer: W, records: &[Record]) -> io::Result<()> {
    writer.write_all(b"[")?;

    for (i, record) in records.iter().enumerate() {
        if i > 0 {
            writer.write_all(b",")?;
        }

        // Serialize each record individually
        let json = serde_json::to_string(record)
            .map_err(|e| io::Error::new(io::ErrorKind::Other, e))?;

        writer.write_all(json.as_bytes())?;
    }

    writer.write_all(b"]")?;
    writer.flush()?;
}

Ok(())
}

fn streaming_array_example() -> io::Result<()> {
    let records = vec![
        Record { id: 1, name: "Alice".to_string(), value: 100.0 },
        Record { id: 2, name: "Bob".to_string(), value: 200.0 },
        Record { id: 3, name: "Carol".to_string(), value: 300.0 },
    ];

    let mut output = Vec::new();
    stream_json_array(&mut output, &records)?;

    println!("Streamed JSON: {}", String::from_utf8_lossy(&output));
}

Ok(())
}

```

Example: Streaming to File

JSON Lines (newline-delimited JSON) is perfect for streaming: one JSON object per line.

```

use serde::Serialize;
use std::fs::File;
use std::io::{self, BufWriter, Write};

#[derive(Serialize)]
struct LogEntry {
    timestamp: u64,
    level: String,
    message: String,
}

```

Example: Stream log entries to file (JSON Lines format)

This example walks through stream log entries to file (json lines format).

```
fn stream_to_file(path: &str) -> io::Result<()> {
    let file = File::create(path)?;
    let mut writer = BufWriter::new(file);

    // Write JSON lines (one JSON object per line)
    for i in 0..1000 {
        let entry = LogEntry {
            timestamp: i,
            level: "INFO".to_string(),
            message: format!("Log message {}", i),
        };

        let json = serde_json::to_string(&entry)
            .map_err(|e| io::Error::new(io::ErrorKind::Other, e))?;

        writeln!(writer, "{}", json)?;
    }

    writer.flush()?;
}

Ok(())
}
```

Example: Streaming Deserialization

```
use serde::Deserialize;
use std::io::{self, BufRead, BufferedReader};
use std::fs::File;

#[derive(Deserialize, Debug)]
struct LogEntry {
    timestamp: u64,
    level: String,
    message: String,
}
```

Example: Stream-process JSON Lines file

This example walks through stream-process json lines file.

```
fn stream_from_file(path: &str) -> io::Result<()> {
    let file = File::open(path)?;
    let reader = BufferedReader::new(file);

    // Process one line at a time (constant memory usage)
    for (line_num, line) in reader.lines().enumerate() {
```

```

let line = line?;

match serde_json::from_str::<LogEntry>(&line) {
    Ok(entry) => {
        // Process entry (filter, transform, aggregate, etc.)
        println!("Entry {}: {:?}", line_num, entry);
    }
    Err(e) => {
        eprintln!("Error parsing line {}: {}", line_num, e);
    }
}
Ok(())
}

```

Example: Streaming with serde_json::Deserializer

serde_json provides a streaming deserializer for processing multiple JSON values.

```

use serde::Deserialize;
use std::io::{self, Cursor};

#[derive(Deserialize, Debug)]
struct Item {
    id: u64,
    name: String,
}

fn streaming_deserializer() -> Result<(), Box

```

Example: Async Streaming with Tokio

Combine streaming serialization with async I/O for maximum efficiency.

```
use serde::{Deserialize, Serialize};
use tokio::io::{AsyncBufReadExt, AsyncWriteExt, BufReader};
use tokio::fs::File;

#[derive(Serialize, Deserialize, Debug)]
struct Record {
    id: u64,
    data: String,
}
```

Example: Async streaming write

This example walks through async streaming write.

```
async fn async_stream_write(path: &str) -> tokio::io::Result<()> {
    let mut file = File::create(path).await?;

    for i in 0..100 {
        let record = Record {
            id: i,
            data: format!("Data {}", i),
        };

        let json = serde_json::to_string(&record)
            .map_err(|e| tokio::io::Error::new(tokio::io::ErrorKind::Other, e))?;

        file.write_all(json.as_bytes()).await?;
        file.write_all(b"\n").await?;
    }

    file.flush().await?;
    Ok(())
}
```

Example: Async streaming read

This example walks through async streaming read.

```
async fn async_stream_read(path: &str) -> tokio::io::Result<()> {
    let file = File::open(path).await?;
    let reader = BufReader::new(file);
    let mut lines = reader.lines();

    while let Some(line) = lines.next_line().await? {
        match serde_json::from_str(&line) {
            Ok(record) => println!("Record: {:?}", record),
        }
    }
}
```

```

        Err(e) => eprintln!("Parse error: {}", e),
    }
}

Ok(())
}

```

Example: Large Dataset Streaming

For very large datasets, implement custom streaming writers with buffering and periodic flushing.

```

use serde::Serialize;
use std::io::{self, Write};

#[derive(Serialize)]
struct DataPoint {
    x: f64,
    y: f64,
    timestamp: u64,
}

```

Example: Custom streaming writer with automatic flushing

This example walks through custom streaming writer with automatic flushing.

```

struct DataStreamWriter<W: Write> {
    writer: W,
    count: usize,
}

impl<W: Write> DataStreamWriter<W> {
    fn new(mut writer: W) -> io::Result<Self> {
        writer.write_all(b"[")?;
        Ok(DataStreamWriter { writer, count: 0 })
    }

    fn write_point(&mut self, point: &DataPoint) -> io::Result<()> {
        if self.count > 0 {
            self.writer.write_all(b",")?;
        }

        let json = serde_json::to_string(point)
            .map_err(|e| io::Error::new(io::ErrorKind::Other, e))?;

        self.writer.write_all(json.as_bytes())?;
        self.count += 1;

        // Flush every 100 records to balance memory and I/O
        if self.count % 100 == 0 {
            self.writer.flush()?;
        }
    }
}

```

```

        Ok(())
    }

    fn finish(mut self) -> io::Result<()> {
        self.writer.write_all(b"]")?;
        self.writer.flush()?;
        Ok(())
    }
}

fn stream_large_dataset() -> io::Result<()> {
    let file = std::fs::File::create("dataset.json")?;
    let mut writer = DataStreamWriter::new(file)?;

    // Stream 1 million points without loading them all into memory
    for i in 0..1_000_000 {
        let point = DataPoint {
            x: i as f64,
            y: (i as f64).sin(),
            timestamp: i,
        };

        writer.write_point(&point)?;
    }

    writer.finish()?;
}

Ok(())
}

```

Example: Custom Streaming Format

For maximum efficiency, implement length-prefixed binary streaming.

```

use serde::Serialize;
use std::io::{self, Write};

```

Example: Length-prefixed binary format for streaming

This example walks through length-prefixed binary format for streaming.

```

struct BinaryStreamWriter<W: Write> {
    writer: W,
}

impl<W: Write> BinaryStreamWriter<W> {
    fn new(writer: W) -> Self {
        BinaryStreamWriter { writer }
    }
}

```

```

fn write_record<T: Serialize>(&mut self, record: &T) -> io::Result<()> {
    // Serialize to bytes
    let bytes = bincode::serialize(record)
        .map_err(|e| io::Error::new(io::ErrorKind::Other, e))?;

    // Write length prefix (4 bytes, big-endian)
    let len = bytes.len() as u32;
    self.writer.write_all(&len.to_be_bytes())?;

    // Write data
    self.writer.write_all(&bytes)?;

    Ok(())
}

fn flush(&mut self) -> io::Result<()> {
    self.writer.flush()
}
}

#[derive(Serialize)]
struct Message {
    id: u64,
    content: String,
}

fn binary_streaming_example() -> io::Result<()> {
    let mut writer = BinaryStreamWriter::new(Vec::new());

    for i in 0..10 {
        let msg = Message {
            id: i,
            content: format!("Message {}", i),
        };
        writer.write_record(&msg)?;
    }

    writer.flush()?;
}

Ok(())
}

```

Summary

This chapter covered serialization patterns using serde:

- 1. Serde Patterns:** Derive Serialize/Deserialize, field attributes (rename, skip, default), custom serializers
- 2. Zero-Copy Deserialization:** Borrow from input with `&str`, `#[serde(borrow)]`, 10x faster, O(1) memory

3. **Schema Evolution:** #[serde(default)] for new fields, rename/alias for compatibility, versioned enums
4. **Binary vs Text Formats:** JSON (readable), bincode (smallest/fastest), MessagePack (cross-language binary)
5. **Streaming Serialization:** StreamDeserializer, process GB files in MB RAM, incremental parsing

Key Takeaways: - Serde separates data structures from formats—one derive, all formats - Zero-cost abstraction: compiled code as fast as hand-written - Zero-copy deserialization 10x faster with O(1) memory - Schema evolution via default/rename/alias enables gradual rollout - Binary formats 2-5x smaller, 10x faster than JSON - Streaming essential for large files (process > RAM size)

Format Selection Guide: - **JSON:** REST APIs, debugging, human-readable configs - **Bincode:** Rust-to-Rust IPC, caching (smallest, fastest) - **MessagePack/CBOR:** Cross-language binary RPC - **TOML:** Simple application configs - **YAML:** Complex nested configs (Kubernetes)

Performance Guidelines: - Use zero-copy (&str) for high-throughput parsing - Binary formats for bandwidth/storage-constrained - Streaming for files > available RAM - JSON for debugging/development, binary for production

Production Patterns: - Schema evolution with #[serde(default)] for backward compatibility - Versioned APIs with tagged enums - Zero-copy for log parsing (10x throughput) - Streaming for large dataset processing - Format-agnostic types (support multiple formats)

Common Mistakes: - Forgetting #[serde(default)] when adding fields → breaks old data - Using String when &str would work → unnecessary allocation - Loading entire file before parsing → OOM for large files - Not versioning schemas → breaking changes painful - Choosing wrong format (JSON for everything) → performance problems

Declarative Macros

This chapter covers declarative macros (macro_rules!); pattern matching on syntax to generate code at compile-time. Macros enable variadic arguments, DSLs, and zero-cost abstractions impossible with functions. Pattern match input syntax, expand to template code.

Pattern 1: Macro Patterns and Repetition

Problem: Functions have fixed signatures—can't accept variable number of arguments. `println!("{} {}", a, b, c)` needs different function for each arg count.

Solution: Use macro_rules! to pattern-match syntax and generate code.

Why It Matters: Zero-cost abstraction—macro expansion compiles to optimal code, no runtime overhead. Variadic macros without variadics—`vec![1, 2, 3]` expands to optimal Vec construction.

Use Cases: Collection literals (`vec!`, `hashmap!`), variadic functions (`println!`, `format!`, `write!`), testing macros (`assert_eq!`, `assert!`), DSL construction (`sql!`, `html!`), builders (setters for all fields), trait implementations (for all numeric types), derive-like custom macros.

Example: Basic Pattern Matching

Create macros that accept different syntax patterns.

The simplest patterns: - () matches empty invocation: `my_macro!()` - (\$name:expr) matches an expression and binds it to \$name - Multiple patterns create different “overloads” for different invocation styles

Example: Simple macro without arguments

This example walks through simple macro without arguments.

```
macro_rules! say_hello {
    () => {
        println!("Hello, World!");
    };
}
```

Example: Macro with a single argument

This example walks through macro with a single argument.

```
// $name:expr means "match any expression, call it $name"
macro_rules! greet {
    ($name:expr) => {
        println!("Hello, {}!", $name);
    };
}
```

Example: Macro with multiple patterns

This example walks through macro with multiple patterns.

```
// This demonstrates how one macro can have different "overloads"
macro_rules! calculate {
    // Pattern 1: literal "add" followed by two expressions
    (add $a:expr, $b:expr) => {
        $a + $b
    };
    // Pattern 2: literal "sub" followed by two expressions
    (sub $a:expr, $b:expr) => {
        $a - $b
    };
    // Pattern 3: literal "mul" followed by two expressions
    (mul $a:expr, $b:expr) => {
        $a * $b
    };
}

fn basic_examples() {
```

```

say_hello!(); // Expands to: println!("Hello, World!");
greet!("Alice"); // Expands to: println!("Hello, {}!", "Alice");

let sum = calculate!(add 5, 3); // Expands to: 5 + 3
let product = calculate!(mul 4, 7); // Expands to: 4 * 7
println!("Sum: {}, Product: {}", sum, product);
}

```

Example: Fragment Specifiers

Fragment specifiers tell the macro what kind of syntax to expect. Each specifier matches a different part of Rust's grammar.

Why fragment specifiers matter: - **Type safety:** `$x:ty` only matches types, preventing runtime errors - **Flexibility:** Different fragments let you match exactly what you need - **Hygiene:** Some fragments (like `ident`) interact with macro hygiene

Common specifiers: - `expr`: Any expression (`2 + 2, vec![1, 2, 3], if x { y } else { z }`) - `ident`: An identifier (variable name, function name, etc.) - `ty`: A type (`i32, Vec<String>, &'a str`) - `pat`: A pattern (used in `match`, `let`, function parameters) - `stmt`: A statement (ends with `;`) - `block`: A block expression (`{ ... }`) - `item`: An item (function, struct, impl, etc.) - `tt`: Token tree (a single token or group in delimiters)

Example: Different fragment types

This example walks through different fragment types.

```

macro_rules! fragment_examples {
    // expr - expression
    // Matches: 5 + 3, vec![1, 2], function_call()
    ($e:expr) => {
        println!("Expression: {}", $e);
    };
    // ident - identifier
    // Matches: x, my_var, SomeStruct
    // This creates a variable with that name
    ($i:ident) => {
        let $i = 42;
    };
    // ty - type
    // Matches: i32, Vec<String>, &str
    ($t:ty) => {
        std::mem::size_of::<$t>()
    };
    // pat - pattern
    // Matches: Some(42), _, x @ 1..=5
    ($p:pat) => {
        match Some(42) {
            $p => println!("Matched!"),
            _ => println!("Not matched"),
        }
    };
}

```

```

};

// stmt - statement
// Matches: let x = 5;, println!("hi");
($s:stmt) => {
    $s
};

// block - block expression
// Matches: { let x = 5; x * 2 }
($b:block) => {
    $b
};

// item - item (function, struct, impl, etc.)
// Matches: fn foo() {}, struct Bar {}, impl Trait for Type {}
($it:item) => {
    $it
};

// meta - attribute contents
// Matches: derive(Debug), inline, cfg(test)
($m:meta) => {
    #[#$m]
    fn dummy() {}
};

// tt - token tree (single token or group in delimiters)
// Matches: x, (a, b), {code}, "string"
($tt:tt) => {
    stringify!($tt)
};

fn fragment_usage() {
    fragment_examples!(5 + 3); // expr: prints "Expression: 8"
    fragment_examples!(x); // ident: creates variable x = 42

    let size = fragment_examples!(i32); // ty: returns 4 (size of i32)
    println!("Size of i32: {}", size);
}

```

Example: Repetition Patterns

Repetitions are the most powerful feature of declarative macros. They let you match and generate variable amounts of code.

Repetition syntax: - `$(...)*` matches zero or more times - `$(...)+` matches one or more times - `$(...)?` matches zero or one time (optional) - Separator: `$(...),*` matches comma-separated items

Why repetitions matter: Creating `vec![1, 2, 3]` or `println!("{} {}", a, b)` requires matching an arbitrary number of elements—impossible without repetitions.

Example: Basic repetition with *

This example walks through basic repetition with `*`.

```
// This is how vec! works internally
macro_rules! create_vec {
    // $($elem:expr),* means:
    // - Match zero or more expressions
    // - Separated by commas
    // - Bind each to $elem
    ($($elem:expr),*) => {
        {
            let mut v = Vec::new();
            $(
                v.push($elem); // Repeat this for each matched $elem
            )*
            v
        }
    };
}
```

Example: Repetition with + (one or more)

This example walks through repetition with + (one or more).

```
// Requires at least one argument, unlike *
macro_rules! sum {
    ($($num:expr),+) => {
        {
            let mut total = 0;
            $(
                total += $num; // Repeat for each number
            )+
            total
        }
    };
}
```

Example: Optional repetition with ?

This example walks through optional repetition with ?.

```
// Allows an optional second argument
macro_rules! optional_value {
    ($val:expr $(, $default:expr)? => {
        Some($val) $.or(Some($default))?
    });
}
```

Example: Multiple repetitions

This example walks through multiple repetitions.

```

// This is how HashMap literals could work
macro_rules! hash_map {
    // Trailing comma is optional: $(,)??
    ($($key:expr => $val:expr),* $(,)??) => {
        {
            let mut map = std::collections::HashMap::new();
            $(
                map.insert($key, $val);
            )*
            map
        }
    };
}

fn repetition_examples() {
    // create_vec! accepts any number of arguments
    let v = create_vec![1, 2, 3, 4, 5];
    println!("Vector: {:?}", v);

    // sum! requires at least one argument
    let total = sum!(1, 2, 3, 4, 5);
    println!("Sum: {}", total);

    // hash_map! with optional trailing comma
    let map = hash_map! {
        "name" => "Alice",
        "role" => "Developer", // Trailing comma works
    };
    println!("Map: {:?}", map);
}

```

Example: Nested Repetitions

Nested repetitions allow matching multi-dimensional structures like matrices or tables.

When to use nested repetitions: - Generating multi-dimensional data structures - Processing tables or grids - Creating multiple related items with shared structure

```

//=====
// Matrix creation with nested repetitions
// Outer repetition: rows
// Inner repetition: elements in each row
//=====

macro_rules! matrix {
    ($([$($elem:expr),*]),* $(,)??) => {
        vec![
            $(
                vec![$($elem),*] // Inner: elements in row
            ),* // Outer: rows
        ]
    }
}

```

```
    };  
}
```

Example: Multiple types of repetitions

This example walks through multiple types of repetitions.

```
// Generate multiple functions from a template  
macro_rules! function_table {  
    {  
        $($fn $name:ident($($param:ident: $type:ty),*) -> $ret:ty $body:block  
            )*  
    } => {  
        $($fn $name($($param: $type),*) -> $ret $body  
            )*  
    };  
  
    fn nested_examples() {  
        // Create a 3x3 matrix  
        let mat = matrix![  
            [1, 2, 3],  
            [4, 5, 6],  
            [7, 8, 9],  
        ];  
  
        for row in &mat {  
            println!("{}: {:?}", row);  
        }  
  
        // Generate multiple functions at once  
        function_table! {  
            fn add(a: i32, b: i32) -> i32 {  
                a + b  
            }  
  
            fn multiply(x: i32, y: i32) -> i32 {  
                x * y  
            }  
        }  
  
        println!("Add: {}", add(5, 3));  
        println!("Multiply: {}", multiply(4, 7));  
    }  
}
```

Counting and Indexing

Counting elements in a macro requires recursive expansion—declarative macros don't have loops or counters.

The counting trick: Use recursion to add 1 for each element until you hit the base case (empty).

```
//=====
// Count arguments using recursive expansion
// How it works:
// count!(a b c) → 1 + count!(b c) → 1 + 1 + count!(c) → 1 + 1 + 1 + count!() → 1 + 1 + 1 + 0
// → 3
//=====

macro_rules! count {
    () => (0); // Base case: no tokens = 0
    ($head:tt $($tail:tt)*) => (1 + count!($($tail)*)); // Recursive: 1 + count(rest)
}
```

Example: Generate indexed names

This example walks through how to generate indexed names.

```
// Creates a struct with the specified field names
macro_rules! create_fields {
    ($($name:ident),*) => {
        struct GeneratedStruct {
            $($name: i32,
            )*
        }
    };
}

//=====
// Tuple indexing pattern
// Manually provides accessors for tuple elements
//=====

macro_rules! tuple_access {
    ($tuple:expr, 0) => { $tuple.0 };
    ($tuple:expr, 1) => { $tuple.1 };
    ($tuple:expr, 2) => { $tuple.2 };
    ($tuple:expr, 3) => { $tuple.3 };
}

fn counting_examples() {
    // Count tokens at compile time
    let count = count!(a b c d e);
    println!("Count: {}", count); // Prints 5

    let tuple = (1, "hello", 3.14, true);
    println!("First: {}", tuple_access!(tuple, 0));
    println!("Second: {}", tuple_access!(tuple, 1));
}
```

Example: Pattern Matching with Guards

Pattern matching in macros can match specific literals or any syntax, giving you fine control over what invocations are valid.

Pattern matching strategies: - Match specific literals to create keyword-based DSLs - Match general patterns with fragment specifiers - Combine both for flexible yet constrained syntax

Example: Match specific literals

This example walks through match specific literals.

```
macro_rules! match_literal {
    (true) => { "It's true!" };
    (false) => { "It's false!" };
    ($other:expr) => { "It's something else" };
}
```

Example: Match different types of expressions

This example walks through match different types of expressions.

```
// stringify! turns code into a string literal
macro_rules! describe_expr {
    ($e:expr) => {{
        println!("Expression: {}", stringify!($e));
        $e
    }};
}
```

Example: Complex pattern matching

This example walks through complex pattern matching.

```
macro_rules! operation {
    // Match literal operators as tokens
    ($a:expr, +, $b:expr) => {
        $a + $b
    };
    ($a:expr, -, $b:expr) => {
        $a - $b
    };
    ($a:expr, *, $b:expr) => {
        $a * $b
    };
    ($a:expr, /, $b:expr) => {
        $a / $b
    };
}
```

```

fn pattern_matching_examples() {
    println!("{}", match_literal!(true)); // "It's true!"
    println!("{}", match_literal!(42)); // "It's something else"

    // operation! creates a mini calculator language
    let result = operation!(10, +, 5);
    println!("Result: {}", result); // 15
}

```

Pattern 2: Hygiene and Scoping

Problem: Macro-generated variables collide with caller's variables—C's #define SWAP uses temp, but caller has temp variable, conflict! Macro introduces \$x but user has x—which wins?

Solution: Rust macros are hygienic—compiler renames macro-generated variables to avoid collisions. Variables from macro and caller exist in different “syntax contexts”.

Why It Matters: Prevents subtle name collision bugs—user's x won't conflict with macro's internal x. Makes macros composable: nested macro calls work without name clashes.

Use Cases: All macros (hygiene is default behavior), library macros using \$crate for paths, nested macro invocations, macros generating helper functions/structs, temporary variables in macros, composable macro systems.

Example: Hygienic Variables Pattern

Generate temporary variables without colliding with user code.

Example: Macro-generated variables are hygienic

This example walks through macro-generated variables are hygienic.

```

// The 'x' inside the macro is separate from the 'x' outside
macro_rules! hygienic_example {
    () => {
        let x = 42; // This x doesn't conflict with outer x
        println!("Inner x: {}", x);
    };
}

fn hygiene_test() {
    let x = 100;
    println!("Outer x: {}", x); // Prints 100

    hygienic_example!(); // Prints "Inner x: 42"

    println!("Outer x again: {}", x); // Still 100 (not affected by macro)
}

```

Example: Breaking Hygiene with \$name:ident

Sometimes you *want* to create or modify variables in the caller's scope. Use `ident` fragment specifiers to intentionally break hygiene.

When to break hygiene: - DSLs that create variables for the user - Macros that modify existing variables - Code generation patterns where the user expects side effects

Example: Intentionally capture variables from caller's scope

This example walks through intentionally capture variables from caller's scope.

```
// The ident fragment specifier creates a variable in the outer scope
macro_rules! set_value {
    ($var:ident = $val:expr) => {
        let $var = $val; // Creates $var in caller's scope
    };
}

macro_rules! increment {
    ($var:ident) => {
        $var += 1; // Modifies caller's variable
    };
}

fn breaking_hygiene() {
    set_value!(counter = 0);
    println!("Counter: {}", counter); // Works because we used ident

    increment!(counter);
    println!("Counter: {}", counter); // 1
}
```

Example: Macro Scope and Ordering

Unlike functions, macros must be defined *before* they're used. Macros are expanded in a single pass through the file.

Scoping rules: - Macros are visible from their definition point onward - Macros can call other macros (including themselves recursively) - `#[macro_export]` makes a macro available to other crates

Example: Macros must be defined before use

This example walks through macros must be defined before use.

```
macro_rules! early_macro {
    () => {
        println!("Defined early");
    };
}
```

```

fn can_use_early() {
    early_macro!(); // Works – macro defined above
}

//=====
// This won't compile if called before definition:
// late_macro!(); // ERROR: macro not yet defined
//=====

macro_rules! late_macro {
    () => {
        println!("Defined late");
    };
}

fn can_use_late() {
    late_macro!(); // Works – macro defined above this function
}

```

Example: Module Visibility

Macros have special visibility rules compared to other items.

Key differences: - Macros don't respect privacy boundaries by default - `#[macro_export]` exports to the crate root, not the current module - Importing macros requires special syntax in older Rust editions

Example: Macros can be exported from modules

This example walks through macros can be exported from modules.

```

mod macros {
    // #[macro_export] makes this available at crate root
    #[macro_export]
    macro_rules! public_macro {
        () => {
            println!("Public macro from module");
        };
    }

    // Non-exported macros are private to the module
    macro_rules! private_macro {
        () => {
            println!("Private macro");
        };
    }

    pub fn use_private() {
        private_macro!(); // Module can use its own private macros
    }
}

```

Example: Can use public_macro anywhere in the crate

This example walks through can use public_macro anywhere in the crate.

```
fn visibility_example() {
    public_macro!();
    // private_macro!(); // Error: not in scope
    macros::use_private(); // But can call function that uses it
}
```

Example: Context Capture

Macros can intentionally capture context to provide convenient DSLs.

The context pattern: Create a scope with predefined variables that the user's code can access.

```
//=====
// Capture context intentionally
// Provides a 'context' variable to the user's block
//=====

macro_rules! with_context {
    ($name:ident, $body:block) => {
        {
            let context = "macro context";
            let $name = context; // Bind to user's chosen name
            $body // User code can access $name
        }
    };
}

fn context_example() {
    with_context!(ctx, {
        println!("Context: {}", ctx); // ctx is provided by the macro
    });
}
```

Pattern 3: DSL Construction

Problem: Domain-specific code in Rust verbose—writing SQL queries as strings loses compile-time checking. HTML templates as strings have no type safety.

Solution: Build DSLs with macros that parse custom syntax at compile-time. sql!

Why It Matters: Compile-time type safety—SQL column typos become compile errors, not runtime. Domain code readable: select!(user.name from users where |u| u.active) vs manual iterator chains.

Use Cases: SQL query builders (type-safe at compile-time), HTML templates (yew, maud), test DSLs (assert_matches!, mock!), configuration DSLs, state machine definitions, parser combinators, JSON builders, regex DSLs, markup languages.

Example: SQL-Style DSL Pattern

Create readable query syntax that compiles to efficient iterator code.

Example: SQL-like query syntax compiled to iterator chains

This example walks through sql-like query syntax compiled to iterator chains.

```
macro_rules! select {
    // select field1, field2 from table where condition
    ($($field:ident),+ from $table:ident where $condition:expr) => {
        {
            let results = $table
                .iter()
                .filter(|row| $condition(row)) // WHERE clause
                .map(|row| {
                    ($row.$field,)+
                }) // SELECT clause
            .collect::<Vec<_>>();
            results
        }
    };
}

#[derive(Debug)]
struct User {
    id: u32,
    name: String,
    age: u32,
}

fn sql_dsl_example() {
    let users = vec![
        User { id: 1, name: "Alice".to_string(), age: 30 },
        User { id: 2, name: "Bob".to_string(), age: 25 },
        User { id: 3, name: "Carol".to_string(), age: 35 },
    ];

    // Looks like SQL, compiles to efficient iterator code
    let results = select!(name, age from users where |u: &User| u.age > 26);
    println!("Results: {:?}", results);
    // Output: [("Alice", 30), ("Carol", 35)]
}
```

Example: Configuration DSL

Create a structured configuration syntax that parses at compile time.

```
//=====
// Configuration DSL with nested structure
// section { key: value, key: value }
```

```

//=====
macro_rules! config {
    {
        $(
            $section:ident {
                $($(
                    $key:ident: $value:expr
                )),* $(,)?
            }
        )*
    } => {
        {
            use std::collections::HashMap;

            let mut config = HashMap::new();

            $($(
                let mut section = HashMap::new();
                $($(
                    section.insert(stringify!($key), $value.to_string());
                ))*
                config.insert(stringify!($section), section);
            ))*

            config
        }
    };
}

fn config_dsl_example() {
    let settings = config! {
        database {
            host: "localhost",
            port: 5432,
            name: "mydb",
        }
        server {
            host: "0.0.0.0",
            port: 8080,
            workers: 4,
        }
    };

    println!("Config: {:?}", settings);
    // Produces a nested HashMap structure
}

```

Example: HTML-like DSL

Generate HTML strings with XML-like syntax (simplified version of real templating engines).

Example: HTML-like DSL (simplified)

This example walks through html-like dsl (simplified).

```
macro_rules! html {
    // Self-closing tag: <br />
    (<$tag:ident />) => {
        format!("<{} />", stringify!($tag))
    };

    // Tag with content: <p>text</p>
    (<$tag:ident> $($content:tt)* </> $close:ident) => {
        format!("<{}>{}</{}>",
            stringify!($tag),
            html!($($content)*), // Recursively process content
            stringify!($close))
    };

    // Text content
    ($text:expr) => {
        $text.to_string()
    };

    // Multiple elements
    ($($element:tt)*) => {
        {
            let mut result = String::new();
            $($(
                result.push_str(&html!($element)));
            )*
            result
        }
    };
}

fn html_dsl_example() {
    let page = html! {
        <html>
            <body>
                <h1>"Hello, World!"</h1>
                <p>"This is a paragraph."</p>
                <br />
            </body>
        </html>
    };
    println!("{}", page);
    // Produces: <html><body><h1>Hello, World!</h1><p>This is a paragraph.</p><br /></body></html>
}
```

Example: State Machine DSL

Define state machines declaratively, compiling to efficient match-based transitions.

```
//=====
// State machine DSL
// states: [State1, State2]
// transitions: { State1 -> State2 on Event }
//=====

macro_rules! state_machine {
(
    states: [$(state:ident),* $(,)?]
    transitions: {
        $(
            $from:ident -> $to:ident on $event:ident
        )*
    }
) => {
    #[derive(Debug, Clone, Copy, PartialEq)]
    enum State {
        $($state),*
    }

    #[derive(Debug)]
    enum Event {
        $($event),*
    }

    struct StateMachine {
        current_state: State,
    }

    impl StateMachine {
        fn new(initial: State) -> Self {
            StateMachine { current_state: initial }
        }

        fn transition(&mut self, event: Event) -> Result<(), String> {
            // Pattern match on (current_state, event) to find transitions
            let new_state = match (self.current_state, event) {
                $(
                    (State::$from, Event::$event) => State::$to,
                )*
                _ => return Err(format!("Invalid transition from {:?}", self.current_state)),
            };

            self.current_state = new_state;
            Ok(())
        }

        fn current(&self) -> State {
    }
}
```

```

        self.current_state
    }
}
};

fn state_machine_example() {
    state_machine! {
        states: [Idle, Running, Paused, Stopped]
        transitions: {
            Idle -> Running on Start
            Running -> Paused on Pause
            Paused -> Running on Resume
            Running -> Stopped on Stop
            Paused -> Stopped on Stop
        }
    }

    let mut sm = StateMachine::new(State::Idle);
    println!("Initial state: {:?}", sm.current());

    sm.transition(Event::Start).unwrap();
    println!("After start: {:?}", sm.current()); // Running

    sm.transition(Event::Pause).unwrap();
    println!("After pause: {:?}", sm.current()); // Paused
}

```

Pattern 4: Code Generation Patterns

Problem: Implementing trait for all numeric types (u8, u16, u32, u64, u128, i8, ...) is 10+ identical impls. Tuple trait impls for (T1), (T1, T2), ..., (T1...T12) exponential boilerplate.

Solution: Macros generate impl blocks via repetition. Define trait impl template, list types, macro generates all.

Why It Matters: DRY—define once, generate many. Adding u256 type?

Use Cases: Trait impls for primitives (all numeric types), tuple trait impls (arity 1-12), enum From/Into/Display, struct getters/setters/builders, newtype pattern automation, format string wrappers, test case generation.

Example: Trait Implementation Generation

Implement same trait for many types without copy-paste.

Example: Note: This example uses the `paste` crate for identifier concatenation

This example walks through how to note: this example uses the `paste` crate for identifier concatenation.

```

// Add to Cargo.toml: paste = "1.0"

macro_rules! accessors {
(
    struct $name:ident {
        $($field:ident: $type:ty),* $(,)?*
    }
) => {
    struct $name {
        $($field: $type,)*
    }

    impl $name {
        // Generate getters
        $($(
            pub fn $field(&$self) -> &$type {
                &$self.$field
            }
        ),*

        paste::paste! {
            // _mut suffix for mutable accessor
            pub fn [<$field _mut>](&mut $self) -> &mut $type {
                &mut $self.$field
            }
        }

        // set_ prefix for setter
        pub fn [<set_ $field>](&mut $self, value: $type) {
            $self.$field = value;
        }
    }
}
};

accessors! {
    struct Person {
        name: String,
        age: u32,
    }
}

fn accessor_example() {
    let mut person = Person {
        name: "Alice".to_string(),
        age: 30,
    };
}

```

```
    println!("Name: {}", person.name());
    person.set_age(31);
    println!("Age: {}", person.age());
}
```

Example: Trait Implementation Generation

Generate repetitive trait implementations automatically.

Example: Generate From implementations for enum variants

This example walks through how to generate from implementations for enum variants.

```
macro_rules! impl_from {
    ($from:ty => $to:ty, $variant:ident) => {
        impl From<$from> for $to {
            fn from(value: $from) -> Self {
                <$to>::$variant(value)
            }
        };
    };
}

enum Value {
    Integer(i64),
    Float(f64),
    String(String),
    Bool(bool),
}
```

Example: Generate From impls for each variant

This example walks through how to generate from impls for each variant.

```
impl_from!(i64 => Value, Integer);
impl_from!(f64 => Value, Float);
impl_from!(String => Value, String);
impl_from!(bool => Value, Bool);

fn trait_impl_example() {
    let int_value: Value = 42i64.into();
    let string_value: Value = "hello".to_string().into();
    // Now you can use .into() to convert to Value
}
```

Example: Test Generation

Generate test cases from a compact specification.

Example: Generate multiple test functions from a template

This example walks through how to generate multiple test functions from a template.

```
macro_rules! generate_tests {
(
$fn_name:ident {
$(

    $test_name:ident: ($($input:expr),*) => $expected:expr
),* $(,)??
)
=> {
#[cfg(test)]
mod tests {
use super::*;

$(

#[test]
fn $test_name() {
let result = $fn_name($($input),* );
assert_eq!(result, $expected);
}
)*
}
};

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

Example: Generate 3 test functions

This example walks through how to generate 3 test functions.

```
generate_tests! {
add {
    test_add_positive: (2, 3) => 5,
    test_add_negative: (-2, -3) => -5,
    test_add_zero: (0, 5) => 5,
}
//=====
// Expands to:
// #[test] fn test_add_positive() { assert_eq!(add(2, 3), 5); }
// #[test] fn test_add_negative() { assert_eq!(add(-2, -3), -5); }
// #[test] fn test_add_zero() { assert_eq!(add(0, 5), 5); }
//=====
```

Example: Bitflags Pattern

Generate bitflag types with operations (similar to the `bitflags` crate).

Example: Simplified bitflags implementation

This example walks through simplified bitflags implementation.

```
macro_rules! bitflags {
(
    $(#[[$attr:meta]])*
    $vis:vis struct $name:ident: $type:ty {
        $(
            $(#[[$flag_attr:meta]])*
            const $flag:ident = $value:expr;
        )*
    }
) => {
    $(#[[$attr]])*
    #[derive(Copy, Clone, PartialEq, Eq)]
    $vis struct $name {
        bits: $type,
    }

    impl $name {
        $(
            $(#[[$flag_attr]])*
            pub const $flag: Self = Self { bits: $value };
        )*

        pub const fn empty() -> Self {
            Self { bits: 0 }
        }

        pub const fn all() -> Self {
            Self { bits: $(0 | $value)|* }
        }

        pub const fn contains(&self, other: Self) -> bool {
            (self.bits & other.bits) == other.bits
        }

        pub const fn insert(&mut self, other: Self) {
            self.bits |= other.bits;
        }

        pub const fn remove(&mut self, other: Self) {
            self.bits &= !other.bits;
        }
    }

    // Implement | operator for combining flags
}
```

```

impl std::ops::BitOr for $name {
    type Output = Self;

    fn bitor(self, rhs: Self) -> Self::Output {
        Self { bits: self.bits | rhs.bits }
    }
}

bitflags! {
    pub struct Permissions: u32 {
        const READ = 0b001;
        const WRITE = 0b010;
        const EXECUTE = 0b100;
    }
}

fn bitflags_example() {
    let perms = Permissions::READ | Permissions::WRITE;
    println!("Has read: {}", perms.contains(Permissions::READ));
    println!("Has execute: {}", perms.contains(Permissions::EXECUTE));
}

```

Pattern 5: Macro Debugging

Problem: Macro errors cryptic—"no rules expected token `ident`" doesn't show which rule failed. Expansion invisible—can't see generated code.

Solution: Use `cargo expand` to view full expansion (`cargo install cargo-expand`, then `cargo expand`). `trace_macros!(true)` logs which rules matched.

Why It Matters: Debug 10x faster by seeing actual generated code. `cargo expand` reveals what macro produces—often obvious bugs.

Use Cases: All macro development (`cargo expand` essential), debugging "no rules expected" errors, understanding library macros (`expand tokio::main!`), teaching macros, code review of macro-heavy code, performance analysis (see if macro optimal), verifying hygiene.

Example: `cargo expand` Tool

View expanded macro output to understand what code is generated.

```

# Install cargo-expand
cargo install cargo-expand

# Expand macros in your code
cargo expand

# Expand specific module

```

```
cargo expand module_name

# Expand with color output
cargo expand --color always
```

Example: Debug Printing with compile_error!

Force a compile error that shows the macro input—useful for understanding what the macro receives.

Example: Debug macro by showing its input as a compile error

This example walks through debug macro by showing its input as a compile error.

```
macro_rules! debug_macro {
    ($($tt:tt)*) => {
        compile_error!(concat!("Macro input: ", stringify!($($tt)*)));
    };
}

//=====
// This will show the exact input at compile time
// debug_macro!(some input here);
// Compile error: "Macro input: some input here"
//=====
```

Example: Tracing Macro Expansion

Print macro inputs at runtime to trace execution.

Example: Trace macro invocations at runtime

This example walks through trace macro invocations at runtime.

```
macro_rules! trace {
    ($($arg:tt)*) => {
        {
            eprintln!("Macro trace: {}", stringify!($($arg)*));
            $($arg)* // Still execute the code
        }
    };
}

fn tracing_example() {
    let x = trace!(5 + 3);
    println!("Result: {}", x);
    // Stderr: "Macro trace: 5 + 3"
    // Stdout: "Result: 8"
}
```

Example: Common Debugging Patterns

Example: 1. Echo pattern - see what the macro receives

This example walks through 1. echo pattern - see what the macro receives.

```
macro_rules! echo {
    ($($tt:tt)*) => {
        {
            println!("Macro received: {}", stringify!($($tt)*));
            $($tt)*
        }
    };
}
```

Example: 2. Type introspection

This example walks through 2. type introspection.

```
macro_rules! show_type {
    ($expr:expr) => {
        {
            fn type_of<T>(_: &T) -> &'static str {
                std::any::type_name::<T>()
            }
            let value = $expr;
            println!("Type of {}: {}", stringify!($expr), type_of(&value));
            value
        }
    };
}
```

Example: 3. Count token trees

This example walks through 3. count token trees.

```
macro_rules! count_tts {
    () => { 0 };
    ($odd:tt $($rest:tt)*) => { 1 + count_tts!($($rest)*) };
}

fn debugging_patterns() {
    echo!(println!("Hello"));

    let x = show_type!(vec![1, 2, 3]);
    // Prints: "Type of vec![1, 2, 3]: alloc::vec::Vec<i32>"

    const COUNT: usize = count_tts!(a b c d e);
```

```
    println!("Token count: {}", COUNT); // 5
}
```

Example: Error Messages with Custom Diagnostics

Provide helpful error messages when macro invocation is invalid.

Example: Validate input and provide clear error messages

This example walks through validate input and provide clear error messages.

```
macro_rules! validate_input {
    (empty) => {
        compile_error!("Input cannot be empty!");
    };
    ($($valid:tt)*) => {
        // Process valid input
        $($valid)*
    };
}
```

Example: Better error messages

This example walks through better error messages.

```
macro_rules! require_literal {
    ($lit:literal) => { $lit };
    ($other:expr) => {
        compile_error!(concat!(
            "Expected a literal value, got expression: ",
            stringify!($other)
        ));
    };
}
```

Summary

This chapter covered declarative macros (`macro_rules!`):

1. **Macro Patterns and Repetition:** Pattern matching syntax, `$(...)*` for repetition, fragment specifiers, variadic arguments
2. **Hygiene and Scoping:** Automatic variable renaming prevents collisions, `$crate` for cross-crate paths
3. **DSL Construction:** Custom syntax for domain logic, compile-time validation, zero runtime overhead
4. **Code Generation Patterns:** Generate trait impls, tuple impls, builders, eliminate boilerplate
5. **Macro Debugging:** `cargo expand`, `trace_macros!`, `rust-analyzer`, `compile_error!` debugging

Key Takeaways: - Macros operate on syntax at compile-time—generate code before type checking - Zero-cost abstraction: macro expansion compiles to optimal code - Variadic macros via `$(...)*` repetition - `vec![1, 2, 3, ..., N]` - Hygiene prevents name collisions automatically - DSLs provide type safety at compile-time vs runtime string parsing - Code generation eliminates boilerplate: one macro → 50 impls

Fragment Specifiers: - `expr`: expressions (`1 + 2, foo()`) - `ident`: identifiers (variable names) - `ty`: types (`u32, Vec`) - `stmt`: statements - `pat`: patterns (match arms) - `tt`: token tree (any token) - `item`: items (fn, struct, impl)

When to Use Macros: - Variadic functions (`println!, vec!`) - DSLs with compile-time validation - Eliminating boilerplate (trait impls for many types) - Zero-cost abstractions impossible with functions - Custom syntax that compiles to efficient Rust

When NOT to Use Macros: - Functions work—prefer functions (simpler, better errors) - Trait system suffices—traits more composable - Procedural macros better fit—more power, cleaner code - One-off code—not worth macro complexity

Debugging Tips: - `cargo expand` to see generated code - `trace_macros!(true)` to log pattern matching - Start simple, add complexity incrementally - Test macro with various inputs - `compile_error!` for debug output

Common Patterns: - Collection literals: `vec![1, 2, 3]` - Variadic `println!("{} {}", a, b)` - DSLs: `sql!` (`SELECT * FROM users`) - Trait impl generation for primitives - Builder pattern automation

Best Practices: - Document macro patterns and examples - Provide clear compile errors - Use `$crate` for library macros - Test edge cases (empty, single, many elements) - Keep macros simple—complexity hurts maintainability

Procedural Macros

This chapter covers procedural macros—full Rust functions that manipulate TokenStreams. Three types: `derive` (`##[derive(Trait)]`), `attribute` (`##[my_attr]`), `function-like` (`sql!`). Must be separate crate. Use `syn` to parse, `quote` to generate. Powers `serde`, `tokio`, `clap` ecosystem.

Procedural macros require separate crate with `proc-macro = true`:

```
# Cargo.toml for the main crate
[dependencies]
my_macros = { path = "my_macros" }

# my_macros/Cargo.toml
[package]
name = "my_macros"
version = "0.1.0"
edition = "2021"

[lib]
proc-macro = true

[dependencies]
```

```
syn = { version = "2.0", features = ["full", "extra-traits"] }
quote = "1.0"
proc-macro2 = "1.0"
```

Pattern 1: Derive Macros

Problem: Auto-implementing traits for many types tedious—manual Debug impl for 50 structs, forget to add new fields. Serde needs Serialize/Deserialize for every type—manual impls error-prone.

Solution: Write proc_macro_derive function that receives TokenStream (struct definition). Parse with syn::parse into DeriveInput (AST).

Why It Matters: Powers entire derive ecosystem—serde, clap, thiserror all use proc_macro_derive. Type-safe codegen: inspects actual struct definition.

Use Cases: serde Serialize/Deserialize (JSON/binary), Clone/Debug/PartialEq derives, builder patterns (derive_builder crate), ORM models (Diesel, SeaORM), command-line parsers (clap), error types (thiserror), validation (validator), getters/setters.

Example: Basic Derive Pattern

Create simple derive macro for custom trait.

```
//=====
// my_macros/src/lib.rs
//=====

use proc_macro::TokenStream;
use quote::quote;
use syn::{parse_macro_input, DeriveInput};

#[proc_macro_derive(HelloWorld)]
pub fn hello_world_derive(input: TokenStream) -> TokenStream {
    // Parse the input tokens into a syntax tree
    let input = parse_macro_input!(input as DeriveInput);

    // Get the name of the struct/enum
    let name = &input.ident;

    // Generate the implementation
    let expanded = quote! {
        impl HelloWorld for #name {
            fn hello_world() {
                println!("Hello, World! My name is {}", stringify!(#name));
            }
        };
    };

    TokenStream::from(expanded)
}
```

```

//=====
// main.rs - using the derive macro
//=====

trait HelloWorld {
    fn hello_world();
}

#[derive(HelloWorld)]
struct MyStruct;

#[derive(HelloWorld)]
struct AnotherStruct;

fn main() {
    MyStruct::hello_world();
    AnotherStruct::hello_world();
}

```

Example: Derive Macro with Field Access

```

use proc_macro::TokenStream;
use quote::quote;
use syn::{parse_macro_input, DeriveInput, Data, Fields};

#[proc_macro_derive(Describe)]
pub fn describe_derive(input: TokenStream) -> TokenStream {
    let input = parse_macro_input!(input as DeriveInput);
    let name = &input.ident;

    // Handle different data types
    let description = match &input.data {
        Data::Struct(data_struct) => {
            match &data_struct.fields {
                Fields::Named(fields) => {
                    let field_names: Vec<_> = fields.named
                        .iter()
                        .map(|f| &f.ident)
                        .collect();

                    quote! {
                        impl Describe for #name {
                            fn describe(&self) -> String {
                                format!(
                                    "{} {{ {} }}",
                                    stringify!(#name),
                                    vec![

                                        #(format!("{}: {:?}", stringify!(#field_names),
                                                self.#field_names))
                                         ,*
                                    ].join(", ")
                            }
                        }
                    }
                }
            }
        }
    };
}
```

```

        )
    }
}
}

Fields::Unnamed(fields) => {
    let field_indices: Vec<_> = (0..fields.unnamed.len())
        .map(syn::Index::from)
        .collect();

    quote! {
        impl Describe for #name {
            fn describe(&self) -> String {
                format!(
                    "{}({}:?)",
                    stringify!(#name),
                    #(self.#field_indices,)*
                )
            }
        }
    }
}

Fields::Unit => {
    quote! {
        impl Describe for #name {
            fn describe(&self) -> String {
                stringify!(#name).to_string()
            }
        }
    }
}

Data::Enum(_) => {
    quote! {
        impl Describe for #name {
            fn describe(&self) -> String {
                format!("{}:", self)
            }
        }
    }
}

Data::Union(_) => {
    panic!("Unions are not supported");
}

};

TokenStream::from(description)
}

```

```

//=====
// Usage

```

```

//=====
trait Describe {
    fn describe(&self) -> String;
}

#[derive(Describe)]
struct Person {
    name: String,
    age: u32,
}

#[derive(Describe)]
struct Point(i32, i32);

fn main() {
    let person = Person {
        name: "Alice".to_string(),
        age: 30,
    };
    println!("{}", person.describe());

    let point = Point(10, 20);
    println!("{}", point.describe());
}

```

Example: Derive Macro with Attributes

```

use proc_macro::TokenStream;
use quote::quote;
use syn::{parse_macro_input, DeriveInput, Data, Fields, Attribute};

#[proc_macro_derive(Builder, attributes(builder))]
pub fn builder_derive(input: TokenStream) -> TokenStream {
    let input = parse_macro_input!(input as DeriveInput);
    let name = &input.ident;
    let builder_name = syn::Ident::new(&format!("{}Builder", name), name.span());

    let fields = match &input.data {
        Data::Struct(data) => {
            match &data.fields {
                Fields::Named(fields) => &fields.named,
                _ => panic!("Builder only supports named fields"),
            }
        }
        _ => panic!("Builder only supports structs"),
    };

    // Generate builder fields
    let builder_fields = fields.iter().map(|f| {
        let name = &f.ident;
        let ty = &f.ty;
        quote! {

```

```

        #name: Option<#ty>
    }
});

// Generate setter methods
let setters = fields.iter().map(|f| {
    let name = &f.ident;
    let ty = &f.ty;
    quote! {
        pub fn #name(mut self, value: #ty) -> Self {
            self.#name = Some(value);
            self
        }
    }
});

// Generate build method
let build_fields = fields.iter().map(|f| {
    let name = &f.ident;
    quote! {
        #name: self.#name.ok_or(concat!("Field not set: ", stringify!(#name)))?
    }
});

let expanded = quote! {
    pub struct #builder_name {
        #(#builder_fields,)*
    }

    impl #builder_name {
        pub fn new() -> Self {
            Self {
                #(#name: None,)*
            }
        }

        #(#setters)*

        pub fn build(self) -> Result<#name, Box<dyn std::error::Error>> {
            Ok(#name {
                #(#build_fields,)*
            })
        }
    }

    impl #name {
        pub fn builder() -> #builder_name {
            #builder_name::new()
        }
    }
};

// Need to fix field names in builder initialization

```

```

let field_names = fields.iter().map(|f| &f.ident);

let expanded = quote! {
    pub struct #builder_name {
        #(#builder_fields,)*
    }

    impl #builder_name {
        pub fn new() -> Self {
            Self {
                #(#field_names: None,)*
            }
        }

        #(#setters)*

        pub fn build(self) -> Result<#name, Box<dyn std::error::Error>> {
            Ok(#name {
                #(#build_fields,)*
            })
        }
    }

    impl #name {
        pub fn builder() -> #builder_name {
            #builder_name::new()
        }
    }
};

TokenStream::from(expanded)
}

```

```

//=====
// Usage
//=====

#[derive(Builder)]
struct User {
    username: String,
    email: String,
    age: u32,
}

fn main() {
    let user = User::builder()
        .username("alice".to_string())
        .email("alice@example.com".to_string())
        .age(30)
        .build()
        .unwrap();
}

```

Example: Derive Macro for Enums

```
use proc_macro::TokenStream;
use quote::quote;
use syn::{parse_macro_input, DeriveInput, Data, DataEnum};

#[proc_macro_derive(EnumIter)]
pub fn enum_iter_derive(input: TokenStream) -> TokenStream {
    let input = parse_macro_input!(input as DeriveInput);
    let name = &input.ident;

    let variants = match &input.data {
        Data::Enum(DataEnum { variants, .. }) => variants,
        _ => panic!("EnumIter only works on enums"),
    };

    // Only support unit variants for simplicity
    let variant_idents: Vec<_> = variants
        .iter()
        .map(|variant| &variant.ident)
        .collect();

    let expanded = quote! {
        impl #name {
            pub fn variants() -> &'static [&#name] {
                &[
                    #( #name::#variant_idents,)*
                ]
            }

            pub fn variant_names() -> &'static [&'static str] {
                &[
                    #(stringify!(#variant_idents),)*
                ]
            }
        }
    };

    TokenStream::from(expanded)
}
```

```
//=====
// Usage
//=====
#[derive(Debug, Copy, Clone, EnumIter)]
enum Color {
    Red,
    Green,
    Blue,
}
```

```

fn main() {
    for color in Color::variants() {
        println!("{}: {}", color);
    }

    for name in Color::variant_names() {
        println!("{}: {}", name);
    }
}

```

Pattern 2: Attribute Macros

Problem: Need to modify/wrap functions without changing their code—add timing, logging, tracing. Want aspect-oriented programming (cross-cutting concerns).

Solution: Write proc_macro_attribute function receiving two TokenStreams: attribute args and item being annotated. Parse item (function, struct, impl) with syn.

Why It Matters: Enables tokio::main and tokio::test (essential for async). tracing::instrument adds automatic logging (non-invasive observability).

Use Cases: tokio::main/test (async runtime), tracing::instrument (automatic logging), web framework routes (actix, axum, rocket), test fixtures, memoization/caching, error handling wrappers, performance timing, authorization checks.

Example: Attribute Macro Pattern

Wrap function with timing/logging without modifying its body.

```

use proc_macro::TokenStream;
use quote::quote;
use syn::{parse_macro_input, ItemFn};

#[proc_macro_attribute]
pub fn timing(_attr: TokenStream, item: TokenStream) -> TokenStream {
    let input = parse_macro_input!(item as ItemFn);

    let fn_name = &input.sig.ident;
    let fn_block = &input.block;
    let fn_sig = &input.sig;
    let fn_vis = &input.vis;
    let fn_attrs = &input.attrs;

    let expanded = quote! {
        #(fn_attrs)*
        fn #fn_vis #fn_sig {
            let start = std::time::Instant::now();
            let result = (#fn_block)();
            let elapsed = start.elapsed();
            println!("Function {} took {:?}", stringify!(#fn_name), elapsed);
            result
        }
    };
    quote!(#fn_vis #fn_name(#expanded));
}

```

```

        }

    TokenStream::from(expanded)
}

//=====
// Usage
//=====
#[timing]
fn slow_function() {
    std::thread::sleep(std::time::Duration::from_millis(100));
}

fn main() {
    slow_function(); // Prints: Function slow_function took ~100ms
}

```

Example: Attribute Macro with Parameters

```

use proc_macro::TokenStream;
use quote::quote;
use syn::{parse_macro_input, ItemFn, parse::{Parse, ParseStream}, LitStr};

struct LogArgs {
    prefix: LitStr,
}

impl Parse for LogArgs {
    fn parse(input: ParseStream) -> syn::Result<Self> {
        Ok(LogArgs {
            prefix: input.parse()?,
        })
    }
}

#[proc_macro_attribute]
pub fn log(attr: TokenStream, item: TokenStream) -> TokenStream {
    let args = parse_macro_input!(attr as LogArgs);
    let input = parse_macro_input!(item as ItemFn);

    let prefix = args.prefix.value();
    let fn_name = &input.sig.ident;
    let fn_block = &input.block;
    let fn_sig = &input.sig;
    let fn_vis = &input.vis;

    let expanded = quote! {
        #fn_vis #fn_sig {
            println!("{} Entering {}", #prefix, stringify!(#fn_name));
            let result = (#fn_block)();
        }
    };
    TokenStream::from(expanded)
}

```

```

        println!("{} Exiting {}", #prefix, stringify!(#fn_name));
    result
}
};

TokenStream::from(expanded)
}

```

```

//=====
// Usage
//=====
#[log("[INFO]")]
fn my_function() {
    println!("Doing work...");
}

fn main() {
    my_function();
    // Output:
    // [INFO] Entering my_function
    // Doing work...
    // [INFO] Exiting my_function
}

```

Example: Method Attribute Macro

```

use proc_macro::TokenStream;
use quote::quote;
use syn::{parse_macro_input, ImplItemFn};

#[proc_macro_attribute]
pub fn cache(_attr: TokenStream, item: TokenStream) -> TokenStream {
    let input = parse_macro_input!(item as ImplItemFn);

    let fn_name = &input.sig.ident;
    let fn_block = &input.block;
    let fn_sig = &input.sig;
    let fn_vis = &input.vis;

    let cache_name = syn::Ident::new(
        &format!("{}_cache", fn_name),
        fn_name.span()
    );

    let expanded = quote! {
        #fn_vis #fn_sig {
            use std::sync::Mutex;
            use std::collections::HashMap;

            lazy_static::lazy_static! {

```

```

        static ref #cache_name: Mutex<HashMap<String, _*> =
    Mutex::new(HashMap::new());
}

// Simple cache implementation
// In real code, you'd want to hash the arguments properly
#fn_block
}
};

TokenStream::from(expanded)
}

```

Example: Struct Attribute Macro

```

use proc_macro::TokenStream;
use quote::quote;
use syn::{parse_macro_input, ItemStruct};

#[proc_macro_attribute]
pub fn add_debug_info(_attr: TokenStream, item: TokenStream) -> TokenStream {
    let mut input = parse_macro_input!(item as ItemStruct);

    // Add a field to the struct
    if let syn::Fields::Named(ref mut fields) = input.fields {
        fields.named.push(
            syn::Field::parse_named(
                .parse2(quote! { _debug_info: String })
                .unwrap()
            );
    }

    let name = &input.ident;

    let expanded = quote! {
        #input

        impl #name {
            pub fn debug_info(&self) -> &str {
                &self._debug_info
            }
        }
    };

    TokenStream::from(expanded)
}

```

```

//=====
// Usage
//=====
#[add_debug_info]

```

```

struct MyStruct {
    value: i32,
}

fn main() {
    let s = MyStruct {
        value: 42,
        _debug_info: "Created at startup".to_string(),
    };
    println!("{}", s.debug_info());
}

```

Pattern 3: Function-like Macros

Problem: Need custom syntax beyond derive/attribute—sql! needs full SQL parsing with compile-time checking.

Solution: Write proc_macro function receiving TokenStream. Implement syn::parse::Parse for custom syntax structs.

Why It Matters: Enables sophisticated DSLs—sqlx::query! validates SQL against database schema at compile-time.

Use Cases: SQL DSLs (sqlx, diesel—type-checked queries), HTML templates (yew html!, maud), configuration DSLs, query builders, compile-time regex validation, format string checking, JSON literals with validation, embedded language DSLs.

Example: Function-like Macro Pattern

Create SQL-like macro with custom parsing.

```

use proc_macro::TokenStream;
use quote::quote;
use syn::{parse::Parse, ParseStream}, Expr, Token};

struct SqlQuery {
    query: Expr,
}

impl Parse for SqlQuery {
    fn parse(input: ParseStream) -> syn::Result<Self> {
        Ok(SqlQuery {
            query: input.parse()?,
        })
    }
}

#[proc_macro]
pub fn sql(input: TokenStream) -> TokenStream {
    let SqlQuery { query } = parse_macro_input!(input as SqlQuery);
}

```

```

let expanded = quote! {
    {
        let query_str = #query;
        // Validate SQL at compile time
        println!("Executing SQL: {}", query_str);
        query_str
    }
};

TokenStream::from(expanded)
}

```

```

//=====
// Usage
//=====
fn main() {
    let query = sql!("SELECT * FROM users WHERE age > 18");
    println!("Query: {}", query);
}

```

Example: Complex Function-like Macro

```

use proc_macro::TokenStream;
use quote::quote;
use syn::{
    parse::{Parse, ParseStream},
    punctuated::Punctuated,
    Ident, Token, Expr,
};

struct HashMapLiteral {
    entries: Punctuated<KeyValue, Token![,]>,
}

struct KeyValue {
    key: Expr,
    _arrow: Token![=>],
    value: Expr,
}

impl Parse for KeyValue {
    fn parse(input: ParseStream) -> syn::Result<Self> {
        Ok(KeyValue {
            key: input.parse()?,
            _arrow: input.parse()?,
            value: input.parse()?,
        })
    }
}

impl Parse for HashMapLiteral {

```

```

fn parse(input: ParseStream) -> syn::Result<Self> {
    Ok(HashMapLiteral {
        entries: input.parse_terminated(KeyValue::parse, Token![], []))?,
    })
}

#[proc_macro]
pub fn hashmap(input: TokenStream) -> TokenStream {
    let HashMapLiteral { entries } = parse_macro_input!(input as HashMapLiteral);

    let insertions = entries.iter().map(|kv| {
        let key = &kv.key;
        let value = &kv.value;
        quote! {
            map.insert(#key, #value);
        }
    });
}

let expanded = quote! {
{
    let mut map = std::collections::HashMap::new();
    #(insertions)*
    map
}
};

TokenStream::from(expanded)
}

```

```

//=====
// Usage
//=====

fn main() {
    let map = hashmap! {
        "name" => "Alice",
        "role" => "Developer",
        "level" => "Senior"
    };

    println!("{}:?}", map);
}

```

Example: DSL Function-like Macro

```

use proc_macro::TokenStream;
use quote::quote;
use syn::{
    parse::{Parse, ParseStream},
    Ident, Token, Expr, braced,
};

```

```

struct RouteDefinition {
    method: Ident,
    _comma: Token![],,
    path: Expr,
    _arrow: Token![=>],
    handler: Expr,
}

impl Parse for RouteDefinition {
    fn parse(input: ParseStream) -> syn::Result<Self> {
        Ok(RouteDefinition {
            method: input.parse()?,
            _comma: input.parse()?,
            path: input.parse()?,
            _arrow: input.parse()?,
            handler: input.parse()?,
        })
    }
}

struct Routes {
    routes: Vec<RouteDefinition>,
}

impl Parse for Routes {
    fn parse(input: ParseStream) -> syn::Result<Self> {
        let mut routes = Vec::new();

        while !input.is_empty() {
            routes.push(input.parse()?;

            if !input.is_empty() {
                input.parse::<Token![;]>()?;
            }
        }

        Ok(Routes { routes })
    }
}

#[proc_macro]
pub fn routes(input: TokenStream) -> TokenStream {
    let Routes { routes } = parse_macro_input!(input as Routes);

    let route_matches = routes.iter().map(|route| {
        let method = &route.method;
        let path = &route.path;
        let handler = &route.handler;

        quote! {
            if request.method == stringify!(#method) && request.path == #path {
                return #handler(request);
            }
        }
    });
}

```

```

        }
    });

let expanded = quote! {
    |request: Request| -> Response {
        #(#route_matches)*
        Response::not_found()
    }
};

TokenStream::from(expanded)
}

```

Pattern 4: Token Stream Manipulation

Problem: Need fine-grained token control beyond syn's abstractions. syn doesn't support your custom syntax.

Solution: Work with TokenStream and TokenTree directly. Iterate tokens, match on TokenTree variants (Group, Ident, Punct, Literal).

Why It Matters: Maximum flexibility—handle any syntax, even invalid Rust. Performance: direct token manipulation faster than full syn parse.

Use Cases: Custom DSL parsers (syntax not in syn), performance-critical macros (skip syn overhead), advanced derive scenarios, token filters/transformers, macro composition, debugging token structure, embedded language parsers.

Example: Direct TokenStream Pattern

Manipulate tokens directly without syn parsing.

```

use proc_macro::{TokenStream, TokenTree, Delimiter, Group, Punct, Spacing};
use quote::quote;

#[proc_macro]
pub fn reverse_tokens(input: TokenStream) -> TokenStream {
    let tokens: Vec<TokenTree> = input.into_iter().collect();
    let reversed: TokenStream = tokens.into_iter().rev().collect();
    reversed
}

```

Example: Token Inspection

```

use proc_macro::TokenStream;

#[proc_macro]
pub fn inspect_tokens(input: TokenStream) -> TokenStream {
    eprintln!("Token count: {}", input.clone().into_iter().count());
}

```

```

for token in input.clone() {
    match token {
        proc_macro::TokenTree::Group(g) => {
            eprintln!("Group: delimiter={:{}?}{:?}", g.delimiter());
        }
        proc_macro::TokenTree::Ident(i) => {
            eprintln!("Ident: {}", i);
        }
        proc_macro::TokenTree::Punct(p) => {
            eprintln!("Punct: {} (spacing={:{}?}{:?}", p.as_char(), p.spacing());
        }
        proc_macro::TokenTree::Literal(l) => {
            eprintln!("Literal: {}", l);
        }
    }
}

input
}

```

Example: Building TokenStream from Scratch

```

use proc_macro::{TokenStream, TokenTree, Ident, Span, Literal};

#[proc_macro]
pub fn build_struct(_input: TokenStream) -> TokenStream {
    let mut tokens = TokenStream::new();

    // struct
    tokens.extend(Some(TokenTree::Ident(Ident::new("struct", Span::call_site()))));

    // MyStruct
    tokens.extend(Some(TokenTree::Ident(Ident::new("MyStruct", Span::call_site()))));

    // { ... }
    let mut fields = TokenStream::new();
    fields.extend(Some(TokenTree::Ident(Ident::new("value", Span::call_site()))));
    fields.extend(Some(TokenTree::Punct(proc_macro::Punct::new(':', proc_macro::Spacing::Alone))));
    fields.extend(Some(TokenTree::Ident(Ident::new("i32", Span::call_site()))));

    tokens.extend(Some(TokenTree::Group(
        proc_macro::Group::new(proc_macro::Delimiter::Brace, fields)
    )));

    tokens
}

```

Example: Span Manipulation

```
use proc_macro::TokenStream;
use quote::quote;
use syn::{parse_macro_input, Ident, parse::{Parse, ParseStream}};

struct SpanExample {
    name: Ident,
}

impl Parse for SpanExample {
    fn parse(input: ParseStream) -> syn::Result<Self> {
        Ok(SpanExample {
            name: input.parse()?,
        })
    }
}

#[proc_macro]
pub fn with_span(input: TokenStream) -> TokenStream {
    let SpanExample { name } = parse_macro_input!(input as SpanExample);

    // Get the span of the input identifier
    let span = name.span();

    // Create a new identifier with the same span
    let prefixed = Ident::new(&format!("prefixed_{}", name), span);

    let expanded = quote! {
        let #prefixed = stringify!(#name);
    };

    TokenStream::from(expanded)
}
```

Pattern 5: Macro Helper Crates (syn, quote)

Problem: Parsing TokenStream manually is tedious—80% of macro is parsing boilerplate. Generating code with string concatenation error-prone (unbalanced braces, hygiene bugs).

Solution: Use syn to parse TokenStream into AST (DeriveInput, ItemFn, etc.). quote!

Why It Matters: syn eliminates 90% of parsing code—DeriveInput has all struct info. quote!

Use Cases: All proc macros (syn+quote ubiquitous), derive macros (syn::DeriveInput), attribute parsing (darling), testing macros (proc_macro2), custom parsing (syn::parse::Parse), code generation (quote!), field iteration, type inspection.

Example: syn Parsing Pattern

Parse Rust syntax from TokenStream into typed AST.

```

use syn::{
    parse::{Parse, ParseStream},
    Ident, Token, Type, Visibility,
    braced, punctuated::Punctuated,
};

//=====
// Parse a struct definition
//=====

struct StructDef {
    vis: Visibility,
    _struct_token: Token![_struct],
    name: Ident,
    fields: Punctuated<Field, Token![_,>>,
}

struct Field {
    name: Ident,
    _colon: Token![_:],
    ty: Type,
}

impl Parse for Field {
    fn parse(input: ParseStream) -> syn::Result<Self> {
        Ok(Field {
            name: input.parse()?,
            _colon: input.parse()?,
            ty: input.parse()?,
        })
    }
}

impl Parse for StructDef {
    fn parse(input: ParseStream) -> syn::Result<Self> {
        let content;
        Ok(StructDef {
            vis: input.parse()?,
            _struct_token: input.parse()?,
            name: input.parse()?,
            fields: {
                braced!(content in input);
                content.parse_terminated(Field::parse, Token![_,>])?
            },
        })
    }
}

```

Example: Using quote for Code Generation

```

use quote::quote;
use syn::Ident;

```

```

fn generate_getter(struct_name: &Ident, field_name: &Ident, field_type: &syn::Type) ->
    proc_macro2::TokenStream {
    quote! {
        impl #struct_name {
            pub fn #field_name(&self) -> &#field_type {
                &self.#field_name
            }
        }
    }
}

fn generate_multiple_methods(name: &str, count: usize) -> proc_macro2::TokenStream {
    let methods = (0..count).map(|i| {
        let method_name = format_ident!("method_{}", i);
        quote! {
            pub fn #method_name(&self) -> i32 {
                #i
            }
        }
    });
    quote! {
        impl MyStruct {
            #(#methods)*
        }
    }
}

```

Example: Advanced syn Features

```

use syn::{
    Attribute, Expr, ExprLit, Lit, Meta, MetaNameValue,
};

//=====
// Parse custom attributes
//=====

fn parse_custom_attribute(attr: &Attribute) -> Option<String> {
    if attr.path().is_ident("doc") {
        if let Meta::NameValue(MetaNameValue {
            value: Expr::Lit(ExprLit {
                lit: Lit::Str(s),
                ..
            }),
            ..
        }) = &attr.meta {
            return Some(s.value());
        }
    }
    None
}

```

```
//=====
// Extract documentation comments
//=====

fn get_doc_comments(attrs: &[Attribute]) -> Vec<String> {
    attrs.iter()
        .filter_map(parse_custom_attribute)
        .collect()
}
```

Example: Combining syn and quote

```
use proc_macro::TokenStream;
use quote::quote;
use syn::{parse_macro_input, DeriveInput, Data, Fields};

#[proc_macro_derive(GetterSetter)]
pub fn getter_setter_derive(input: TokenStream) -> TokenStream {
    let input = parse_macro_input!(input as DeriveInput);
    let name = &input.ident;

    let fields = match &input.data {
        Data::Struct(data) => {
            match &data.fields {
                Fields::Named(fields) => &fields.named,
                _ => panic!("Only named fields supported"),
            }
        }
        _ => panic!("Only structs supported"),
    };

    // Generate getters
    let getters = fields.iter().map(|f| {
        let field_name = &f.ident;
        let field_type = &f.ty;
        quote! {
            pub fn #field_name(&self) -> &#field_type {
                &self.#field_name
            }
        }
    });

    // Generate setters
    let setters = fields.iter().map(|f| {
        let field_name = &f.ident;
        let field_type = &f.ty;
        let setter_name = quote::format_ident!("set_{}", field_name.as_ref().unwrap());
        quote! {
            pub fn #setter_name(&mut self, value: #field_type) {
                self.#field_name = value;
            }
        }
    });
}
```

```

});
```

```

let expanded = quote! {
    impl #name {
        #(#getters)*
        #(#setters)*
    }
};

TokenStream::from(expanded)
}

```

Example: Error Handling in Procedural Macros

```

use proc_macro::TokenStream;
use quote::quote;
use syn::{parse_macro_input, DeriveInput, Error};

#[proc_macro_derive(Validated)]
pub fn validated_derive(input: TokenStream) -> TokenStream {
    let input = parse_macro_input!(input as DeriveInput);

    // Validate that it's a struct
    match &input.data {
        syn::Data::Struct(_) => {}
        _ => {
            return Error::new_spanned(
                &input,
                "Validated can only be derived for structs"
            )
            .to_compile_error()
            .into();
        }
    }
}

let name = &input.ident;

let expanded = quote! {
    impl Validated for #name {
        fn validate(&self) -> Result<(), String> {
            Ok(())
        }
    }
};

TokenStream::from(expanded)
}

```

Example: Custom Parse Implementation

```
use syn::{
    parse::{Parse, ParseStream},
    Ident, Token, LitStr,
    Result,
};

enum ConfigValue {
    String(LitStr),
    Nested(Vec<ConfigItem>),
}

struct ConfigItem {
    key: Ident,
    _eq: Token![=],
    value: ConfigValue,
}

impl Parse for ConfigValue {
    fn parse(input: ParseStream) -> Result<Self> {
        if input.peek(syn::token::Brace) {
            let content;
            syn::braced!(content in input);
            let mut items = Vec::new();
            while !content.is_empty() {
                items.push(content.parse()?);
            }
            Ok(ConfigValue::Nested(items))
        } else {
            Ok(ConfigValue::String(input.parse()?))
        }
    }
}

impl Parse for ConfigItem {
    fn parse(input: ParseStream) -> Result<Self> {
        Ok(ConfigItem {
            key: input.parse()?,
            _eq: input.parse()?,
            value: input.parse()?,
        })
    }
}
```

Example: Using proc_macro2

```
use proc_macro2::{TokenStream, Span, Ident};
use quote::quote;

fn generate_with_proc_macro2() -> TokenStream {
```

```

let struct_name = Ident::new("GeneratedStruct", Span::call_site());
let field_name = Ident::new("value", Span::call_site());

quote! {
    struct #struct_name {
        #field_name: i32,
    }

    impl #struct_name {
        fn new(#field_name: i32) -> Self {
            Self { #field_name }
        }
    }
}

//=====
// Convert between proc_macro and proc_macro2
//=====

use proc_macro::TokenStream as TokenStream1;
use proc_macro2::TokenStream as TokenStream2;

fn convert_token_streams(input: TokenStream1) -> TokenStream1 {
    let tokens: TokenStream2 = input.into();
    // Process with proc_macro2
    let result = quote! { #tokens };
    result.into()
}

```

Example: Complete Example: JSON Serialization Macro

```

use proc_macro::TokenStream;
use quote::quote;
use syn::{parse_macro_input, DeriveInput, Data, Fields};

#[proc_macro_derive(ToJson)]
pub fn to_json_derive(input: TokenStream) -> TokenStream {
    let input = parse_macro_input!(input as DeriveInput);
    let name = &input.ident;

    let json_impl = match &input.data {
        Data::Struct(data) => {
            match &data.fields {
                Fields::Named(fields) => {
                    let field_serializations = fields.named.iter().map(|f| {
                        let field_name = &f.ident;
                        let field_name_str = field_name.as_ref().unwrap().to_string();
                        quote! {
                            format!("\"{}\": {:?}", #field_name_str, self.#field_name)
                        }
                    });

```

```

        quote! {
            impl #name {
                pub fn to_json(&self) -> String {
                    format!(
                        "{{ {} }}",
                        vec![#(field_serializations),*].join(", ")
                )
            }
        }
    }
    _ => panic!("Only named fields supported"),
}
_ => panic!("Only structs supported"),
};

TokenStream::from(json_impl)
}

```

```

//=====
// Usage
//=====
#[derive(ToJson)]
struct Person {
    name: String,
    age: u32,
}

fn main() {
    let person = Person {
        name: "Alice".to_string(),
        age: 30,
    };
    println!("{}", person.to_json());
}

```

Example: Testing Procedural Macros

```

//=====
// tests/integration_tests.rs
//=====
#[test]
fn test_derive_macro() {
    #[derive(HelloWorld)]
    struct TestStruct;

    TestStruct::hello_world();
}

#[test]

```

```

fn test_attribute_macro() {
    #[timing]
    fn test_fn() -> i32 {
        42
    }

    assert_eq!(test_fn(), 42);
}

#[test]
fn test_function_macro() {
    let map = hashmap! {
        "key1" => "value1",
        "key2" => "value2"
    };

    assert_eq!(map.get("key1"), Some(&"value1"));
}

```

Summary

This chapter covered procedural macros:

- Derive Macros:** #[derive(Trait)] auto-implements traits, powers serde/clap, parses struct with syn
- Attribute Macros:** #[my_attr] wraps/modifies items, enables tokio::main, tracing::instrument, web routes
- Function-like Macros:** my_macro!() custom syntax DSLs, SQL/HTML builders, full parsing control
- Token Stream Manipulation:** Direct token access, maximum flexibility, performance-critical macros
- Helper Crates:** syn for parsing, quote! for codegen, darling for attributes, proc_macro2 for testing

Key Takeaways: - Proc macros are Rust functions that manipulate TokenStreams at compile-time - Must be in separate crate with proc-macro = true - Three types: derive, attribute, function-like - syn parses tokens → AST, quote! generates code - Powers entire ecosystem: serde, tokio, clap, diesel, actix-web

Macro Types Comparison: - **Derive:** Auto-implement traits (#[derive(Serialize)]) - **Attribute:** Modify items (#[tokio::main], #[get("/users")]) - **Function-like:** Custom syntax (sql!(), html!())

Essential Crates: - **syn:** Parse TokenStream to AST (DeriveInput, ItemFn, etc.) - **quote:** Generate code with interpolation (quote! { impl #name }) - **proc_macro2:** Testing (works outside proc-macro crate) - **darling:** Declarative attribute parsing (#[my_attr(skip)])

Common Patterns: - Derive for trait impls: #[derive(Serialize, Clone)] - Attribute for wrappers: #[tokio::main], #[tracing::instrument] - Function-like for DSLs: sql!("SELECT *"), html! { }

When to Use Each Type: - Use **derive** when: Auto-implementing traits (serde, builder) - Use **attribute** when: Wrapping/modifying items (async, logging) - Use **function-like** when: Custom DSL syntax (SQL, HTML)

Best Practices: - Always use syn+quote (don't parse manually) - Provide clear compile errors - Test with proc_macro2 (unit tests) - Document with examples - Handle edge cases (empty structs, unit enums) - Use darling for attribute parsing

Common Use Cases: - serde: Serialize/Deserialize derives - tokio: async runtime setup (#[tokio::main]) - clap: Command-line parsing derives - tracing: Automatic instrumentation - Web frameworks: Route registration - ORMs: Model derives (Diesel, SeaORM) - Builders: Auto-generate builder pattern

Debugging Tips: - cargo expand shows macro output - proc_macro2 enables unit testing - eprintln! in macro for debug output - Test incrementally—start simple - Check syn docs for AST structure

FFI & C Interop

This chapter covers FFI (Foreign Function Interface)—calling C from Rust and vice versa. Rust's safety model differs from C: must use unsafe, manage memory carefully, handle strings/callbacks/errors correctly. Essential for integrating existing C libraries and system APIs.

Pattern 1: C ABI Compatibility

Problem: Rust and C have incompatible ABIs—Rust optimizes struct layouts (reorders fields), uses different calling conventions, doesn't guarantee ABI stability between versions. Passing Rust struct to C corrupts data (wrong offsets).

Solution: Use extern "C" to declare C functions and mark Rust functions for C. Use #[repr(C)] to force C-compatible struct layout—fields in declaration order, C alignment rules.

Why It Matters: Enables using decades of C libraries—SQLite, OpenSSL, zlib, libcurl. OS APIs (POSIX, Win32) are C.

Use Cases: System calls (open, read, write), database drivers (SQLite, Postgres FFI), graphics APIs (OpenGL, Vulkan, DirectX), audio/video codecs (ffmpeg), crypto libraries (OpenSSL, libsodium), compression (zlib, lz4), embedded HALs, legacy C code integration.

Example: C ABI Fundamentals

Understand ABI compatibility requirements between Rust and C.

```
//=====
// This struct uses Rust's default representation
// The compiler might reorder fields, add padding
//=====
struct RustStruct {
    a: u8,
    b: u32,
```

```

    c: u16,
}

//=====
// This struct is guaranteed to match C's layout
// Fields appear in memory in the order declared
//=====

#[repr(C)]
struct CCompatibleStruct {
    a: u8,      // 1 byte, followed by 3 bytes of padding
    b: u32,     // 4 bytes, aligned to 4-byte boundary
    c: u16,     // 2 bytes
}

//=====
// Verify the sizes
//=====

fn demonstrate_repr() {
    println!("RustStruct size: {}", std::mem::size_of::<RustStruct>());
    println!("CCompatibleStruct size: {}", std::mem::size_of::<CCompatibleStruct>());

    // Both are likely 12 bytes, but only CCompatibleStruct guarantees
    // the exact layout C expects
}

```

The difference becomes critical when passing data to C libraries. If the layouts don't match exactly, you'll get corrupted data or crashes.

Example: Calling C Functions from Rust

To call a C function from Rust, you first declare it with `extern "C"`. This declaration acts as a promise to the compiler about what exists in the linked C library:

```

//=====
// Declare external C functions
//=====

extern "C" {
    // C: int abs(int n);
    fn abs(n: i32) -> i32;

    // C: void *malloc(size_t size);
    fn malloc(size: usize) -> *mut std::ffi::c_void;

    // C: void free(void *ptr);
    fn free(ptr: *mut std::ffi::c_void);

    // C: double sqrt(double x);
    fn sqrt(x: f64) -> f64;
}

fn use_c_functions() {
    unsafe {

```

```

// All C function calls are unsafe
// The Rust compiler can't verify C code's safety guarantees
let result = abs(-42);
println!("abs(-42) = {}", result);

let root = sqrt(16.0);
println!("sqrt(16.0) = {}", root);
}
}

```

Notice that we must wrap C function calls in `unsafe` blocks. This is Rust’s way of saying: “Beyond this point, the safety guarantees are up to you.” C doesn’t have Rust’s borrow checker, null-safety, or bounds checking, so the compiler can’t verify that the C code won’t cause undefined behavior.

Example: Exposing Rust Functions to C

Sometimes you need to go the other direction—exposing Rust functions so C code can call them. This is common when embedding Rust in existing C applications or creating C-compatible libraries:

```

//=====
// A Rust function callable from C
//=====

#[no_mangle] // Don't change the function name during compilation
pub extern "C" fn rust_add(a: i32, b: i32) -> i32 {
    a + b
}

//=====
// Prevent name mangling for easier linking
//=====

#[no_mangle]
pub extern "C" fn rust_compute_average(values: *const f64, count: usize) -> f64 {
    // Safety: Caller must ensure:
    // 1. values is valid for count elements
    // 2. values is properly aligned
    // 3. values is not mutated during this call
    unsafe {
        if values.is_null() || count == 0 {
            return 0.0;
        }

        let slice = std::slice::from_raw_parts(values, count);
        let sum: f64 = slice.iter().sum();
        sum / count as f64
    }
}

```

The `#[no_mangle]` attribute is crucial. Normally, Rust “mangles” function names to encode type information, which helps with overloading but makes the names unreadable. C expects simple, predictable names like `rust_add`, not something like `_ZN4rust8rust_add17h3b3c3d3e3f3g3hE`.

Example: Type Mapping Between C and Rust

Understanding how C types map to Rust types is essential for correct FFI:

```
use std::os::raw::{c_char, c_int, c_long, c_ulong, c_void};

// C type          -> Rust equivalent
// char            -> c_char (usually i8 or u8, platform-dependent)
// int             -> c_int (usually i32)
// long            -> c_long (i32 on 32-bit, i64 on 64-bit)
// unsigned long   -> c_ulong
// void*           -> *mut c_void or *const c_void
// bool            -> bool (C99) or c_int (C89)
// size_t           -> usize
// float           -> f32
// double          -> f64

// Example: working with C types
extern "C" {
    fn process_data(
        buffer: *mut c_char,
        size: usize,
        flags: c_int,
    ) -> c_long;
}

fn use_c_types() {
    let mut buffer = vec![0u8; 100];

    unsafe {
        let result = process_data(
            buffer.as_mut_ptr() as *mut c_char,
            buffer.len(),
            42,
        );

        if result >= 0 {
            println!("Processed {} bytes", result);
        }
    }
}
```

The `std::os::raw` module provides platform-independent type aliases that match C's types on the current platform. Always use these rather than assuming `int` is `i32`—on some platforms, it might not be.

Example: Struct Padding and Alignment

C compilers insert padding between struct fields to satisfy alignment requirements. Rust does the same with `#[repr(C)]`, but understanding this is crucial for debugging layout issues:

```

#[repr(C)]
struct Padded {
    a: u8,      // 1 byte
    // 3 bytes padding
    b: u32,     // 4 bytes (must be 4-byte aligned)
    c: u8,      // 1 byte
    // 3 bytes padding (to make struct size multiple of largest alignment)
}

#[repr(C, packed)]
struct NoPadding {
    a: u8,      // 1 byte
    b: u32,     // 4 bytes (no padding, may be misaligned!)
    c: u8,      // 1 byte
}

fn examine_layout() {
    println!("Padded size: {}, align: {}", 
        std::mem::size_of::<Padded>(),
        std::mem::align_of::<Padded>()
    );
    // Padded size: 12, align: 4

    println!("NoPadding size: {}, align: {}", 
        std::mem::size_of::<NoPadding>(),
        std::mem::align_of::<NoPadding>()
    );
    // NoPadding size: 6, align: 1
}

```

The `packed` attribute removes padding, which can save space but may cause performance issues or crashes on architectures that don't support misaligned access. Only use it when interfacing with C code that explicitly uses packed structs.

Pattern 2: String Conversions

Problem: Rust `&str` (UTF-8, length-prefixed, can contain NUL) incompatible with C `*char` (null-terminated, no encoding, stops at `\0`). Ownership unclear: who frees the string?

Solution: Use `CString` to create owned null-terminated C strings. Use `CStr` to borrow C strings.

Why It Matters: Strings are most error-prone FFI aspect—wrong conversion causes use-after-free, buffer overrun, null pointer deref, memory leaks. FFI boundaries in production: file paths, error messages, configuration.

Use Cases: Passing filenames to C (`fopen`, `stat`), error messages from C (`strerror`), configuration strings, logging to C libraries, command-line arguments, environment variables, C string parsing, text processing across FFI.

Example: CString/CStr Pattern

Problem: Convert between Rust strings and C null-terminated strings safely.

```
use std::ffi::{CString, CStr};
use std::os::raw::c_char;

//=====
// Rust to C
//=====

fn rust_string_to_c() {
    let rust_string = "Hello, C!";

    // Create a CString (allocates, adds null terminator)
    let c_string = CString::new(rust_string).expect("CString::new failed");

    // Get a pointer suitable for C
    let c_ptr: *const c_char = c_string.as_ptr();

    unsafe {
        // Pass to C function
        some_c_function(c_ptr);
    }

    // c_string is dropped here, freeing the memory
}

//=====
// C to Rust
//=====

unsafe fn c_string_to_rust(c_ptr: *const c_char) -> String {
    // Safety: Caller must ensure c_ptr is valid and null-terminated

    if c_ptr.is_null() {
        return String::new();
    }

    // Create a CStr (borrows the C string)
    let c_str = CStr::from_ptr(c_ptr);

    // Convert to Rust String
    // to_string_lossy replaces invalid UTF-8 with ⚡
    c_str.to_string_lossy().into_owned()
}

extern "C" {
    fn some_c_function(s: *const c_char);
}
```

The key insight here is that `CString::new()` can fail. Why? Because Rust strings can contain null bytes, but C strings can't (null terminates the string). If you try to create a `CString` from a Rust string containing `\0`, you'll get an error:

```
use std::ffi::CString;

fn demonstrate_null_bytes() {
    let with_null = "Hello\0World";

    match CString::new(with_null) {
        Ok(_) => println!("Success"),
        Err(e) => println!("Failed: {}", e),
        // Output: Failed: nul byte found in provided data at position: 5
    }

    // If you need to handle this, strip null bytes first:
    let without_null: String = with_null.chars()
        .filter(|&c| c != '\0')
        .collect();

    let c_string = CString::new(without_null).unwrap();
}
```

Example: `OsString` and `OsStr`

While `CString` handles null-terminated strings, operating system paths present a different challenge. File paths aren't always valid UTF-8—Windows uses UTF-16, and Unix allows arbitrary bytes (except null). This is where `OsString` and `OsStr` come in:

```
use std::ffi::{OsString, OsStr};
use std::path::{Path, PathBuf};

fn working_with_os_strings() {
    // Creating an OsString
    let os_string = OsString::from("my_file.txt");

    // Converting between Path and OsStr
    let path = Path::new("/home/user/document.txt");
    let os_str: &OsStr = path.as_os_str();

    // Attempting UTF-8 conversion (may fail on Windows or Unix)
    match os_str.to_str() {
        Some(s) => println!("Valid UTF-8: {}", s),
        None => println!("Path contains invalid UTF-8"),
    }

    // Lossy conversion (replaces invalid UTF-8 with \0)
    let string = os_str.to_string_lossy();
    println!("Path (lossy): {}", string);
}
```

`OsString` is particularly important when writing cross-platform code:

```
use std::ffi::{OsString, OsStr};

#[cfg(windows)]
fn platform_specific_path() -> OsString {
    use std::os::windows::ffi::OsStringExt;

    // Windows uses UTF-16
    let wide: Vec<u16> = vec![0x0048, 0x0065, 0x006C, 0x006C, 0x006F]; // "Hello"
    OsString::from_wide(&wide)
}

#[cfg(unix)]
fn platform_specific_path() -> OsString {
    use std::os::unix::ffi::OsStringExt;

    // Unix allows arbitrary bytes
    let bytes: Vec<u8> = vec![0x48, 0x65, 0x6C, 0x6C, 0x6F]; // "Hello"
    OsString::from_vec(bytes)
}
```

Example: String Ownership Across FFI

One of the most common bugs in FFI code is getting ownership wrong. Consider these scenarios:

```
use std::ffi::{CString, CStr};
use std::os::raw::c_char;

//=====
// CORRECT: Rust owns the string
//=====

#[no_mangle]
pub extern "C" fn rustCreatesString() -> *mut c_char {
    let s = CString::new("Hello from Rust").unwrap();

    // Transfer ownership to C
    // C must call rust_free_string() when done
    s.into_raw()
}

#[no_mangle]
pub unsafe extern "C" fn rustFreeString(s: *mut c_char) {
    if !s.is_null() {
        // Take ownership back and drop
        let _ = CString::from_raw(s);
    }
}

//=====
// CORRECT: C owns the string
```

```

//=====
#[no_mangle]
pub unsafe extern "C" fn rust_uses_c_string(s: *const c_char) {
    if s.is_null() {
        return;
    }

    // Borrow the string, don't take ownership
    let c_str = CStr::from_ptr(s);
    println!("C string: {}", c_str.to_string_lossy());

    // s is still valid here; C will free it
}

//=====
// WRONG: This leaks memory!
//=====

#[no_mangle]
pub extern "C" fn leaked_string() -> *const c_char {
    let s = CString::new("This will leak").unwrap();
    s.as_ptr() // s is dropped here, but pointer escapes!
}

```

The golden rule: **whoever allocates the memory must free it**. If Rust allocates, Rust must free (even if C holds the pointer temporarily). If C allocates, C must free.

Example: Practical Example: File Path Handling

Here's a complete example showing proper string handling across FFI boundaries:

```

use std::ffi::{CString, CStr, OsStr};
use std::os::raw::c_char;
use std::path::Path;

//=====
// C API for opening a file
//=====

extern "C" {
    fn fopen(filename: *const c_char, mode: *const c_char) -> *mut std::ffi::c_void;
    fn fclose(file: *mut std::ffi::c_void) -> i32;
}

/// Safe wrapper around C's fopen
fn open_file(path: &Path, mode: &str) -> Option<*mut std::ffi::c_void> {
    // Convert Path to CString
    // This can fail if the path contains null bytes
    let path_str = path.to_str()?;
    let c_path = CString::new(path_str).ok()?;

    let c_mode = CString::new(mode).ok()?;

```

```

let file = fopen(c_path.as_ptr(), c_mode.as_ptr());

if file.is_null() {
    None
} else {
    Some(file)
}
}

fn file_handling_example() {
    let path = Path::new("test.txt");

    if let Some(file) = open_file(path, "r") {
        println!("File opened successfully");

        unsafe {
            fclose(file);
        }
    } else {
        println!("Failed to open file");
    }
}

```

This pattern—wrapping unsafe C calls in safe Rust functions—is the key to good FFI code. You contain the unsafety in small, well-tested functions and expose safe APIs to the rest of your code.

Pattern 3: Callback Patterns

Problem: C callbacks expect function pointers (extern “C” fn), but Rust closures capture environment (not C-compatible). Need stateful callbacks—closure with captured state called from C.

Solution: Use extern “C” fn for stateless callbacks. For stateful: Box::into_raw() passes closure as void*, trampoline function extracts state.

Why It Matters: Async APIs need callbacks—C libraries can’t block Rust futures. Signal handlers require callbacks.

Use Cases: Event loops (GUI toolkits—GTK, Qt), async I/O libraries, signal handlers (SIGINT, SIGTERM), qsort comparators, thread spawn callbacks (pthread_create), plugin systems, C library hooks, timer callbacks.

Example: Function Pointer Callback Pattern

Problem: Register Rust function as C callback for simple stateless cases.

```

use std::os::raw::c_int;

//=====
// C library API
//=====
```

```

extern "C" {
    fn register_callback(callback: extern "C" fn(c_int));
    fn trigger_callbacks();
}

//=====
// Our callback function
//=====

extern "C" fn my_callback(value: c_int) {
    println!("Callback received: {}", value);
}

fn simple_callback_example() {
    unsafe {
        register_callback(my_callback);
        trigger_callbacks();
    }
}

```

This works because `my_callback` is a simple function pointer. No state, no closures, just a function. But what if you need state?

Example: Callbacks with State (User Data Pattern)

Most C libraries support a “user data” or “context” pointer—an opaque `void*` that gets passed back to your callback:

```

use std::os::raw::{c_int, c_void};

extern "C" {
    fn register_callback_with_data(
        callback: extern "C" fn(*mut c_void, c_int),
        user_data: *mut c_void,
    );
    fn trigger_callbacks_with_data();
}

struct CallbackState {
    count: i32,
    name: String,
}

extern "C" fn stateful_callback(user_data: *mut c_void, value: c_int) {
    unsafe {
        // Cast the void pointer back to our state
        let state = &mut *(user_data as *mut CallbackState);

        state.count += 1;
        println!("{}: Callback #{} received value {}",
            state.name, state.count, value);
    }
}

```

```

fn stateful_callback_example() {
    let mut state = CallbackState {
        count: 0,
        name: "MyCallback".to_string(),
    };

    unsafe {
        register_callback_with_data(
            stateful_callback,
            &mut state as *mut _ as *mut c_void,
        );
    }

    trigger_callbacks_with_data();
}

println!("Final count: {}", state.count);
}

```

This pattern is powerful but dangerous. You must ensure: 1. The state outlives the callback registration 2. The state isn't moved (moving would invalidate the pointer) 3. No other code mutates the state during callbacks (data race!)

Example: Thread-Safe Callbacks

What if callbacks can be triggered from different threads? Now you need thread-safe state:

```

use std::sync::{Arc, Mutex};
use std::os::raw::{c_int, c_void};

extern "C" {
    fn register_threadsafe_callback(
        callback: extern "C" fn(*mut c_void, c_int),
        user_data: *mut c_void,
    );
}

struct ThreadSafeState {
    data: Arc<Mutex<Vec<i32>>>,
}

extern "C" fn threadsafe_callback(user_data: *mut c_void, value: c_int) {
    unsafe {
        let state = &*(user_data as *const ThreadSafeState);

        // Lock the mutex before accessing shared data
        if let Ok(mut data) = state.data.lock() {
            data.push(value);
        }
    }
}

```

```

fn threadsafe_example() {
    let state = ThreadSafeState {
        data: Arc::new(Mutex::new(Vec::new())),
    };

    unsafe {
        register_threadsafe_callback(
            threadsafe_callback,
            &state as *const _ as *mut c_void,
        );
    }

    // Keep state alive for the duration of the program
    std::mem::forget(state);
}

```

Notice the `std::mem::forget` at the end. This is necessary because we're giving C a pointer to Rust-owned data. If `state` were dropped, the pointer would become invalid. `forget` leaks the memory, which is usually wrong, but here it's intentional—the callback might be called at any time in the future.

Example: Cleaning Up Callbacks

Of course, leaking memory isn't ideal. Better C libraries provide a way to unregister callbacks:

```

use std::os::raw::{c_int, c_void};

extern "C" {
    fn register_callback_managed(
        callback: extern "C" fn(*mut c_void, c_int),
        user_data: *mut c_void,
    ) -> c_int; // Returns handle

    fn unregister_callback(handle: c_int);
}

struct ManagedCallback {
    handle: c_int,
    state: Box<CallbackState>,
}

impl ManagedCallback {
    fn new(name: String) -> Self {
        let state = Box::new(CallbackState {
            count: 0,
            name,
        });

        unsafe {
            let handle = register_callback_managed(
                stateful_callback,
                &*state as *const _ as *mut c_void,
            );

```

```

        ManagedCallback { handle, state }
    }
}

impl Drop for ManagedCallback {
    fn drop(&mut self) {
        unsafe {
            unregister_callback(self.handle);
        }
        // state is automatically dropped after this
    }
}

struct CallbackState {
    count: i32,
    name: String,
}

extern "C" fn stateful_callback(user_data: *mut c_void, value: c_int) {
    // Same as before
    unsafe {
        let state = &mut *(user_data as *mut CallbackState);
        state.count += 1;
    }
}

```

Now we have RAII-style cleanup. When `ManagedCallback` is dropped, it automatically unregisters the callback and cleans up the state. This is much safer.

Example: Closures as Callbacks (Advanced)

Can we use Rust closures as C callbacks? It's tricky, but possible for non-capturing closures:

```

use std::os::raw::c_int;

extern "C" {
    fn register_simple_callback(callback: extern "C" fn(c_int));
}

fn closure_callback_example() {
    // Non-capturing closure can be coerced to function pointer
    let callback: extern "C" fn(c_int) = {
        extern "C" fn wrapper(value: c_int) {
            println!("Closure callback: {}", value);
        }
        wrapper
    };

    unsafe {
        register_simple_callback(callback);
    }
}

```

```
    }  
}
```

But capturing closures don't work directly—they have state, and C function pointers can't carry state. You need the user data pattern for that.

Pattern 4: Error Handling Across FFI

Problem: Rust `Result<T, E>` and panic incompatible with C's `errno`/return codes/NULL. Panics unwinding into C are undefined behavior (crashes, memory corruption).

Solution: Convert `Result` to `i32` return codes (`0` = success, `<0` = error). Set `errno` for POSIX compatibility.

Why It Matters: Panicking into C is undefined behavior—absolutely must prevent. Production FFI must never panic.

Use Cases: All FFI functions (must handle errors properly), library wrappers (translate `Result` to `errno`), system calls, C callbacks (cannot panic), plugin interfaces, language bindings (Python, Ruby calling Rust), error propagation.

Example: Error Translation Pattern

Convert Rust `Result` to C error codes and prevent panics.

```
use std::os::raw::{c_int, c_char};  
use std::ffi::CString;  
use std::io;  
  
extern "C" {  
    // Returns -1 on error, sets errno  
    fn c_function(path: *const c_char) -> c_int;  
  
    // Gets the errno value  
    fn __errno_location() -> *mut c_int;  
}  
  
fn errno() -> i32 {  
    unsafe { *__errno_location() }  
}  
  
fn safe_c_function(path: &str) -> io::Result<i32> {  
    // Convert string  
    let c_path = CString::new(path)  
        .map_err(|_| io::Error::new(io::ErrorKind::InvalidInput, "Invalid path"))?  
  
    // Call C function  
    let result = unsafe { c_function(c_path.as_ptr()) };  
  
    if result == -1 {  
        // Error occurred, check errno
```

```

        let err = errno();
        Err(io::Error::from_raw_os_error(err))
    } else {
        Ok(result)
    }
}

fn error_handling_example() {
    match safe_c_function("/some/path") {
        Ok(result) => println!("Success: {}", result),
        Err(e) => eprintln!("Error: {}", e),
    }
}

```

This pattern—checking the return value and converting `errno` to a Rust error—is common when wrapping POSIX functions.

Example: Exposing Rust Errors to C

Going the other direction is trickier. C can't understand `Result` or panics. You need to design a C-compatible error API:

```

use std::os::raw::{c_int, c_char};
use std::ffi::CString;

//=====
// Error codes for C
//=====

const SUCCESS: c_int = 0;
const ERROR_NULL_POINTER: c_int = -1;
const ERROR_INVALID_INPUT: c_int = -2;
const ERROR_COMPUTATION_FAILED: c_int = -3;

#[no_mangle]
pub extern "C" fn rust_compute(
    input: *const c_char,
    output: *mut f64,
) -> c_int {
    // Check for null pointers
    if input.is_null() || output.is_null() {
        return ERROR_NULL_POINTER;
    }

    unsafe {
        // Convert C string to Rust
        let c_str = std::ffi::CStr::from_ptr(input);
        let rust_str = match c_str.to_str() {
            Ok(s) => s,
            Err(_) => return ERROR_INVALID_INPUT,
        };

        // Do computation
    }
}

```

```

    let result = match compute_internal(rust_str) {
        Ok(r) => r,
        Err(_) => return ERROR_COMPUTATION_FAILED,
    };

    // Write result
    *output = result;
    SUCCESS
}

fn compute_internal(input: &str) -> Result<f64, &'static str> {
    input.parse().map_err(|_| "Invalid number")
}

//=====
// Helper for getting error messages
//=====

#[no_mangle]
pub extern "C" fn rust_error_message(error_code: c_int) -> *const c_char {
    let msg = match error_code {
        ERROR_NULL_POINTER => "Null pointer provided\0",
        ERROR_INVALID_INPUT => "Invalid input\0",
        ERROR_COMPUTATION_FAILED => "Computation failed\0",
        _ => "Unknown error\0",
    };

    msg.as_ptr() as *const c_char
}

```

This provides a C-friendly error API: integer codes with a function to get error messages. The pattern of using out-parameters (the `output` pointer) is also very C-friendly.

Example: Panic Safety

One critical consideration: **Rust must never panic across FFI boundaries**. If Rust code panics while called from C, the behavior is undefined. Always catch panics:

```

use std::panic;
use std::os::raw::c_int;

#[no_mangle]
pub extern "C" fn safe_rust_function(value: c_int) -> c_int {
    // Catch any panics
    let result = panic::catch_unwind(|| {
        // Code that might panic
        risky_computation(value)
    });

    match result {
        Ok(value) => value,
        Err(_) => {

```

```

        eprintln!("Rust function panicked!");
        -1 // Error code
    }
}

fn risky_computation(value: c_int) -> c_int {
    if value < 0 {
        panic!("Negative values not allowed");
    }
    value * 2
}

```

`catch_unwind` prevents panics from crossing the FFI boundary, giving you a chance to log the error and return a safe error code.

Example: Error Context and Debugging

For complex FFI code, maintaining error context is important:

```

use std::os::raw::c_int;
use std::sync::Mutex;
use std::cell::RefCell;

//=====
// Thread-local error storage
//=====
thread_local! {
    static LAST_ERROR: RefCell<Option<String>> = RefCell::new(None);
}

fn set_last_error(error: String) {
    LAST_ERROR.with(|e| {
        *e.borrow_mut() = Some(error);
    });
}

#[no_mangle]
pub extern "C" fn rust_function_with_error(value: c_int) -> c_int {
    if value < 0 {
        set_last_error(format!("Invalid value: {}", value));
        return -1;
    }

    // ... computation ...

    0
}

#[no_mangle]
pub extern "C" fn rust_get_last_error() -> *const std::os::raw::c_char {
    LAST_ERROR.with(|e| {

```

```

match &*e.borrow() {
    Some(err) => {
        // Note: This is simplified. In production, you'd want to
        // manage the string's lifetime more carefully
        let c_str = std::ffi::CString::new(err.as_str()).unwrap();
        c_str.into_raw()
    }
    None => std::ptr::null(),
}
}

#[no_mangle]
pub unsafe extern "C" fn rust_clear_last_error() {
    LAST_ERROR.with(|e| {
        *e.borrow_mut() = None;
    });
}

```

This provides detailed error messages while maintaining a C-compatible API.

Pattern 5: bindgen Patterns

Problem: Manually writing FFI bindings tedious—100+ functions, 50+ structs, error-prone. Struct layouts wrong (misaligned fields).

Solution: bindgen auto-generates Rust bindings from C headers. Parses with libclang (actual compiler).

Why It Matters: Eliminates 90% manual FFI work—hundreds of lines auto-generated. Keeps bindings in sync with C headers automatically.

Use Cases: Wrapping C libraries (SQLite, libcurl, OpenSSL, SDL), system API bindings (libc, Win32), graphics APIs (OpenGL, Vulkan), audio/video libraries (ffmpeg), embedded HALs, automatic binding generation, maintaining C library wrappers.

Example: bindgen Setup Pattern

Problem: Automatically generate Rust bindings from C headers at build time.

```

# Cargo.toml
[build-dependencies]
bindgen = "0.69"

```

Then create a build script:

```

//=====
// build.rs
//=====
use std::env;
use std::path::PathBuf;

```

```

fn main() {
    // Tell cargo to invalidate the built crate whenever the wrapper changes
    println!("cargo:rerun-if-changed=wrapper.h");

    // Link to C library
    println!("cargo:rustc-link-lib=mylib");

    // Generate bindings
    let bindings = bindgen::Builder::default()
        .header("wrapper.h")
        .parse_callbacks(Box::new(bindgen::CargoCallbacks::new()))
        .generate()
        .expect("Unable to generate bindings");

    // Write bindings to $OUT_DIR/bindings.rs
    let out_path = PathBuf::from(env::var("OUT_DIR").unwrap());
    bindings
        .write_to_file(out_path.join("bindings.rs"))
        .expect("Couldn't write bindings!");
}

```

Create a wrapper header that includes the library you want to bind:

```

// wrapper.h
#include <mylib.h>

```

Then use the generated bindings in your Rust code:

```

//=====
// lib.rs or main.rs
//=====

#![allow(non_upper_case_globals)]
#![allow(non_camel_case_types)]
#![allow(non_snake_case)]

include!(concat!(env!("OUT_DIR"), "/bindings.rs"));

fn use_bindings() {
    unsafe {
        // Use the generated bindings
        let result = some_c_function(42);
        println!("Result: {}", result);
    }
}

```

The `allow` attributes silence warnings about the generated code not following Rust naming conventions.

Example: Configuring bindgen

bindgen is highly configurable. You can control what gets generated:

```

//=====
// build.rs
//=====
use bindgen;

fn main() {
    let bindings = bindgen::Builder::default()
        .header("wrapper.h")

        // Allowlist: only generate bindings for these
        .allowlist_function("my_.*") // Regex: functions starting with my_
        .allowlist_type("MyStruct")
        .allowlist_var("MY_CONSTANT")

        // Blocklist: don't generate bindings for these
        .blocklist_function("internal_.*")

        // Generate comments from C documentation
        .generate_comments(true)

        // Use core instead of std (for no_std environments)
        .use_core()

        // Derive additional traits
        .derive_default(true)
        .derive_debug(true)
        .derive_eq(true)

        // Handle C++ (if needed)
        .clang_arg("-x")
        .clang_arg("c++")

        // Custom type mappings
        .raw_line("use std::os::raw::c_char;")

        .generate()
        .expect("Unable to generate bindings");

    // Write bindings...
}

```

This level of control is essential for large libraries where you only need a subset of functionality.

Example: Wrapping Generated Bindings

Generated bindings are completely unsafe. Best practice is to wrap them in safe Rust APIs:

```

//=====
// Generated by bindgen
//=====
mod ffi {

```

```
    include!(concat!(env!("OUT_DIR"), "/bindings.rs"));
}

//=====
// Safe wrapper
//=====

pub struct Database {
    handle: *mut ffi::db_t,
}

impl Database {
    pub fn open(path: &str) -> Result<Self, String> {
        let c_path = std::ffi::CString::new(path)
            .map_err(|_| "Invalid path")?;

        let handle = unsafe {
            ffi::db_open(c_path.as_ptr())
        };

        if handle.is_null() {
            Err("Failed to open database".to_string())
        } else {
            Ok(Database { handle })
        }
    }

    pub fn query(&self, sql: &str) -> Result<Vec<String>, String> {
        let c_sql = std::ffi::CString::new(sql)
            .map_err(|_| "Invalid SQL")?;

        unsafe {
            let result = ffi::db_query(self.handle, c_sql.as_ptr());

            if result.is_null() {
                return Err("Query failed".to_string());
            }

            // Process results...
            // (This is simplified; real code would parse the result)

            ffi::db_free_result(result);
            Ok(vec![])
        }
    }

    impl Drop for Database {
        fn drop(&mut self) {
            unsafe {
                ffi::db_close(self.handle);
            }
        }
    }
}
```

```
//=====
// Safe to send between threads if C library is thread-safe
//=====
unsafe impl Send for Database {}
```

Now users of your library can work with `Database` using safe, idiomatic Rust, while all the FFI complexity is hidden.

Example: Handling Opaque Types

C libraries often use opaque pointers—pointers to types whose definition isn't in the header:

```
// In C header
typedef struct db_connection db_connection_t;
db_connection_t* db_connect(const char* url);
void db_disconnect(db_connection_t* conn);
```

bindgen generates an opaque type:

```
//=====
// Generated
//=====
#[repr(C)]
pub struct db_connection {
    _unused: [u8; 0],
}
```

You can't construct this directly (it has zero size!), but you can hold pointers to it. This is actually perfect—it prevents you from incorrectly constructing instances:

```
pub struct Connection {
    ptr: *mut ffi::db_connection,
}

impl Connection {
    pub fn connect(url: &str) -> Result<Self, String> {
        let c_url = std::ffi::CString::new(url)
            .map_err(|_| "Invalid URL")?;

        let ptr = unsafe { ffi::db_connect(c_url.as_ptr()) };

        if ptr.is_null() {
            Err("Connection failed".to_string())
        } else {
            Ok(Connection { ptr })
        }
    }
}
```

```

impl Drop for Connection {
    fn drop(&mut self) {
        unsafe {
            ffi::db_disconnect(self.ptr);
        }
    }
}

```

Example: Function Pointers and Callbacks with bindgen

bindgen handles C function pointers automatically:

```

// C header
typedef void (*callback_t)(int value, void* user_data);
void register_callback(callback_t callback, void* user_data);

```

Generated Rust:

```

//=====
// Generated
//=====
pub type callback_t = Option<unsafe extern "C" fn(value: ::std::os::raw::c_int, user_data:
    *mut ::std::os::raw::c_void)>;

extern "C" {
    pub fn register_callback(callback: callback_t, user_data: *mut ::std::os::raw::c_void);
}

```

You can then implement callbacks using the patterns we discussed earlier:

```

extern "C" fn my_callback(value: i32, user_data: *mut std::ffi::c_void) {
    unsafe {
        let state = &mut *(user_data as *mut MyState);
        state.handle_event(value);
    }
}

struct MyState {
    count: i32,
}

impl MyState {
    fn handle_event(&mut self, value: i32) {
        self.count += value;
        println!("Event: {}, Total: {}", value, self.count);
    }
}

```

Summary

This chapter covered FFI (Foreign Function Interface) patterns for C interop:

1. **C ABI Compatibility**: extern "C", #[repr(C)], raw pointers, calling conventions
2. **String Conversions**: CString/CStr, null-termination, UTF-8 validation, ownership
3. **Callback Patterns**: Function pointers, stateful callbacks, panic boundaries, context pointers
4. **Error Handling**: Result to errno, catch_unwind(), return codes, preventing UB
5. **bindgen Patterns**: Auto-generate bindings, build.rs integration, allowlist/blocklist

Key Takeaways: - FFI bridges Rust safety with C's unsafety—careful handling required - All FFI calls are unsafe—must verify C library guarantees - extern "C" for C calling convention, #[repr(C)] for C struct layout - Strings most error-prone: CString/CStr for null-terminated conversion - Panics across FFI are undefined behavior—always catch_unwind() - bindgen automates 90% of FFI work for large C libraries

Critical Safety Rules: - **Never panic into C**: Use catch_unwind() at FFI boundary - **Validate all pointers**: Check for NULL before dereferencing - **Manage lifetimes**: Ensure pointers outlive their use - **Match layouts exactly**: Use #[repr(C)] and verify with tests - **Handle errors**: Translate Result to C conventions properly

Common Patterns: - Wrap unsafe FFI in safe Rust APIs - Use CString::new() → as_ptr() for Rust → C strings - Use CStr::from_ptr() → to_str() for C → Rust strings - Box::into_raw() for transferring ownership to C - catch_unwind() in callbacks to prevent unwinding into C

When to Use FFI: - Existing C libraries too valuable to rewrite (SQLite, OpenSSL) - System APIs (OS interfaces are C) - Incremental migration (rewrite gradually) - Performance-critical code already optimized in C - Ecosystem integration (Python/Ruby call Rust via C FFI)

Best Practices: - Encapsulate unsafety in small, well-tested functions - Document all invariants and safety requirements - Use bindgen for large C APIs - Test FFI boundaries extensively - Validate all data crossing boundary - Never trust C to uphold Rust's invariants

Common Pitfalls: - Panicking across FFI boundary (undefined behavior) - String lifetime issues (use-after-free) - Struct layout mismatches (wrong #[repr]) - Forgetting to validate UTF-8 from C - Double-free or memory leaks - Integer overflow in size conversions

Tools: - **bindgen**: Auto-generate bindings from C headers - **cbindgen**: Generate C headers from Rust (reverse) - **cargo expand**: View generated code - **Miri**: Detect undefined behavior (limited FFI support) - **Valgrind**: Memory errors at runtime

Network Programming

This chapter covers network programming patterns—TCP/UDP for low-level protocols, HTTP client/server for web services, WebSocket for real-time bidirectional communication. Rust's async ecosystem enables high-performance, concurrent network applications with safety guarantees.

Pattern 1: TCP Server/Client Patterns

Problem: Need reliable bidirectional communication between client and server. Simple echo server handles one client at time (blocks).

Solution: Use tokio's async TcpListener and TcpStream. tokio::spawn() spawns task per connection.

Why It Matters: TCP foundation for HTTP, SSH, FTP, databases—essential protocol. Async I/O solves C10K problem (10K concurrent connections).

Use Cases: Chat servers (persistent connections per user), game servers (player connections), database protocols (Postgres, Redis), custom TCP protocols, proxy servers, load balancers, monitoring agents, message brokers, SSH servers.

Example: Async TCP Server Pattern

Handle thousands of concurrent TCP connections efficiently.

```
use std::net::{TcpListener, TcpStream};
use std::io::{Read, Write, BufReader, BufRead};
use std::thread;

/// A basic echo server that handles one client at a time
/// This is synchronous and will block on each operation
fn simple_echo_server() -> std::io::Result<()> {
    // Bind to localhost on port 8080
    // The "0.0.0.0:8080" address means "listen on all network interfaces"
    let listener = TcpListener::bind("127.0.0.1:8080")?;
    println!("Server listening on port 8080");

    // Accept connections in a loop
    for stream in listener.incoming() {
        match stream {
            Ok(stream) => {
                println!("New connection from: {}", stream.peer_addr()?);
                handle_client(stream)?;
            }
            Err(e) => {
                eprintln!("Connection failed: {}", e);
            }
        }
    }
    Ok(())
}

fn handle_client(mut stream: TcpStream) -> std::io::Result<()> {
    let mut buffer = [0; 1024];

    loop {
        // Read data from the client
        let bytes_read = stream.read(&mut buffer)?;
```

```

    // If bytes_read is 0, the client has disconnected
    if bytes_read == 0 {
        println!("Client disconnected");
        break;
    }

    // Echo the data back to the client
    stream.write_all(&buffer[..bytes_read])?;
    stream.flush()?;
}

Ok(())
}

```

This simple server has a critical limitation: it can only handle one client at a time. While one client is connected, other clients attempting to connect will have to wait. For production use, we need concurrent handling.

Example: Multi-threaded TCP Server

To handle multiple clients simultaneously, we can spawn a thread for each connection. This allows the server to accept new connections while existing connections are being serviced:

```

use std::net::{TcpListener, TcpStream};
use std::io::{Read, Write};
use std::thread;

/// Multi-threaded server that spawns a thread per client
/// This scales better but can exhaust system resources with many connections
fn multithreaded_server() -> std::io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:8080")?;
    println!("Multi-threaded server listening on port 8080");

    for stream in listener.incoming() {
        match stream {
            Ok(stream) => {
                // Spawn a new thread for each connection
                thread::spawn(move || {
                    if let Err(e) = handle_client_thread(stream) {
                        eprintln!("Error handling client: {}", e);
                    }
                });
            }
            Err(e) => {
                eprintln!("Connection failed: {}", e);
            }
        }
    }

    Ok(())
}

```

```

fn handle_client_thread(mut stream: TcpStream) -> std::io::Result<()> {
    let addr = stream.peer_addr()?;
    println!("Thread handling client: {}", addr);

    let mut buffer = [0; 1024];

    loop {
        let bytes_read = stream.read(&mut buffer)?;

        if bytes_read == 0 {
            println!("Client {} disconnected", addr);
            break;
        }

        // Echo back
        stream.write_all(&buffer[..bytes_read])?;
    }

    Ok(())
}

```

While this works well for moderate numbers of clients, each thread consumes system resources. For high-concurrency scenarios, async I/O is more efficient.

Example: Async TCP Server with Tokio

Modern Rust networking typically uses async/await with Tokio. This allows handling thousands of concurrent connections efficiently because tasks are much lighter than threads:

```

use tokio::net::{TcpListener, TcpStream};
use tokio::io::{AsyncReadExt, AsyncWriteExt};

/// Async echo server using Tokio
/// Can handle thousands of concurrent connections efficiently
#[tokio::main]
async fn async_echo_server() -> tokio::io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:8080").await?;
    println!("Async server listening on port 8080");

    loop {
        // Accept is async – other tasks can run while waiting
        let (socket, addr) = listener.accept().await?;
        println!("New connection from {}", addr);

        // Spawn an async task for this connection
        // Tasks are much cheaper than threads
        tokio::spawn(async move {
            if let Err(e) = handle_connection(socket).await {
                eprintln!("Error handling {}: {}", addr, e);
            }
        });
    }
}

```

```

    }

}

async fn handle_connection(mut socket: TcpStream) -> tokio::io::Result<()> {
    let mut buffer = vec![0; 1024];

    loop {
        // Async read - yields to other tasks while waiting for data
        let n = socket.read(&mut buffer).await?;

        if n == 0 {
            // Connection closed
            return Ok(());
        }

        // Echo the data back
        socket.write_all(&buffer[..n]).await?;
    }
}

```

The key advantage here is that `await` points yield control to the runtime, allowing other tasks to make progress. This means one slow client doesn't block others.

Example: Line-based Protocol Server

Many protocols are line-based (like HTTP, SMTP, FTP). Here's a server that reads line by line, which is more realistic than raw bytes:

```

use tokio::net::{TcpListener, TcpStream};
use tokio::io::{AsyncBufReadExt, AsyncWriteExt, BufReader};

/// A line-based protocol server
/// Useful for protocols like SMTP, FTP, or custom text protocols
async fn line_based_server() -> tokio::io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:8080").await?;
    println!("Line-based server listening on port 8080");

    loop {
        let (socket, addr) = listener.accept().await?;
        println!("Connection from {}", addr);

        tokio::spawn(async move {
            // BufReader provides efficient buffered reading
            let (reader, mut writer) = socket.into_split();
            let mut reader = BufReader::new(reader);
            let mut line = String::new();

            loop {
                line.clear();

                // Read until we get a newline
                match reader.read_line(&mut line).await {

```

```

        Ok(0) => {
            // EOF - connection closed
            println!("Client {} disconnected", addr);
            break;
        }
        Ok(_) => {
            // Process the line
            let response = process_command(&line);

            // Send response
            if let Err(e) = writer.write_all(response.as_bytes()).await {
                eprintln!("Failed to write to {}: {}", addr, e);
                break;
            }
        }
        Err(e) => {
            eprintln!("Error reading from {}: {}", addr, e);
            break;
        }
    }
}

fn process_command(line: &str) -> String {
    let line = line.trim();

    match line.to_uppercase().as_str() {
        "HELLO" => "WORLD\n".to_string(),
        "QUIT" => "BYE\n".to_string(),
        _ => format!("ECHO: {}\n", line),
    }
}

```

This pattern is extremely common in network programming. By reading line-by-line, you can implement simple command-response protocols efficiently.

Example: TCP Client

Now let's look at the client side. A TCP client connects to a server and exchanges data:

```

use tokio::net::TcpStream;
use tokio::io::{AsyncReadExt, AsyncWriteExt};

/// Connect to a server and exchange messages
async fn tcp_client_example() -> tokio::io::Result<()> {
    // Connect to the server
    let mut stream = TcpStream::connect("127.0.0.1:8080").await?;
    println!("Connected to server");

    // Send a message

```

```

let message = "Hello, Server!\n";
stream.write_all(message.as_bytes()).await?;

// Read the response
let mut buffer = vec![0; 1024];
let n = stream.read(&mut buffer).await?;

println!("Received: {}", String::from_utf8_lossy(&buffer[..n]));

Ok(())
}

```

For more complex clients, you might want to separate reading and writing:

```

use tokio::net::TcpStream;
use tokio::io::{AsyncBufReadExt, AsyncWriteExt, BufReader};

/// Interactive client that can send and receive concurrently
async fn interactive_client() -> tokio::io::Result<()> {
    let stream = TcpStream::connect("127.0.0.1:8080").await?;
    let (reader, mut writer) = stream.into_split();
    let mut reader = BufReader::new(reader);

    // Spawn a task to handle incoming messages
    let read_handle = tokio::spawn(async move {
        let mut line = String::new();
        loop {
            line.clear();
            match reader.read_line(&mut line).await {
                Ok(0) => break, // Connection closed
                Ok(_) => print!("Server: {}", line),
                Err(e) => {
                    eprintln!("Read error: {}", e);
                    break;
                }
            }
        }
    });
}

// Main task handles user input and sending
let write_handle = tokio::spawn(async move {
    use tokio::io::{stdin, AsyncBufReadExt, BufReader};

    let stdin = BufReader::new(stdin());
    let mut lines = stdin.lines();

    while let Ok(Some(line)) = lines.next_line().await {
        if writer.write_all(format!("{}\n", line).as_bytes()).await.is_err() {
            break;
        }
    }
});

```

```

// Wait for either task to finish
tokio::select! {
    _ = read_handle => println!("Read task finished"),
    _ = write_handle => println!("Write task finished"),
}

Ok(())
}

```

This pattern—splitting reading and writing into separate tasks—is very powerful for building responsive network clients.

Example: Connection Pooling

For clients that make many connections to the same server, connection pooling can significantly improve performance:

```

use tokio::net::TcpStream;
use tokio::sync::Mutex;
use std::sync::Arc;
use std::collections::VecDeque;

/// Simple connection pool for reusing TCP connections
/// In production, use libraries like deadpool or bb8
struct ConnectionPool {
    available: Arc<Mutex<VecDeque<TcpStream>>>,
    address: String,
    max_size: usize,
}

impl ConnectionPool {
    fn new(address: String, max_size: usize) -> Self {
        ConnectionPool {
            available: Arc::new(Mutex::new(VecDeque::new())),
            address,
            max_size,
        }
    }

    async fn acquire(&self) -> tokio::io::Result<PooledConnection> {
        let mut pool = self.available.lock().await;

        // Try to reuse an existing connection
        if let Some(stream) = pool.pop_front() {
            return Ok(PooledConnection {
                stream: Some(stream),
                pool: self.available.clone(),
            });
        }

        // Otherwise create a new connection
    }
}

struct PooledConnection {
    stream: Option<TcpStream>,
    pool: Arc<Mutex<VecDeque<TcpStream>>>,
}

```

```

        drop(pool); // Release lock before async operation
    let stream = TcpStream::connect(&self.address).await?;

    Ok(PooledConnection {
        stream: Some(stream),
        pool: self.available.clone(),
    })
}

/// RAII wrapper that returns connection to pool on drop
struct PooledConnection {
    stream: Option<TcpStream>,
    pool: Arc<Mutex<VecDeque<TcpStream>>,
}

impl Drop for PooledConnection {
    fn drop(&mut self) {
        if let Some(stream) = self.stream.take() {
            let pool = self.pool.clone();
            tokio::spawn(async move {
                pool.lock().await.push_back(stream);
            });
        }
    }
}

impl std::ops::Deref for PooledConnection {
    type Target = TcpStream;

    fn deref(&self) -> &Self::Target {
        self.stream.as_ref().unwrap()
    }
}

impl std::ops::DerefMut for PooledConnection {
    fn deref_mut(&mut self) -> &mut Self::Target {
        self.stream.as_mut().unwrap()
    }
}

```

Pattern 2: UDP Patterns

Problem: Need low-latency connectionless communication where packet loss acceptable. TCP handshake/ack overhead too high for real-time data.

Solution: Use `tokio::net::UdpSocket.bind()` on server.

Why It Matters: Lower latency than TCP (no handshake, no acks)—critical for gaming, VoIP. Essential for real-time where latest data > old data (position updates).

Use Cases: Gaming (player position/state updates), VoIP (audio packets), video streaming (RTP), DNS queries, service discovery (mDNS, SSDP), IoT sensor data, time synchronization (NTP), multicast notifications, DHCP, TFTP.

Example: UDP Echo Server Pattern

Build simple UDP server that receives and responds to datagrams.

```
use tokio::net::UdpSocket;
use std::io;

/// UDP echo server
/// Receives datagrams and echoes them back to the sender
async fn udp_echo_server() -> io::Result<()> {
    // Bind to a port
    let socket = UdpSocket::bind("127.0.0.1:8080").await?;
    println!("UDP server listening on port 8080");

    let mut buffer = vec![0u8; 1024];

    loop {
        // Receive a datagram
        // recv_from returns the number of bytes and the sender's address
        let (len, addr) = socket.recv_from(&mut buffer).await?;

        println!("Received {} bytes from {}", len, addr);

        // Echo it back
        socket.send_to(&buffer[..len], addr).await?;
    }
}
```

Notice how much simpler this is than TCP—no connection management, no accept loop. Each packet is independent.

Example: UDP Client

```
use tokio::net::UdpSocket;

/// Send a UDP message and wait for a response
async fn udp_client_example() -> tokio::io::Result<()> {
    // Bind to any available port
    let socket = UdpSocket::bind("0.0.0.0:0").await?;

    // Connect sets the default destination
    // This doesn't establish a connection (UDP is connectionless)
    // but allows using send/recv instead of send_to/recv_from
    socket.connect("127.0.0.1:8080").await?;

    // Send a message
    let message = b"Hello, UDP Server!";
}
```

```

socket.send(message).await?;

// Wait for a response
let mut buffer = vec![0u8; 1024];
let len = socket.recv(&mut buffer).await?;

println!("Received: {}", String::from_utf8_lossy(&buffer[..len]));

Ok(())
}

```

Example: Broadcast and Multicast

UDP supports broadcasting to multiple recipients. This is useful for service discovery and distributed systems:

```

use tokio::net::UdpSocket;
use std::net::{IpAddr, Ipv4Addr, SocketAddr};

/// Broadcast a message to all hosts on the local network
async fn udp_broadcast() -> tokio::io::Result<()> {
    let socket = UdpSocket::bind("0.0.0.0:0").await?;

    // Enable broadcast
    socket.set_broadcast(true)?;

    // Broadcast address (255.255.255.255 reaches all hosts on local network)
    let broadcast_addr = SocketAddr::new(
        Ipv4Addr::V4(Ipv4Addr::new(255, 255, 255, 255)),
        8080
    );

    let message = b"Service Discovery Request";
    socket.send_to(message, broadcast_addr).await?;

    println!("Broadcast sent");
    Ok(())
}

/// Listen for broadcast messages
async fn udp_broadcast_listener() -> tokio::io::Result<()> {
    let socket = UdpSocket::bind("0.0.0.0:8080").await?;
    socket.set_broadcast(true)?;

    let mut buffer = vec![0u8; 1024];

    loop {
        let (len, addr) = socket.recv_from(&mut buffer).await?;
        println!("Broadcast from {}: {}", addr,
            String::from_utf8_lossy(&buffer[..len])
    );
}

```

```
    }  
}
```

Example: Reliable UDP Pattern

Sometimes you want UDP's low latency but need some reliability. Here's a simple request-response pattern with retries:

```
use tokio::net::UdpSocket;  
use tokio::time::{timeout, Duration};  
  
/// Send a UDP request with retries  
async fn reliable_udp_request(  
    socket: &UdpSocket,  
    message: &[u8],  
    server_addr: &str,  
    retries: usize,  
) -> tokio::io::Result<Vec<u8>> {  
    let mut buffer = vec![0u8; 1024];  
  
    for attempt in 0..retries {  
        // Send the request  
        socket.send_to(message, server_addr).await?;  
  
        // Wait for response with timeout  
        match timeout(Duration::from_secs(2), socket.recv_from(&mut buffer)).await {  
            Ok(Ok((len, _addr))) => {  
                // Success! Return the response  
                return Ok(buffer[..len].to_vec());  
            }  
            Ok(Err(e)) => return Err(e),  
            Err(_) => {  
                // Timeout - retry  
                println!("Attempt {} timed out, retrying...", attempt + 1);  
                continue;  
            }  
        }  
    }  
  
    Err(tokio::io::Error::new(  
        tokio::io::ErrorKind::TimedOut,  
        "All retry attempts failed"  
    ))  
}
```

This pattern is the basis for protocols like QUIC and helps bridge the gap between UDP's speed and TCP's reliability.

Pattern 3: HTTP Client (reqwest)

Problem: Need to make HTTP requests with async I/O. Handle cookies, headers, redirects automatically.

Solution: Use reqwest::Client with built-in connection pool. Async/await API.

Why It Matters: HTTP ubiquitous for APIs—essential for microservices. reqwest production-ready (connection pooling, retries, cookie management).

Use Cases: REST API clients, web scraping, microservice communication, webhook consumers, OAuth authentication flows, file downloads, GraphQL clients, API testing/monitoring, service health checks, data synchronization.

Example: Basic HTTP Client Pattern

Make HTTP requests with JSON payloads efficiently.

```
use reqwest;
use serde::Deserialize;

#[derive(Debug, Deserialize)]
struct ApiResponse {
    message: String,
    status: String,
}

/// Simple GET request
async fn simple_get_request() -> Result<(), Box<dyn std::error::Error>> {
    // GET request returns a Response
    let response = reqwest::get("https://httpbin.org/get").await?;

    println!("Status: {}", response.status());
    println!("Headers: {:?}", response.headers());

    // Read the response body as text
    let body = response.text().await?;
    println!("Body: {}", body);

    Ok(())
}

/// GET request with JSON deserialization
async fn get_json() -> Result<(), Box<dyn std::error::Error>> {
    let response = reqwest::get("https://api.example.com/data")
        .await?
        .json::()
        .await?;

    println!("Response: {:?}", response);
}
```

```
    Ok(())
}
```

The `.json()` method automatically deserializes the response body using serde, making it very convenient for API interactions.

Example: POST Requests and Request Building

For more complex requests, use the `Client` and `RequestBuilder`:

```
use reqwest::{Client, header};
use serde::{Deserialize, Serialize};
use std::collections::HashMap;

#[derive(Serialize)]
struct CreateUser {
    username: String,
    email: String,
}

#[derive(Deserialize, Debug)]
struct User {
    id: u64,
    username: String,
    email: String,
}

/// POST request with JSON body
async fn create_user() -> Result<(), Box
```

```

let mut form_data = HashMap::new();
form_data.insert("username", "bob");
form_data.insert("password", "secret123");

let response = client
    .post("https://example.com/login")
    .form(&form_data)
    .send()
    .await?;

println!("Login status: {}", response.status());
Ok(())
}

```

Example: Request Headers and Authentication

Many APIs require authentication headers. Here's how to add them:

```

use request::{Client, header};

/// Request with custom headers
async fn request_with_auth() -> Result<(), Box

```

```

// All requests with this client will include the default headers
let response = client
    .get("https://api.example.com/data")
    .send()
    .await?;

Ok(())
}

```

Example: Error Handling and Retries

Robust HTTP clients need proper error handling and retry logic:

```

use reqwest::{Client, StatusCode};
use tokio::time::{sleep, Duration};

/// Retry a request on failure
async fn request_with_retry(
    client: &Client,
    url: &str,
    max_retries: u32,
) -> Result<String, Box<dyn std::error::Error>> {
    let mut attempts = 0;

    loop {
        attempts += 1;

        match client.get(url).send().await {
            Ok(response) => {
                match response.status() {
                    StatusCode::OK => {
                        return Ok(response.text().await?);
                    }
                    StatusCode::TOO_MANY_REQUESTS => {
                        // Rate limited - wait and retry
                        if attempts >= max_retries {
                            return Err("Max retries exceeded".into());
                        }
                        println!("Rate limited, waiting...");
                        sleep(Duration::from_secs(5)).await;
                        continue;
                    }
                    status if status.is_server_error() => {
                        // Server error - retry with backoff
                        if attempts >= max_retries {
                            return Err(format!("Server error: {}", status).into());
                        }
                        let backoff = Duration::from_secs(2u64.pow(attempts));
                        println!("Server error, retrying in {:?}", backoff);
                        sleep(backoff).await;
                        continue;
                    }
                }
            }
        }
    }
}

```

```
        }
        status => {
            // Client error - don't retry
            return Err(format!("HTTP error: {}", status).into());
        }
    }
    Err(e) => {
        if attempts >= max_retries {
            return Err(e.into());
        }
        println!("Request failed: {}, retrying...", e);
        sleep(Duration::from_secs(2)).await;
        continue;
    }
}
}
```

This pattern—exponential backoff with retry limits—is essential for building resilient network clients.

Example: Downloading Files

For downloading large files, streaming the response is more memory-efficient than loading everything into memory:

```
use reqwest::Client;
use tokio::fs::File;
use tokio::io::AsyncWriteExt;
use futures_util::StreamExt;

/// Download a file with progress tracking
async fn download_file(
    url: &str,
    output_path: &str,
) -> Result<(), Box
```

```

        file.write_all(&chunk).await?;

        downloaded += chunk.len() as u64;
        if total_size > 0 {
            let percent = (downloaded as f64 / total_size as f64) * 100.0;
            print!("Progress: {:.2}%", percent);
        }
    }

    println!("\nDownload complete!");
    Ok(())
}

```

Pattern 4: HTTP Server (axum, actix-web)

Problem: Need HTTP server with routing, middleware, shared state. Handle concurrent requests safely.

Solution: Use axum Router for routing. State extractor for shared data (Arc for thread-safety).

Why It Matters: Web servers are core infrastructure—REST APIs, microservices, dashboards. axum built on tokio/hyper (100K+ req/s).

Use Cases: REST APIs, web applications, microservices, GraphQL servers (with async-graphql), webhook receivers, admin dashboards, file upload services, proxy/API gateway, authentication services, monitoring endpoints.

Example: axum Server Pattern

Build REST API with routing, JSON, and shared state.

```

use axum::{
    routing::{get, post},
    Router,
    Json,
    extract:: {Path, Query},
    response::IntoResponse,
    http::StatusCode,
};

use serde::{Deserialize, Serialize};
use std::net::SocketAddr;

#[derive(Serialize, Deserialize)]
struct User {
    id: u64,
    username: String,
    email: String,
}

#[derive(Deserialize)]
struct CreateUserRequest {
    username: String,
}

```

```

    email: String,
}

#[derive(Deserialize)]
struct ListQuery {
    page: Option<u32>,
    per_page: Option<u32>,
}

#[tokio::main]
async fn basic_axum_server() {
    // Build our application with routes
    let app = Router::new()
        .route("/", get(root_handler))
        .route("/users", get(list_users).post(create_user))
        .route("/users/:id", get(get_user));

    // Run the server
    let addr = SocketAddr::from(([127, 0, 0, 1], 3000));
    println!("Server running on http://{}", addr);

    axum::Server::bind(&addr)
        .serve(app.into_make_service())
        .await
        .unwrap();
}

//=====
// Handler for GET /
//=====

async fn root_handler() -> &'static str {
    "Hello, World!"
}

//=====
// Handler for GET /users
//=====

async fn list_users(Query(params): Query<ListQuery>) -> Json<Vec<User>> {
    let page = params.page.unwrap_or(1);
    let per_page = params.per_page.unwrap_or(10);

    println!("Listing users: page={}, per_page={}", page, per_page);

    // In a real app, fetch from database
    let users = vec![
        User {
            id: 1,
            username: "alice".to_string(),
            email: "alice@example.com".to_string(),
        },
        User {
            id: 2,
            username: "bob".to_string(),
        }
    ];
}

```

```

        email: "bob@example.com".to_string(),
    },
];

Json(users)
}

//=====
// Handler for GET /users/:id
//=====

async fn get_user(Path(user_id): Path<u64>) -> Result<Json<User>, StatusCode> {
    println!("Getting user {}", user_id);

    // Simulate database lookup
    if user_id == 1 {
        Ok(Json(User {
            id: 1,
            username: "alice".to_string(),
            email: "alice@example.com".to_string(),
        }))
    } else {
        Err(StatusCode::NOT_FOUND)
    }
}

//=====
// Handler for POST /users
//=====

async fn create_user(
    Json(payload): Json<CreateUserRequest>,
) -> (StatusCode, Json<User>) {
    println!("Creating user: {}", payload.username);

    // In a real app, save to database and return the created user
    let user = User {
        id: 42, // Would come from database
        username: payload.username,
        email: payload.email,
    };

    (StatusCode::CREATED, Json(user))
}

```

Notice how axum uses extractors (like `Path`, `Query`, `Json`) to parse and validate request data at compile time. If the types don't match, you get a compile error.

Example: Shared State in axum

Most applications need shared state (database connections, caches, etc.). axum makes this easy with the `State` extractor:

```
use axum::{
    routing::get,
    Router,
    extract::State,
    Json,
};

use std::sync::Arc;
use tokio::sync::RwLock;
use serde::Serialize;

#[derive(Clone)]
struct AppState {
    // Use Arc for shared ownership across tasks
    // RwLock allows multiple readers or one writer
    db: Arc<RwLock<Database>>,
    config: Arc<Config>,
}

struct Database {
    users: Vec<User>,
}

struct Config {
    max_users: usize,
}

#[derive(Serialize, Clone)]
struct User {
    id: u64,
    name: String,
}

#[tokio::main]
async fn stateful_server() {
    let state = AppState {
        db: Arc::new(RwLock::new(Database {
            users: vec![],
        })),
        config: Arc::new(Config {
            max_users: 1000,
        }),
    };

    let app = Router::new()
        .route("/users", get(get_all_users))
        .with_state(state);

    // Run server...
}

async fn get_all_users(
    State(state): State<AppState>,
```

```

) -> Json<Vec<User>> {
    // Acquire read lock
    let db = state.db.read().await;

    // Clone the users (in real app, might want to use pagination)
    Json(db.users.clone())
}

```

The `State` extractor ensures every handler has access to the application state without global variables.

Example: Middleware in axum

Middleware allows you to add cross-cutting concerns like logging, authentication, or CORS:

```

use axum::{
    Router,
    routing::get,
    middleware::{self, Next},
    response::Response,
    http::Request,
};

use std::time::Instant;

#[tokio::main]
async fn middleware_example() {
    let app = Router::new()
        .route("/", get(|| async { "Hello!" }))
        // Add middleware to all routes
        .layer(middleware::from_fn(timing_middleware))
        .layer(middleware::from_fn(auth_middleware));

    // Run server...
}

/// Middleware that logs request timing
async fn timing_middleware<B>(
    request: Request<B>,
    next: Next<B>,
) -> Response {
    let start = Instant::now();
    let uri = request.uri().clone();

    let response = next.run(request).await;

    let elapsed = start.elapsed();
    println!("{} took {:?}", uri, elapsed);

    response
}

/// Middleware that checks authentication
async fn auth_middleware<B>(

```

```

    request: Request<B>,
    next: Next<B>,
) -> Response {
    // Check for auth header
    if let Some(auth_header) = request.headers().get("authorization") {
        if auth_header.to_str().unwrap_or("").starts_with("Bearer ") {
            // Valid auth, continue
            return next.run(request).await;
        }
    }

    // No valid auth
    Response::builder()
        .status(401)
        .body("Unauthorized".into())
        .unwrap()
}

```

Middleware composes nicely, allowing you to build complex request processing pipelines.

Example: actix-web Server

actix-web is known for being extremely fast and feature-rich. It uses the actor model internally:

```

use actix_web::{web, App, HttpServer, HttpResponse, Responder};
use serde::{Deserialize, Serialize};

#[derive(Deserialize, Serialize)]
struct User {
    id: u64,
    name: String,
}

#[actix_web::main]
async fn actix_server() -> std::io::Result<()> {
    HttpServer::new(|| {
        App::new()
            .route("/", web::get().to(index))
            .route("/users", web::get().to(get_users))
            .route("/users", web::post().to(create_user))
            .route("/users/{id}", web::get().to(get_user))
    })
    .bind(("127.0.0.1", 8080))?
    .run()
    .await
}

async fn index() -> impl Responder {
    HttpResponse::Ok().body("Hello from actix-web!")
}

async fn get_users() -> impl Responder {

```

```

let users = vec![
    User { id: 1, name: "Alice".to_string() },
    User { id: 2, name: "Bob".to_string() },
];
}

HttpResponse::Ok().json(users)
}

async fn create_user(user: web::Json<User>) -> impl Responder {
    println!("Creating user: {}", user.name);
    HttpResponse::Created().json(user.into_inner())
}

async fn get_user(path: web::Path<u64>) -> impl Responder {
    let user_id = path.into_inner();

    if user_id == 1 {
        HttpResponse::Ok().json(User {
            id: 1,
            name: "Alice".to_string(),
        })
    } else {
        HttpResponse::NotFound().finish()
    }
}

```

actix-web is slightly more imperative in style compared to axum's declarative approach, but both are excellent choices.

Pattern 5: WebSocket Patterns

Problem: Need full-duplex real-time bidirectional communication. HTTP request-response inadequate for live updates.

Solution: Use tokio-tungstenite for WebSocket (or axum/actix WebSocket support). HTTP upgrade to WebSocket.

Why It Matters: Real-time applications require bidirectional push—can't rely on polling. WebSocket single persistent connection vs HTTP polling overhead (100 req/s vs 1 connection).

Use Cases: Chat applications (real-time messages), live notifications (alerts, updates), collaborative editing (Google Docs-style), stock tickers (price updates), gaming (multiplayer state sync), dashboard updates (metrics, logs), IoT device control, video streaming signaling.

Example: WebSocket Broadcast Pattern

Broadcast messages from any client to all connected WebSocket clients.

```

use axum::{
    routing::get,
    Router,
}

```

```
extract::{
    ws::{WebSocket, WebSocketUpgrade, Message},
    State,
},
response::IntoResponse,
};

use std::sync::Arc;
use tokio::sync::broadcast;

#[derive(Clone)]
struct AppState {
    // Broadcast channel for sending messages to all clients
    tx: broadcast::Sender<String>,
}

#[tokio::main]
async fn websocket_server() {
    // Create broadcast channel
    let (tx, _rx) = broadcast::channel(100);

    let state = AppState { tx };

    let app = Router::new()
        .route("/ws", get(websocket_handler))
        .with_state(state);

    println!("WebSocket server running on http://127.0.0.1:3000");

    axum::Server::bind(&"127.0.0.1:3000".parse().unwrap())
        .serve(app.into_make_service())
        .await
        .unwrap();
}

/// Handle WebSocket upgrade requests
async fn websocket_handler(
    ws: WebSocketUpgrade,
    State(state): State<AppState>,
) -> impl IntoResponse {
    // Complete the WebSocket upgrade
    ws.on_upgrade(|socket| handle_socket(socket, state))
}

/// Handle an individual WebSocket connection
async fn handle_socket(socket: WebSocket, state: AppState) {
    // Split the socket into sender and receiver
    let (mut sender, mut receiver) = socket.split();

    // Subscribe to broadcast channel
    let mut rx = state.tx.subscribe();

    // Spawn a task to send broadcast messages to this client
    let mut send_task = tokio::spawn(async move {
```

```

        while let Ok(msg) = rx.recv().await {
            // Send message to this client
            if sender.send(Message::Text(msg)).await.is_err() {
                break;
            }
        }
    });

// Spawn a task to receive messages from this client
let tx = state.tx.clone();
let mut recv_task = tokio::spawn(async move {
    while let Some(Ok(msg)) = receiver.next().await {
        if let Message::Text(text) = msg {
            // Broadcast the message to all clients
            let _ = tx.send(text);
        }
    }
});

// Wait for either task to finish
tokio::select! {
    _ = (&mut send_task) => recv_task.abort(),
    _ = (&mut recv_task) => send_task.abort(),
}

println!("Client disconnected");
}

```

This creates a simple chat server where all messages are broadcast to all connected clients. Each client gets two tasks: one for receiving broadcasts and one for sending messages.

Example: WebSocket Client

Here's a WebSocket client using tokio-tungstenite:

```

use tokio_tungstenite::{connect_async, tungstenite::protocol::Message};
use futures_util::{StreamExt, SinkExt};

async fn websocket_client() -> Result<(), Box<dyn std::error::Error>> {
    let url = "ws://127.0.0.1:3000/ws";

    // Connect to the server
    let (ws_stream, _) = connect_async(url).await?;
    println!("Connected to {}", url);

    let (mut write, mut read) = ws_stream.split();

    // Spawn a task to handle incoming messages
    let read_handle = tokio::spawn(async move {
        while let Some(msg) = read.next().await {
            match msg {
                Ok(Message::Text(text)) => {

```

```

        println!("Received: {}", text);
    }
    Ok(Message::Close(_)) => {
        println!("Server closed connection");
        break;
    }
    Err(e) => {
        eprintln!("Error: {}", e);
        break;
    }
    _ => {}
}
}

// Send some messages
for i in 0..5 {
    let msg = format!("Message {}", i);
    write.send(Message::Text(msg)).await?;
    tokio::time::sleep(tokio::time::Duration::from_secs(1)).await;
}

// Close the connection
write.send(Message::Close(None)).await?;

read_handle.await?;

Ok(())
}

```

Example: Room-based WebSocket Pattern

For more complex applications like chat rooms or game lobbies, you need to manage multiple rooms:

```

use axum::extract::ws::{WebSocket, Message};
use std::collections::HashMap;
use std::sync::Arc;
use tokio::sync::{RwLock, broadcast};

type RoomId = String;
type UserId = String;

struct ChatServer {
    rooms: Arc<RwLock<HashMap<RoomId, Room>>>,
}

struct Room {
    // Broadcast channel for this room
    tx: broadcast::Sender<ChatMessage>,
    // Connected users
    users: HashMap<UserId, UserInfo>,
}

```

```

struct UserInfo {
    username: String,
}

#[derive(Clone)]
struct ChatMessage {
    user_id: UserId,
    username: String,
    content: String,
}

impl ChatServer {
    fn new() -> Self {
        ChatServer {
            rooms: Arc::new(RwLock::new(HashMap::new())),
        }
    }

    async fn join_room(&self, room_id: RoomId, user_id: UserId, username: String) ->
        broadcast::Receiver<ChatMessage> {
        let mut rooms = self.rooms.write().await;

        let room = rooms.entry(room_id.clone()).or_insert_with(|| {
            let (tx, _) = broadcast::channel(100);
            Room {
                tx,
                users: HashMap::new(),
            }
        });

        room.users.insert(user_id, UserInfo { username });
        room.tx.subscribe()
    }

    async fn send_message(&self, room_id: &RoomId, msg: ChatMessage) {
        let rooms = self.rooms.read().await;
        if let Some(room) = rooms.get(room_id) {
            let _ = room.tx.send(msg);
        }
    }

    async fn leave_room(&self, room_id: &RoomId, user_id: &UserId) {
        let mut rooms = self.rooms.write().await;
        if let Some(room) = rooms.get_mut(room_id) {
            room.users.remove(user_id);

            // Clean up empty rooms
            if room.users.is_empty() {
                rooms.remove(room_id);
            }
        }
    }
}

```

```
    }
}
```

This pattern allows users to join specific rooms and only receive messages from those rooms, which is much more scalable than broadcasting everything to everyone.

Example: Handling Ping/Pong for Keep-Alive

WebSocket connections can silently die (network issues, etc.). Use ping/pong to detect dead connections:

```
use tokio::time::{interval, Duration};
use axum::extract::ws::{WebSocket, Message};

async fn websocket_with_keepalive(mut socket: WebSocket) {
    let mut ping_interval = interval(Duration::from_secs(30));

    loop {
        tokio::select! {
            _ = ping_interval.tick() => {
                // Send ping
                if socket.send(Message::Ping(vec![])).await.is_err() {
                    break;
                }
            }
            msg = socket.recv() => {
                match msg {
                    Some(Ok(Message::Pong(_))) => {
                        // Client is alive
                    }
                    Some(Ok(Message::Text(text))) => {
                        // Handle message
                        println!("Received: {}", text);
                    }
                    Some(Ok(Message::Close(_))) | None => {
                        break;
                    }
                    Some(Err(e)) => {
                        eprintln!("Error: {}", e);
                        break;
                    }
                    _ => {}
                }
            }
        }
    }

    println!("WebSocket connection closed");
}
```

This ensures you detect and clean up dead connections promptly.

Summary

This chapter covered network programming patterns:

1. **TCP Server/Client**: Async TcpListener/TcpStream, tokio::spawn per connection, BufReader for protocols
2. **UDP Patterns**: UdpSocket, send_to/recv_from, broadcast/multicast, connectionless communication
3. **HTTP Client (reqwest)**: Client with connection pool, JSON serialization, cookies, timeouts, retries
4. **HTTP Server (axum)**: Router, extractors (Json, Query, Path), middleware, shared State, type-safe
5. **WebSocket Patterns**: Full-duplex real-time, split read/write, broadcast channel, ping/pong keepalive

Key Takeaways: - Async I/O solves C10K problem—single thread handles 10K+ connections - TCP reliable but higher latency; UDP fast but lossy - reqwest industry standard for HTTP clients (connection pooling, retries) - axum type-safe extractors prevent runtime bugs (compile-time validation) - WebSocket enables server-initiated push (vs HTTP polling overhead)

Protocol Selection: - **TCP**: Reliable ordered delivery (HTTP, SSH, databases) - **UDP**: Low-latency real-time (gaming, VoIP, DNS) - **HTTP**: Request-response APIs (REST, microservices) - **WebSocket**: Bidirectional real-time (chat, live updates)

Performance Patterns: - Connection pooling (reqwest Client, database pools) - Buffering (BufReader for line protocols) - Async spawning (tokio::spawn per connection) - Graceful shutdown (CancellationToken) - Backpressure (bounded channels)

Error Handling: - Network errors common—design for failure - Timeouts essential (prevent hanging on dead connections) - Graceful degradation (retry with backoff) - Connection cleanup (tokio::select! for cancellation) - Validate input (malicious clients)

Best Practices: - Always set timeouts on network operations - Handle connection errors gracefully - Use connection pooling for performance - Implement graceful shutdown - Buffer I/O operations (avoid byte-by-byte) - Validate and sanitize all input - Use TLS for sensitive data

Common Use Cases: - **Chat server**: WebSocket with broadcast channel - **REST API**: axum with JSON extractors - **Game server**: UDP for position updates + TCP for critical events - **Microservices**: reqwest client + axum server - **Proxy**: TCP forwarding with tokio - **Live dashboard**: WebSocket for real-time metrics

Tools: - **tokio**: Async runtime, TcpListener/TcpStream/UdpSocket - **reqwest**: HTTP client with connection pooling - **axum**: HTTP server framework (type-safe, tokio-based) - **tokio-tungstenite**: WebSocket implementation - **tower**: Middleware for HTTP services - **hyper**: Low-level HTTP library (foundation for axum)

Database Patterns

This chapter explores database patterns in Rust: connection pooling for resource management, query builders for type safety, transactions for data integrity, migrations for schema evolution, and choosing between ORM and raw SQL based on specific requirements.

Pattern 1: Connection Pooling

Problem: Opening a fresh database connection per request costs handshakes, auth, and session setup, quickly exhausting `max_connections` and adding 100ms+ latency to every query.

Solution: Keep a pool of ready connections (`r2d2`, `deadpool`). Borrow them with `pool.get()`, configure sensible `max_size`, `min_idle`, and timeouts, and let the smart pointer return them on drop.

Why It Matters: Reusing 10–20 pooled connections serves hundreds of requests with <1 ms checkout time, protects the database from connection storms, and preserves prepared statements/buffers.

Use Cases: Web servers, background workers, GraphQL resolvers, CLI tools hitting DBs repeatedly, serverless globals, and integration test harnesses needing many short-lived queries.

Example: r2d2: The Classic Connection Pool

`r2d2` (Resource Reuse & Recycling Daemon) is Rust's original generic connection pool. It works with any resource that implements its traits, not just databases. This makes it extremely flexible.

```
// Add to Cargo.toml:
// r2d2 = "0.8"
// r2d2_postgres = "0.18"
// postgres = "0.19"

use r2d2::{Pool, PooledConnection};
use r2d2_postgres::{PostgresConnectionManager, postgres::NoTls};
use std::error::Error;

type PostgresPool = Pool<PostgresConnectionManager<NoTls>>;
type PostgresPooledConnection = PooledConnection<PostgresConnectionManager<NoTls>>;

/// Initialize a connection pool
fn create_pool(database_url: &str) -> Result<PostgresPool, Box<dyn Error>> {
    let manager = PostgresConnectionManager::new(
        database_url.parse()?,
        NoTls,
    );

    let pool = Pool::builder()
        .max_size(15)                      // Maximum 15 connections
        .min_idle(Some(5))                  // Keep at least 5 idle connections
        .connection_timeout(std::time::Duration::from_secs(5))
        .build(manager)?;
}
```

```

Ok(pool)
}

/// Example: Using the pool
async fn fetch_user(pool: &PostgresPool, user_id: i32) -> Result<String, Box<dyn Error>> {
    // Get a connection from the pool
    // This blocks if all connections are in use, until one becomes available
    let mut conn = pool.get()?;
    // Use the connection
    let row = conn.query_one(
        "SELECT username FROM users WHERE id = $1",
        &[&user_id],
    )?;
    let username: String = row.get(0);
    // Connection automatically returns to pool when `conn` is dropped
    Ok(username)
}

fn main() -> Result<(), Box<dyn Error>> {
    let pool = create_pool("postgresql://user:pass@localhost/mydb")?;

    // The pool can be cloned cheaply (Arc internally)
    // and shared across threads
    let pool_clone = pool.clone();

    let handle = std::thread::spawn(move || {
        fetch_user(&pool_clone, 42)
    });

    // Both threads can use the pool concurrently
    let username = fetch_user(&pool, 42)?;
    println!("User: {}", username);

    handle.join().unwrap()?;
    Ok(())
}

```

The beauty of this pattern is in the `PooledConnection` type. It's a smart pointer that automatically returns the connection to the pool when dropped. You can't forget to return it—Rust's ownership system enforces correct cleanup.

Example: Tuning Pool Configuration

Pool configuration significantly impacts performance. Here's what each parameter controls:

```

use r2d2::Pool;
use std::time::Duration;

```

```

fn configure_pool_detailed(manager: PostgresConnectionManager<NoTls>) -> PostgresPool {
    Pool::builder()
        // Maximum number of connections to create
        // Higher = more concurrent requests, but more database load
        .max_size(20)

        // Minimum idle connections to maintain
        // Higher = faster response for bursts, but more idle resources
        .min_idle(Some(5))

        // How long to wait for a connection before timing out
        // Too low = errors during load spikes
        // Too high = slow responses when pool exhausted
        .connection_timeout(Duration::from_secs(10))

        // Test connections before use to ensure they're alive
        // Adds overhead but prevents using dead connections
        .test_on_check_out(true)

        // How long a connection can be idle before being closed
        // Prevents accumulating stale connections
        .idle_timeout(Some(Duration::from_secs(300)))

        // Maximum lifetime of a connection before forced recreation
        // Ensures connections don't grow stale over time
        .max_lifetime(Some(Duration::from_secs(1800)))
    }

    .build(manager)
    .unwrap()
}

```

For a typical web application: - **Small app** (< 100 concurrent users): max_size = 5-10 - **Medium app** (100-1000 users): max_size = 10-20 - **Large app** (1000+ users): max_size = 20-50

Going higher than 50 often indicates other bottlenecks.

Example: deadpool: Async-First Connection Pooling

While r2d2 works well with blocking I/O, async applications need async-aware pooling. deadpool is designed specifically for async/await:

```

// Add to Cargo.toml:
// deadpool = { version = "0.10", features = ["managed"] }
// deadpool-postgres = "0.12"
// tokio-postgres = "0.7"
// tokio = { version = "1", features = ["full"] }

use deadpool_postgres::{Config, ManagerConfig, Pool, RecyclingMethod};
use tokio_postgres::NoTls;
use std::error::Error;

/// Create an async connection pool

```

```

fn create_async_pool() -> Result<Pool, Box<dyn Error>> {
    let mut cfg = Config::new();
    cfg.host = Some("localhost".to_string());
    cfg.user = Some("user".to_string());
    cfg.password = Some("password".to_string());
    cfg dbname = Some("mydb".to_string());

    cfg.manager = Some(ManagerConfig {
        recycling_method: RecyclingMethod::Fast,
    });

    cfg.pool = Some(deadpool::managed::PoolConfig {
        max_size: 20,
        timeouts: deadpool::managed::Timeouts {
            wait: Some(std::time::Duration::from_secs(5)),
            create: Some(std::time::Duration::from_secs(5)),
            recycle: Some(std::time::Duration::from_secs(5)),
        },
    });
}

Ok(cfg.create_pool(None, NoTls)?)
}

/// Fetch user asynchronously
async fn fetch_user_async(pool: &Pool, user_id: i32) -> Result<String, Box<dyn Error>> {
    // Get connection asynchronously
    // This awaits instead of blocking
    let client = pool.get().await?;

    // Execute query
    let row = client
        .query_one("SELECT username FROM users WHERE id = $1", &[&user_id])
        .await?;

    let username: String = row.get(0);

    // Connection returns to pool on drop
    Ok(username)
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error>> {
    let pool = create_async_pool()?;

    // Can handle many concurrent queries efficiently
    let tasks = (1..=100).map(|id| {
        let pool = pool.clone();
        tokio::spawn(async move {
            fetch_user_async(&pool, id).await
        })
    });

    // Wait for all queries
}

```

```

let results = futures::future::join_all(tasks).await;

println!("Completed {} queries", results.len());

Ok(())
}

```

The key difference: deadpool's `get()` returns a `Future` that yields to other tasks while waiting for a connection. With r2d2, a thread would block. This makes deadpool much more efficient for high-concurrency async applications.

Example: Health Checks and Monitoring

Production applications need to monitor pool health:

```

use deadpool_postgres::Pool;

async fn monitor_pool_health(pool: &Pool) {
    let status = pool.status();

    println!("Pool status:");
    println!(" Available connections: {}", status.available);
    println!(" Size: {}", status.size);
    println!(" Max size: {}", status.max_size);

    // Alert if pool is exhausted
    if status.available == 0 {
        eprintln!("WARNING: Connection pool exhausted!");
    }

    // Alert if pool is mostly idle (might be oversized)
    if status.available > status.max_size * 3 / 4 {
        eprintln!("INFO: Pool mostly idle, consider reducing size");
    }
}

/// Graceful shutdown
async fn shutdown_pool(pool: Pool) {
    println!("Shutting down pool...");

    // Close the pool
    pool.close();

    // Give active connections time to finish
    tokio::time::sleep(std::time::Duration::from_secs(5)).await;

    println!("Pool shutdown complete");
}

```

Monitoring helps you tune pool size and detect issues before they impact users.

Pattern 2: Query Builders

Problem: Raw SQL strings hide typos, wrong parameter counts, and type mismatches until runtime—and string concatenation invites SQL injection.

Solution: Use compile-time checked builders: SQLx `query!` verifies syntax/columns/types, Diesel's schema DSL provides type-safe composable queries, and SQLx's `QueryBuilder` binds parameters for dynamic clauses.

Why It Matters: Mistakes surface as compiler errors instead of production crashes, refactors update in one place, and bound parameters shut the door on injection.

Use Cases: Everyday CRUD endpoints, GraphQL resolvers, dashboards with dynamic filters, reporting queries, and any service that needs confidence its SQL matches the schema.

Example: SQLx: The Compile-Time Checked Query Builder

SQLx is remarkable: it connects to your database at compile time and verifies your queries. This means SQL errors become compile errors:

```
// Add to Cargo.toml:
// sqlx = { version = "0.7", features = ["runtime-tokio-native-tls", "postgres", "macros"] }

use sqlx::{PgPool, FromRow};

#[derive(FromRow, Debug)]
struct User {
    id: i32,
    username: String,
    email: String,
    created_at: chrono::NaiveDateTime,
}

async fn create_pool(database_url: &str) -> Result<PgPool, sqlx::Error> {
    // SQLx has built-in connection pooling
    PgPool::connect(database_url).await
}

/// Compile-time verified query
async fn fetch_user(pool: &PgPool, user_id: i32) -> Result<User, sqlx::Error> {
    // The query! macro verifies this SQL at compile time
    // It checks:
    // - SQL syntax is valid
    // - Table and columns exist
    // - Parameter types match
    // - Return types match
    let user = sqlx::query_as!(
        User,
        "SELECT id, username, email, created_at FROM users WHERE id = $1",
        user_id
    )
}
```

```

    .fetch_one(pool)
    .await?;

    Ok(user)
}

/// Insert with query!
async fn create_user(
    pool: &PgPool,
    username: &str,
    email: &str,
) -> Result<User, sqlx::Error> {
    let user = sqlx::query_as!(
        User,
        r#"
        INSERT INTO users (username, email, created_at)
        VALUES ($1, $2, NOW())
        RETURNING id, username, email, created_at
        "#,
        username,
        email
    )
    .fetch_one(pool)
    .await?;

    Ok(user)
}

```

If you misspell a column name or use the wrong type, your code won't compile. This is a game-changer for database programming.

Example: Offline Mode for CI/CD

The compile-time checking requires a database connection. For CI/CD where you might not have a live database, SQLx provides offline mode:

```

# Save query metadata locally
cargo sqlx prepare

# This creates .sqlx/ directory with query metadata
# Commit this to version control

# Now compilation works without database
cargo build

```

This workflow gives you compile-time safety without requiring a database in CI.

Example: Dynamic Queries with SQLx

Sometimes you need to build queries dynamically. SQLx supports this too:

```

use sqlx::{PgPool, Postgres, QueryBuilder};

async fn search_users(
    pool: &PgPool,
    username_filter: Option<&str>,
    email_filter: Option<&str>,
    limit: i64,
) -> Result<Vec<User>, sqlx::Error> {
    // QueryBuilder for dynamic queries
    let mut query_builder: QueryBuilder<Postgres> = QueryBuilder::new(
        "SELECT id, username, email, created_at FROM users WHERE 1=1"
    );

    if let Some(username) = username_filter {
        query_builder.push(" AND username LIKE ");
        query_builder.push_bind(format!("'{}'", username));
    }

    if let Some(email) = email_filter {
        query_builder.push(" AND email LIKE ");
        query_builder.push_bind(format!("'{}'", email));
    }

    query_builder.push(" LIMIT ");
    query_builder.push_bind(limit);

    let users = query_builder
        .build_query_as:::<User>()
        .fetch_all(pool)
        .await?;

    Ok(users)
}

```

This is type-safe and SQL-injection safe (all values are bound parameters), while still allowing runtime flexibility.

Example: Diesel: Full-Featured ORM

Diesel is Rust's most mature ORM. It provides a complete type-safe query DSL that never requires raw SQL:

```

// Add to Cargo.toml:
// diesel = { version = "2.1", features = ["postgres", "chrono"] }
// diesel_migrations = "2.1"

use diesel::prelude::*;
use diesel::pg::PgConnection;

//=====
// Define schema (typically generated by Diesel CLI)

```

```

//=====


```

```

let new_user = NewUser {
    username: new_username,
    email: new_email,
};

diesel::insert_into(users)
    .values(&new_user)
    .returning(User::as_returning())
    .get_result(conn)
}

/// Complex query with joins
fn find_active_users(conn: &mut PgConnection) -> QueryResult<Vec<User>> {
    use self::users::dsl::*;

    users
        .filter(created_at.gt(chrono::Utc::now().naive_utc() - chrono::Duration::days(30)))
        .order(username.asc())
        .limit(100)
        .select(User::as_select())
        .load(conn)
}

```

Diesel's approach is entirely type-safe. The compiler ensures your queries are correct at compile time. The trade-off is a steeper learning curve and less flexibility for complex queries.

Example: Diesel with r2d2

Combining Diesel with r2d2 gives you the best of both worlds:

```

use diesel::r2d2::{self, ConnectionManager};
use diesel::PgConnection;

type Pool = r2d2::Pool<ConnectionManager<PgConnection>>;

fn create_diesel_pool(database_url: &str) -> Pool {
    let manager = ConnectionManager::new(database_url);
    r2d2::Pool::builder()
        .max_size(15)
        .build(manager)
        .expect("Failed to create pool")
}

fn fetch_user_pooled(pool: &Pool, user_id: i32) -> QueryResult<User> {
    let mut conn = pool.get()
        .expect("Failed to get connection from pool");

    use self::users::dsl::*;
    users.find(user_id).first(&mut conn)
}

```

This pattern is common in Diesel applications—the ORM handles queries, while the pool manages connections.

Pattern 3: Transaction Patterns

Problem: Multi-step operations without transactions leave money half-transferred, rows orphaned, and concurrent edits overwriting each other.

Solution: Wrap sequences in transactional APIs (`pool.begin()`, Diesel's `transaction` closure) so they commit only on success, use savepoints for partial rollbacks, and add optimistic locking/version checks for concurrent writers.

Why It Matters: ACID semantics prevent corruption, the type system forces explicit commits or automatic rollbacks, and conflict detection avoids silent lost updates.

Use Cases: Payments, inventory reservations, multi-table writes, user onboarding flows, audit logging, and collaborative editors that need optimistic concurrency.

Example: Basic Transactions with SQLx

```
use sqlx::{PgPool, Postgres, Transaction};

async fn transfer_money(
    pool: &PgPool,
    from_account: i32,
    to_account: i32,
    amount: f64,
) -> Result<(), sqlx::Error> {
    // Start a transaction
    let mut tx: Transaction<Postgres> = pool.begin().await?;

    // Debit from account
    sqlx::query!(
        "UPDATE accounts SET balance = balance - $1 WHERE id = $2",
        amount,
        from_account
    )
    .execute(&mut *tx)
    .await?;

    // Credit to account
    sqlx::query!(
        "UPDATE accounts SET balance = balance + $1 WHERE id = $2",
        amount,
        to_account
    )
    .execute(&mut *tx)
    .await?;

    // Commit the transaction
    // If we return early (error), the transaction auto-rolls back on drop
```

```
    tx.commit().await?;

    Ok(())
}
```

The key insight: if any operation fails, the transaction automatically rolls back when `tx` is dropped. You can't forget to handle errors—the type system enforces it.

Example: Nested Transactions (Savepoints)

Some databases support savepoints for nested transactions:

```
use sqlx::{PgPool, Postgres, Transaction};

async fn complex_operation(pool: &PgPool) -> Result<(), sqlx::Error> {
    let mut tx = pool.begin().await?;

    // Operation 1
    sqlx::query!("INSERT INTO logs (message) VALUES ('Starting')")
        .execute(&mut *tx)
        .await?;

    // Create a savepoint for nested transaction
    let mut savepoint = tx.begin().await?;

    // Try a risky operation
    match risky_operation(&mut savepoint).await {
        Ok(_) => {
            // Success – commit the savepoint
            savepoint.commit().await?;
        }
        Err(e) => {
            // Failure – rollback just the savepoint
            // The outer transaction continues
            eprintln!("Risky operation failed: {}", e);
            savepoint.rollback().await?;
        }
    }

    // Operation 2 (happens regardless of risky_operation outcome)
    sqlx::query!("INSERT INTO logs (message) VALUES ('Completed')")
        .execute(&mut *tx)
        .await?;

    tx.commit().await?;

    Ok(())
}

async fn risky_operation(tx: &mut Transaction<'_, Postgres>) -> Result<(), sqlx::Error> {
    sqlx::query!("INSERT INTO risky_table (value) VALUES (100)")
        .execute(&mut **tx)
```

```
.await?;

Ok(())
}
```

Savepoints allow partial rollbacks, which is useful for complex multi-step operations.

Example: Transaction Isolation Levels

Different isolation levels provide different guarantees:

```
use sqlx::{PgPool, Postgres};

async fn set_isolation_level(pool: &PgPool) -> Result<(), sqlx::Error> {
    let mut tx = pool.begin().await?;

    // Set isolation level
    sqlx::query("SET TRANSACTION ISOLATION LEVEL SERIALIZABLE")
        .execute(&mut *tx)
        .await?;

    // Perform operations with serializable isolation
    // This prevents phantom reads and ensures true serializability

    tx.commit().await?;

    Ok(())
}
```

The four standard isolation levels are:

1. **Read Uncommitted**: Can see uncommitted changes (dirty reads)
2. **Read Committed**: Only sees committed data (default in PostgreSQL)
3. **Repeatable Read**: Sees a consistent snapshot
4. **Serializable**: Full isolation, as if transactions ran serially

Higher isolation means fewer anomalies but more contention and potential for deadlocks.

Example: Diesel Transactions

Diesel uses a different pattern for transactions:

```
use diesel::prelude::*;
use diesel::result::Error;

fn transfer_with_diesel(
    conn: &mut PgConnection,
    from: i32,
    to: i32,
    amount: i32,
) -> Result<(), Error> {
```

```

conn.transaction(|conn| {
    // All operations inside this closure are in a transaction

    diesel::update(accounts::table.find(from))
        .set(accounts::balance.eq(accounts::balance - amount))
        .execute(conn)?;

    diesel::update(accounts::table.find(to))
        .set(accounts::balance.eq(accounts::balance + amount))
        .execute(conn)?;

    // Return Ok to commit, Err to rollback
    Ok(())
})
}

```

The closure-based API is elegant: returning `Ok` commits, returning `Err` rolls back. You can't accidentally forget to commit or rollback.

Example: Optimistic Locking

For concurrent updates, optimistic locking prevents lost updates:

```

use sqlx::PgPool;

#[derive(sqlx::FromRow)]
struct Document {
    id: i32,
    content: String,
    version: i32,
}

async fn update_with_optimistic_lock(
    pool: &PgPool,
    doc_id: i32,
    new_content: String,
) -> Result<(), Box

```

```

        WHERE id = $2 AND version = $3
        "#,
        new_content,
        doc_id,
        doc.version
    )
.execute(pool)
.await?;

if result.rows_affected() > 0 {
    // Success - we updated it
    return Ok(());
}

// Someone else updated it - retry
println!("Conflict detected, retrying...");
tokio::time::sleep(std::time::Duration::from_millis(100)).await;
}
}

```

This pattern avoids database locks while preventing lost updates. It's ideal for long-running operations or distributed systems.

Pattern 4: Migration Strategies

Problem: Ad-hoc SQL changes leave environments out of sync, makes onboarding painful, and gives no safe rollback when a deployment goes wrong.

Solution: Treat schema changes as code: timestamped up/down migrations via SQLx or Diesel, committed to Git, run automatically (or via CLI) in every environment, and split risky changes into multi-phase, zero-downtime steps.

Why It Matters: Reproducible schemas, automated CI checks, audit trails, straightforward rollbacks, and collaborative workflows all depend on consistent, versioned migrations.

Use Cases: Any production database, CI pipelines, blue/green deploys, multi-developer teams coordinating schema changes, and new hires needing one command to match prod.

Example: SQLx Migrations

SQLx includes a built-in migration system:

```

# Create migrations directory
mkdir -p migrations

# Add a migration
sqlx migrate add create_users_table

```

This creates a file like `migrations/20240101000000_create_users_table.sql`:

```
-- migrations/20240101000000_create_users_table.sql
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(255) NOT NULL UNIQUE,
    email VARCHAR(255) NOT NULL UNIQUE,
    password_hash VARCHAR(255) NOT NULL,
    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMP NOT NULL DEFAULT NOW()
);

CREATE INDEX idx_users_email ON users(email);
CREATE INDEX idx_users_username ON users(username);
```

Run migrations programmatically:

```
use sqlx::postgres::PgPoolOptions;

#[tokio::main]
async fn main() -> Result<(), Box

```

Or use the CLI:

```
# Run all pending migrations
sqlx migrate run

# Revert last migration
sqlx migrate revert
```

Example: Diesel Migrations

Diesel has a more sophisticated migration system:

```
# Install Diesel CLI
cargo install diesel_cli --no-default-features --features postgres

# Setup Diesel
```

```
diesel setup

# Create migration
diesel migration generate create_users_table
```

This creates up and down migration files:

```
-- migrations/2024-01-01-000000_create_users_table/up.sql
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(255) NOT NULL UNIQUE,
    email VARCHAR(255) NOT NULL UNIQUE,
    created_at TIMESTAMP NOT NULL DEFAULT NOW()
);

-- migrations/2024-01-01-000000_create_users_table/down.sql
DROP TABLE users;
```

Run migrations:

```
# Apply all pending migrations
diesel migration run

# Revert last migration
diesel migration revert

# Redo last migration (useful for development)
diesel migration redo
```

Diesel automatically updates `src/schema.rs` with the table definitions, keeping your Rust code in sync with the database.

Example: Migration Best Practices

Follow these principles for successful migrations:

1. Make Migrations Reversible

Always provide down migrations:

```
-- up.sql
ALTER TABLE users ADD COLUMN phone VARCHAR(20);

-- down.sql
ALTER TABLE users DROP COLUMN phone;
```

This allows you to roll back changes if something goes wrong.

2. Never Modify Existing Migrations

Once a migration runs in production, it's immutable. Create a new migration to fix issues:

```
-- WRONG: Modifying old migration
-- migrations/001_create_users.sql (don't edit this!)

-- RIGHT: Create new migration
-- migrations/002_fix_users_table.sql
ALTER TABLE users ALTER COLUMN email TYPE VARCHAR(320);
```

3. Test Migrations Thoroughly

Test both up and down migrations:

```
# Test up migration
diesel migration run

# Test down migration
diesel migration revert

# Test redo (down then up)
diesel migration redo
```

4. Keep Migrations Small

Small migrations are easier to review and debug:

```
-- GOOD: One focused change
-- migrations/003_add_user_index.sql
CREATE INDEX idx_users_created_at ON users(created_at);

-- AVOID: Multiple unrelated changes
-- migrations/004_big_changes.sql
CREATE INDEX idx_users_created_at ON users(created_at);
ALTER TABLE posts ADD COLUMN featured BOOLEAN;
CREATE TABLE tags (...);
```

Example: Data Migrations

Sometimes you need to migrate data, not just schema:

```
-- migrations/005_normalize_emails/up.sql

-- Add new column
ALTER TABLE users ADD COLUMN email_normalized VARCHAR(255);

-- Populate with normalized emails
UPDATE users SET email_normalized = LOWER(email);
```

```
-- Make it NOT NULL
ALTER TABLE users ALTER COLUMN email_normalized SET NOT NULL;

-- Add unique constraint
ALTER TABLE users ADD CONSTRAINT users_email_normalized_unique UNIQUE (email_normalized);

-- Create index
CREATE INDEX idx_users_email_normalized ON users(email_normalized);
```

For complex data migrations, use application code:

```
use sqlx::PgPool;

async fn run_data_migration(pool: &PgPool) -> Result<(), sqlx::Error> {
    // Start transaction
    let mut tx = pool.begin().await?;

    // Fetch users in batches
    let mut offset = 0;
    let batch_size = 1000;

    loop {
        let users: Vec<(i32, String)> = sqlx::query_as(
            "SELECT id, email FROM users LIMIT $1 OFFSET $2"
        )
        .bind(batch_size)
        .bind(offset)
        .fetch_all(&mut *tx)
        .await?;

        if users.is_empty() {
            break;
        }

        // Process each user
        for (id, email) in users {
            let normalized = email.to_lowercase();
            sqlx::query!(
                "UPDATE users SET email_normalized = $1 WHERE id = $2",
                normalized,
                id
            )
            .execute(&mut *tx)
            .await?;
        }

        offset += batch_size;
    }

    tx.commit().await?;
}
```

```
Ok(())
}
```

Processing in batches prevents memory exhaustion on large datasets.

Example: Zero-Downtime Migrations

For production systems that can't have downtime, follow this pattern:

1. **Add new column (nullable)**: Deploy this first, application ignores it
2. **Backfill data**: Populate the new column
3. **Update application**: Start using the new column
4. **Make column NOT NULL**: After all data is backfilled
5. **Remove old column**: After confirming new column works

Example timeline:

```
-- Migration 1: Add nullable column
ALTER TABLE users ADD COLUMN new_email VARCHAR(320);

-- Migration 2: Backfill data (run with application handling both columns)
UPDATE users SET new_email = old_email WHERE new_email IS NULL;

-- Migration 3: Make NOT NULL (after deploy using new column)
ALTER TABLE users ALTER COLUMN new_email SET NOT NULL;

-- Migration 4: Drop old column (after confirming everything works)
ALTER TABLE users DROP COLUMN old_email;
```

Each step can be deployed independently without downtime.

Pattern 5: ORM vs Raw SQL

Problem: ORMs make CRUD safe but struggle with complex SQL features, while raw SQL gives full power at the expense of compile-time checks and refactor safety.

Solution: Mix and match—use Diesel (or similar ORM) for routine operations where type safety shines, and fall back to SQLx/raw SQL for analytics, window functions, CTEs, or database-specific features, still binding parameters to avoid injection.

Why It Matters: You get the best of both worlds: painless CRUD refactors plus full SQL expressiveness when needed, without forcing the entire codebase into one paradigm.

Use Cases: CRUD-heavy modules, admin panels, and migrations via ORM; reporting, full-text search, JSONB operators, and tuned analytical queries via SQLx or handcrafted SQL.

Example: Hybrid Approach

The best solution often combines both:

```

use sqlx::PgPool;
use diesel::prelude::*;

//=====
// Simple CRUD: Use ORM
//=====

fn create_user_orm(conn: &mut PgConnection, name: &str) -> QueryResult<User> {
    diesel::insert_into(users::table)
        .values(users::username.eq(name))
        .get_result(conn)
}

//=====
// Complex analytics: Use raw SQL
//=====

async fn get_sales_report(pool: &PgPool) -> Result<Vec<SalesData>, sqlx::Error> {
    sqlx::query_as!(SalesData, r#"
        SELECT
            date_trunc('day', created_at) as day,
            COUNT(*) as total_orders,
            SUM(amount) as total_revenue,
            AVG(amount) as avg_order_value
        FROM orders
        WHERE created_at >= NOW() - INTERVAL '30 days'
        GROUP BY day
        ORDER BY day DESC
    "#)
        .fetch_all(pool)
        .await
}

//=====
// Database-specific features: Use raw SQL
//=====

async fn full_text_search(pool: &PgPool, query: &str) -> Result<Vec<Document>, sqlx::Error> {
    sqlx::query_as!(Document, r#"
        SELECT id, title, content
        FROM documents
        WHERE to_tsvector('english', title || ' ' || content) @@ plainto_tsquery('english',
        $1)
        ORDER BY ts_rank(to_tsvector('english', title || ' ' || content),
        plainto_tsquery('english', $1)) DESC
        LIMIT 20
    "#, query)
        .fetch_all(pool)
        .await
}

```

When to Use Each

Use an ORM (Diesel) when: - Building standard CRUD operations - Type safety is critical - You want database portability - Your team is learning SQL - Migrations are frequent

Use raw SQL (SQLx) when: - Queries are complex (joins, subqueries, CTEs) - Performance is critical - Using database-specific features - Queries are dynamic - Debugging is important

Use both when: - Building a real application (most cases) - Different parts have different needs - You want flexibility

Example: Complete Application Example

Here's a realistic application structure combining patterns:

```
use sqlx::PgPool;
use deadpool_postgres::Pool;

//=====
// Application configuration
//=====

pub struct AppState {
    pool: PgPool,
}

//=====
// User repository using SQLx
//=====

pub struct UserRepository {
    pool: PgPool,
}

impl UserRepository {
    pub fn new(pool: PgPool) -> Self {
        Self { pool }
    }

    pub async fn create(&self, username: &str, email: &str) -> Result<User, sqlx::Error> {
        sqlx::query_as!(
            User,
            r#"
                INSERT INTO users (username, email, created_at)
                VALUES ($1, $2, NOW())
                RETURNING id, username, email, created_at
            "#,
            username,
            email
        )
        .fetch_one(&self.pool)
        .await
    }

    pub async fn find_by_id(&self, id: i32) -> Result<Option<User>, sqlx::Error> {
        sqlx::query_as!(
            User,
            "SELECT id, username, email, created_at FROM users WHERE id = $1",
            id
        )
        .fetch_optional(&self.pool)
        .await
    }
}
```

```

        )
        .fetch_optional(&self.pool)
        .await
    }

    pub async fn list_active(&self, days: i32) -> Result<Vec<User>, sqlx::Error> {
        sqlx::query_as!(
            User,
            r#"
                SELECT id, username, email, created_at
                FROM users
                WHERE created_at > NOW() - $1 * INTERVAL '1 day'
                ORDER BY created_at DESC
            "#,
            days
        )
        .fetch_all(&self.pool)
        .await
    }

}

//=====
// Analytics using raw SQL for complex queries
//=====

pub struct Analytics {
    pool: PgPool,
}

impl Analytics {
    pub async fn user_growth_report(&self) -> Result<Vec<GrowthMetric>, sqlx::Error> {
        sqlx::query_as!(
            GrowthMetric,
            r#"
                WITH daily_signups AS (
                    SELECT
                        date_trunc('day', created_at) as signup_date,
                        COUNT(*) as new_users
                    FROM users
                    GROUP BY signup_date
                )
                SELECT
                    signup_date as date,
                    new_users,
                    SUM(new_users) OVER (ORDER BY signup_date) as cumulative_users
                FROM daily_signups
                ORDER BY signup_date DESC
                LIMIT 30
            "#
        )
        .fetch_all(&self.pool)
        .await
    }
}

```

```

#[derive(sqlx::FromRow)]
pub struct User {
    pub id: i32,
    pub username: String,
    pub email: String,
    pub created_at: chrono::NaiveDateTime,
}

#[derive(sqlx::FromRow)]
pub struct GrowthMetric {
    pub date: Option<chrono::NaiveDateTime>,
    pub new_users: Option<i64>,
    pub cumulative_users: Option<i64>,
}

```

This structure separates concerns: simple operations use type-safe queries, while analytics uses raw SQL for complex aggregations.

Summary

This chapter covered database patterns for production Rust applications:

1. **Connection Pooling:** r2d2 (sync), deadpool (async); reuse connections eliminating 50-200ms overhead
2. **Query Builders:** SQLx compile-time verification, Diesel type-safe DSL, prevent SQL errors at compile-time
3. **Transaction Patterns:** ACID guarantees via type system, savepoints, optimistic locking, isolation levels
4. **Migration Strategies:** Version-controlled schema as code, up/down migrations, zero-downtime deployments
5. **ORM vs Raw SQL:** Hybrid approach—Diesel for CRUD, SQLx for complex queries, maximize both safety and power

Key Takeaways:

- Connection pooling mandatory for production: 50-200x faster than per-request connections
- Compile-time SQL verification eliminates runtime errors: typos become compile errors
- Type system enforces transaction safety: auto-rollback on drop, explicit commit required
- Migrations as code = reproducible schemas, rollback capability, team coordination
- Hybrid ORM/SQL approach: type safety for simple queries, SQL power for complex analytics

Performance Guidelines:

- Pool sizing: small apps 5-10, medium 10-20, large 20-50 connections (rarely need >50)
- Async pooling (deadpool) for high-concurrency: avoids thread blocking, better multiplexing
- Transaction isolation: READ COMMITTED (default) vs REPEATABLE READ vs SERIALIZABLE (trade-off: consistency vs concurrency)
- Prepared statements: reused via pooled connections, query planning cached
- Batch processing: process data migrations in batches (1000-10000 rows) prevents memory exhaustion

Best Practices:

- Monitor pool health: alert if available = 0 (exhausted) or > 75% (oversized)
- Use transactions for multi-step operations: prevents partial failures
- Never modify applied migrations: create new migration to fix
- Test migrations: `run` then `revert` then `redo` before production
- Keep

migrations small: one logical change per file - Use savepoints for partial rollback: outer transaction continues if inner fails - Implement connection validation: `test_on_check_out` detects dead connections - Configure timeouts: `connection_timeout` prevents indefinite waiting - Batch data migrations: process 1000-10000 rows at a time - Use indexes: query performance depends on proper indexing (outside scope of patterns, but critical)

Testing & Benchmarking

This chapter explores Rust's testing ecosystem end-to-end: built-in unit tests, property-based testing with proptest/quickcheck, coverage-guided fuzzing via cargo-fuzz, mutation testing tools that stress-check your suite, formal verification with Kani/Prusti, trait-based mocking, integration testing patterns, and Criterion benchmarks for performance measurement and regression detection.

Pattern 1: Unit Test Patterns

Problem: Rust ensures memory safety but not domain correctness; manual spot-checking misses edge cases, error paths, and regressions.

Solution: Lean on `#[test]`, assertion macros, and `#[should_panic]` to encode expectations. Keep tests near code via `#[cfg(test)]`, use RAII for setup/teardown, and tag slow suites with `#[ignore]`.

Why It Matters: Automated tests document intent, catch bugs before review, and provide fearless refactoring—especially on rarely exercised Err/panic branches.

Use Cases: Math/string helpers, API contracts, validation logic, panic semantics, regression reproducers, and any code that changes frequently.

Example: The Basics of Rust Testing

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

    #[test]
    fn test_addition() {
        assert_eq!(2 + 2, 4);
    }

    #[test]
    fn test_subtraction() {
        let result = 10 - 3;
        assert_eq!(result, 7);
    }
}
```

Run tests with `cargo test`. The test runner automatically discovers and executes all test functions, reporting successes and failures.

The `#[cfg(test)]` attribute ensures test modules only compile during testing. This keeps test code out of release builds, reducing binary size.

Example: Assertion Macros

Rust provides several assertion macros for different scenarios:

```
#[test]
fn assertion_examples() {
    // Basic equality
    assert_eq!(5, 2 + 3);

    // Inequality
    assert_ne!(5, 6);

    // Boolean assertions
    assert!(true);
    assert!(5 > 3, "5 should be greater than 3");

    // Custom error messages
    let x = 10;
    assert_eq!(x, 10, "x should be 10, but was {}", x);
}
```

The `assert_eq!` and `assert_ne!` macros provide better error messages than `assert!` because they show both the expected and actual values when tests fail.

Example: Testing Error Cases

Good tests verify both success and failure paths. Rust makes this elegant:

```
fn divide(a: i32, b: i32) -> Result<i32, String> {
    if b == 0 {
        Err("Division by zero".to_string())
    } else {
        Ok(a / b)
    }
}

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

    #[test]
    fn test_divide_success() {
        assert_eq!(divide(10, 2), Ok(5));
    }

    #[test]
    fn test_divide_by_zero() {
        assert!(divide(10, 0).is_err());
    }
}
```

```
}

#[test]
fn test_divide_error_message() {
    match divide(10, 0) {
        Err(msg) => assert_eq!(msg, "Division by zero"),
        Ok(_) => panic!("Expected error, got Ok"),
    }
}
```

Testing error cases is crucial—many bugs lurk in error paths because they’re harder to test manually.

Example: Testing Panics

Some functions should panic in certain conditions. Test this with `#[should_panic]`:

```
fn validate_age(age: u32) {
    if age > 150 {
        panic!("Age {} is unrealistic", age);
    }
}

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

    #[test]
    #[should_panic]
    fn test_invalid_age_panics() {
        validate_age(200);
    }

    #[test]
    #[should_panic(expected = "unrealistic")]
    fn test_panic_message() {
        validate_age(200);
    }
}
```

The `expected` parameter verifies that the panic message contains specific text, ensuring you’re panicking for the right reason.

Example: Organizing Tests

As codebases grow, test organization becomes important. Here are common patterns:

```
//=====
// src/math.rs
//=====

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

```

        a + b
    }

pub fn multiply(a: i32, b: i32) -> i32 {
    a * b
}

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

    mod addition_tests {
        use super::*;

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

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

        #[test]
        fn test_mixed_signs() {
            assert_eq!(add(-2, 3), 1);
        }
    }

    mod multiplication_tests {
        use super::*;

        #[test]
        fn test_positive_numbers() {
            assert_eq!(multiply(2, 3), 6);
        }

        #[test]
        fn test_by_zero() {
            assert_eq!(multiply(5, 0), 0);
        }
    }
}

```

Nested modules help organize related tests, making the test suite easier to navigate and maintain.

Example: Test Setup and Teardown

Sometimes tests need setup or cleanup. Rust doesn't have built-in setup/teardown hooks, but you can use regular Rust patterns:

```

struct TestContext {
    temp_dir: std::path::PathBuf,
}

impl TestContext {
    fn new() -> Self {
        let temp_dir = std::env::temp_dir().join(format!("test_{}", uuid::Uuid::new_v4()));
        std::fs::create_dir_all(&temp_dir).unwrap();
        TestContext { temp_dir }
    }
}

impl Drop for TestContext {
    fn drop(&mut self) {
        // Cleanup happens automatically when TestContext is dropped
        let _ = std::fs::remove_dir_all(&self.temp_dir);
    }
}

#[test]
fn test_with_temp_directory() {
    let ctx = TestContext::new();

    // Use ctx.temp_dir for testing
    let test_file = ctx.temp_dir.join("test.txt");
    std::fs::write(&test_file, "test content").unwrap();

    assert!(test_file.exists());

    // ctx is dropped here, cleaning up temp_dir
}

```

This pattern uses RAII (Resource Acquisition Is Initialization) for automatic cleanup. The compiler guarantees cleanup happens, even if the test panics.

Example: Ignoring and Filtering Tests

During development, you might want to skip expensive or unfinished tests:

```

#[test]
#[ignore]
fn expensive_test() {
    // This test takes a long time
    std::thread::sleep(std::time::Duration::from_secs(10));
}

#[test]
#[ignore = "Not yet implemented"]
fn todo_test() {

```

```
        unimplemented!()
    }
```

Run ignored tests with `cargo test -- --ignored`. Run all tests (including ignored) with `cargo test -- --include-ignored`.

Filter tests by name:

```
# Run only tests with "addition" in the name
cargo test addition

# Run tests in a specific module
cargo test math::tests::addition_tests
```

Example: Testing Private Functions

Tests in the same file can access private functions:

```
fn internal_helper(x: i32) -> i32 {
    x * 2
}

pub fn public_api(x: i32) -> i32 {
    internal_helper(x) + 1
}

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

    #[test]
    fn test_internal_helper() {
        // Can test private functions
        assert_eq!(internal_helper(5), 10);
    }

    #[test]
    fn test_public_api() {
        assert_eq!(public_api(5), 11);
    }
}
```

This is a deliberate design decision. Tests in the same module are part of the implementation, so they can access private details.

Pattern 2: Property-Based Testing

Problem: Example-based tests cover only inputs you imagine, leaving unseen edge cases, MIN/MAX values, and weird permutations unchecked.

Solution: Property-based testing (proptest/quickcheck) generates hundreds of random inputs and shrinks failures, letting you describe invariants instead of enumerating cases.

Why It Matters: Automatic exploration surfaces bugs humans miss, shrinking provides minimal reproducers, and one property can replace dozens of example tests.

Use Cases: Pure functions, data-structure invariants, serialization round-trips, parsers, crypto/compression transforms, and deterministic state machines.

Example: Can do better

```
fn sort(mut vec: Vec<i32>) -> Vec<i32> {
    vec.sort();
    vec
}

#[test]
fn test_sort() {
    assert_eq!(sort(vec![3, 1, 2]), vec![1, 2, 3]);
}
```

This test is fine, but what about: - Empty vectors? - Single-element vectors? - Already-sorted vectors? - Reverse-sorted vectors? - Duplicate elements? - Very large vectors? - Vectors with MIN and MAX values?

You could write tests for each case, but you'd still miss edge cases. Property-based testing explores the input space automatically.

Example: Introduction to proptest

proptest is Rust's leading property-based testing library. It generates random test cases and verifies your properties hold:

```
// Add to Cargo.toml:
// [dev-dependencies]
// proptest = "1.0"

use proptest::prelude::*;

proptest! {
    #[test]
    fn test_sort_properties(mut vec: Vec<i32>) {
        let sorted = sort(vec.clone());

        // Property 1: Output length equals input length
        prop_assert_eq!(sorted.len(), vec.len());

        // Property 2: Output is sorted
        for i in 1..sorted.len() {
            prop_assert!(sorted[i - 1] <= sorted[i]);
        }
    }
}
```

```

    }

    // Property 3: Output contains same elements as input
    vec.sort();
    prop_assert_eq!(sorted, vec);
}

}

```

proptest generates hundreds of random vectors and verifies all three properties hold. If it finds a failure, it “shrinks” the input to find the minimal failing case.

Example: Shrinking: Finding Minimal Failing Cases

Shrinking is proptest’s killer feature. When a test fails, proptest tries smaller, simpler inputs to find the smallest case that still fails:

```

fn buggy_absolute_value(x: i32) -> i32 {
    if x < 0 {
        -x
    } else {
        x
    }
}

proptest! {
    #[test]
    fn test_absolute_value(x: i32) {
        let result = buggy_absolute_value(x);
        prop_assert!(result >= 0);
    }
}

```

This test fails because `buggy_absolute_value(i32::MIN)` panics (overflow). proptest might initially find the failure with a large negative number, but it shrinks to the simplest failing case: `i32::MIN`.

Example: Custom Generators

Sometimes you need specific input patterns:

```

use proptest::prelude::*;

//=====
// Generate vectors of length 1-100
//=====

prop_compose! {
    fn vec_1_to_100()(vec in prop::collection::vec(any::<i32>(), 1..=100)) -> Vec<i32> {
        vec
    }
}

//=====

```

```
// Generate email-like strings
//=====
prop_compose! {
    fn email_strategy()(
        username in "[a-z]{3,10}",
        domain in "[a-z]{3,10}",
        tld in "(com|org|net)"
    ) -> String {
        format!("{}@{}.{}", username, domain, tld)
    }
}

proptest! {
    #[test]
    fn test_with_custom_generator(vec in vec_1_to_100()) {
        prop_assert!(!vec.is_empty());
        prop_assert!(vec.len() <= 100);
    }

    #[test]
    fn test_email_parsing(email in email_strategy()) {
        prop_assert!(email.contains('@'));
        prop_assert!(email.contains('.'));
    }
}
```

Custom generators let you focus testing on realistic inputs while still getting proptest's shrinking and reporting.

Example: Testing Invariants

Property-based testing excels at verifying invariants—properties that should always hold:

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

fn merge_maps(mut a: HashMap<String, i32>, b: HashMap<String, i32>) -> HashMap<String, i32> {
    for (k, v) in b {
        *a.entry(k).or_insert(0) += v;
    }
    a
}

proptest! {
    #[test]
    fn test_merge_properties(
        a: HashMap<String, i32>,
        b: HashMap<String, i32>,
    ) {
        let merged = merge_maps(a.clone(), b.clone());

        // Property 1: All keys from both maps are in the result
        for key in a.keys().chain(b.keys()) {
```

```

        prop_assert!(merged.contains_key(key));
    }

// Property 2: Values are summed correctly
for key in merged.keys() {
    let expected = a.get(key).unwrap_or(&0) + b.get(key).unwrap_or(&0);
    prop_assert_eq!(merged[key], expected);
}

// Property 3: Merging with empty map is identity
let empty: HashMap<String, i32> = HashMap::new();
prop_assert_eq!(merge_maps(a.clone(), empty.clone()), a);
}
}

```

These properties completely specify `merge_maps`' behavior without testing specific examples.

Example: QuickCheck

QuickCheck is another property-based testing library, inspired by Haskell's QuickCheck:

```

// Add to Cargo.toml:
// [dev-dependencies]
// quickcheck = "1.0"
// quickcheck_macros = "1.0"

use quickcheck::{quickcheck, TestResult};
use quickcheck_macros::quickcheck;

#[quickcheck]
fn reverse_twice_is_identity(vec: Vec<i32>) -> bool {
    let mut reversed = vec.clone();
    reversed.reverse();
    reversed.reverse();
    vec == reversed
}

#[quickcheck]
fn concat_length(a: Vec<i32>, b: Vec<i32>) -> bool {
    let mut c = a.clone();
    c.extend(b.iter());
    c.len() == a.len() + b.len()
}

```

QuickCheck's syntax is slightly different from proptest, but the concept is the same. Choose based on your preference—both are excellent.

Example: When to Use Property-Based Testing

Property-based testing shines when:

- **Testing pure functions:** Functions without side effects have clear properties
- **Testing data structures:** Invariants like “BST is always sorted” are perfect for properties
- **Finding edge cases:** You want to discover bugs you haven’t thought of
- **Testing serialization:** Round-tripping properties like `deserialize(serialize(x)) == x`

It’s less useful for:

- **Testing specific business logic:** “User discount is 10% for orders over \$100” is better as an example
- **Testing I/O:** Hard to generate meaningful random database queries or file operations
- **Complex stateful systems:** Can work but requires sophisticated generators

Pattern 3: Coverage-Guided Fuzzing

Problem: Static test sets rarely include adversarial byte sequences, so parsers and `unsafe` code still panic or blow up on malformed inputs.

Solution: `cargo-fuzz` (libFuzzer/AFL) mutates inputs guided by coverage, hammering targets that accept `&[u8]` or `Arbitrary` structs while sanitizers catch UB.

Why It Matters: Fuzzers discover crashers humans never craft, shrink them to minimal reproducers, and can run unattended to guard against future regressions.

Use Cases: Binary/text parsers, protocol stacks, CLI argument handling, unsafe abstractions, codecs, deserializers, and any surface open to untrusted data.

Example: Setting Up cargo-fuzz

```
cargo install cargo-fuzz
cargo fuzz init
```

This creates a `fuzz/` workspace. Add a target:

```
// fuzz_targets/parse_expr.rs
#![no_main]
use libfuzzer_sys::fuzz_target;

fuzz_target!(<|data: &[u8]| {
    if let Ok(expr) = my_crate::Expr::parse_from_bytes(data) {
        // Round trip: serialize then parse again
        let encoded = expr.to_bytes();
        let decoded = my_crate::Expr::parse_from_bytes(&encoded).unwrap();
        assert_eq!(expr, decoded);
    }
});
```

Run it with `cargo fuzz run parse_expr`. Crashes are saved in `fuzz/artifacts/parse_expr/`.

Example: Fuzzing Structured Inputs with `arbitrary`

```
// fuzz_targets/http_request.rs
#![no_main]
use arbitrary::Arbitrary;
use libfuzzer_sys::fuzz_target;

#[derive(Arbitrary, Debug)]
struct Request<'a> {
    method: &'a str,
    path: &'a str,
    body: &'a [u8],
}

fuzz_target!(<|req: Request| {
    let _ = my_http::handle_request(req.method, req.path, req.body);
});
```

The `arbitrary` derive creates structured random data (methods, paths, payloads), enabling deeper protocol coverage. Persist interesting seeds by copying them into `fuzz/corpus/http_request/`.

Example: Sanitizers and CI

Enable sanitizers to catch undefined behavior:

```
RUSTFLAGS="-Zsanitizer=address" \
RUSTC_BOOTSTRAP=1 \
cargo fuzz run parse_expr
```

In CI, run fuzzers for a bounded time:

```
cargo fuzz run parse_expr -- --max_total_time=60
```

Store the corpus to reuse progress between runs.

Pattern 4: Mutation Testing

Problem: Coverage numbers say code executed, not that tests would fail if behavior changes; weak assertions let bugs slip through untouched.

Solution: Mutation tools (`cargo-mutants`, `mutagen`) systematically tweak operators, constants, and control flow, then re-run tests to see which mutations survive.

Why It Matters: Surviving mutants highlight missing or shallow assertions, giving a concrete to-do list for hardening critical logic.

Use Cases: Pricing/auth pipelines, parsers, financial formulas, protocol state machines—any code where regressions are costly.

Example: cargo-mutants Workflow

```
cargo install cargo-mutants  
cargo mutants
```

Sample output:

```
Mutant 12: src/calculator.rs:42 replaced `>` with `>=`
Result: survived (tests passed)
```

Add or strengthen tests until important mutants die, then re-run with `cargo mutants --mutants 12` to confirm.

Example: Targeted Mutants

Focus on hot modules:

```
cargo mutants --mutate src/pricing.rs --mutate src/tax.rs
```

Pair with `--list` to inspect generated mutants before running them.

Example: Mutagen Annotations

`mutagen` instruments code at compile time:

```
// Cargo.toml
[dev-dependencies]
mutagen = "0.1"

// src/lib.rs
#[cfg_attr(test, mutagen::mutate)]
pub fn is_eligible(age: u8) -> bool {
    age >= 18
}
```

Running `cargo test` under `RUSTFLAGS="--cfg mutate"` generates mutants on the fly, surfacing weak tests without separate tooling.

Pattern 5: Formal Verification with Kani

Problem: Even deep tests only sample behaviors; safety-critical code sometimes needs proofs that no input can violate invariants.

Solution: Model checkers like Kani or provers like Prusti explore all executions within bounds using `#[kani::proof]` functions and nondeterministic inputs.

Why It Matters: Proofs guarantee absence of panics/overflow in small kernels, validating `unsafe` code or financial logic beyond what fuzzing can cover.

Use Cases: Unsafe abstractions, lock-free primitives, crypto/math kernels, serialization code, and embedded control algorithms.

Example: Verifying a Safe Add

```
// src/lib.rs
pub fn checked_add(a: u32, b: u32) -> Option<u32> {
    a.checked_add(b)
}

// proofs/add.rs
#[kani::proof]
fn checked_add_never_wraps() {
    let a = kani::any::<u32>();
    let b = kani::any::<u32>();
    if let Some(sum) = checked_add(a, b) {
        assert!(sum >= a && sum >= b);
    }
}
```

Run `cargo kani proofs/add.rs`. Kani symbolically explores all `u32` combinations and proves the postcondition.

Example: Proving State Machines

```
#[derive(Clone, Copy, PartialEq, Eq)]
enum DoorState { Locked, Unlocked }

fn next(state: DoorState, code_entered: bool) -> DoorState {
    match (state, code_entered) {
        (DoorState::Locked, true) => DoorState::Unlocked,
        (DoorState::Unlocked, false) => DoorState::Locked,
        _ => state,
    }
}

#[kani::proof]
fn door_never_skips_locked_state() {
    let code1 = kani::any::<bool>();
    let code2 = kani::any::<bool>();

    let s1 = next(DoorState::Locked, code1);
    let s2 = next(s1, code2);

    assert!(matches!(s2, DoorState::Locked | DoorState::Unlocked));
}
```

For more complex models, consider Prusti or Creusot for contract-based verification.

Pattern 6: Mock and Stub Patterns

Problem: Tests that talk to real databases, HTTP APIs, or queues are slow, flaky, and hard to coerce into failure modes.

Solution: Depend on traits, supply real implementations in production and mocks/fakes/stubs in tests (handwritten or via `mockall`), and inject them via generics or builders.

Why It Matters: Mocked tests run instantly, can simulate any error, and remain deterministic/parallelizable without external setup.

Use Cases: Database adapters, HTTP clients, payment/email integrations, file/queue abstractions, cache layers, or any boundary crossing process.

Example: Trait-Based Mocking

The most idiomatic approach uses traits:

```
//=====
// Define a trait for the dependency
//=====

trait EmailService {
    fn send_email(&self, to: &str, subject: &str, body: &str) -> Result<(), String>;
}

//=====
// Real implementation
//=====

struct SmtpEmailService {
    server: String,
}

impl EmailService for SmtpEmailService {
    fn send_email(&self, to: &str, subject: &str, body: &str) -> Result<(), String> {
        // Actually send email via SMTP
        println!("Sending to {} via {}", to, self.server);
        Ok(())
    }
}

//=====
// Mock for testing
//=====

struct MockEmailService {
    sent_emails: std::sync::Mutex<Vec<(String, String)>>,
}

impl MockEmailService {
    fn new() -> Self {
        MockEmailService {
            sent_emails: Default::default(),
        }
    }

    fn send_email(&self, to: &str, subject: &str, body: &str) -> Result<(), String> {
        self.sent_emails.lock().unwrap().push((to.to_string(), subject.to_string()));
        Ok(())
    }
}
```

```

        sent_emails: std::sync::Mutex::new(Vec::new()),
    }
}

fn emails_sent(&self) -> Vec<(String, String, String)> {
    self.sent_emails.lock().unwrap().clone()
}
}

impl EmailService for MockEmailService {
    fn send_email(&self, to: &str, subject: &str, body: &str) -> Result<(), String> {
        self.sent_emails.lock().unwrap().push((
            to.to_string(),
            subject.to_string(),
            body.to_string(),
        ));
        Ok(())
    }
}

//=====
// Application code uses the trait
//=====

struct UserService<E: EmailService> {
    email_service: E,
}

impl<E: EmailService> UserService<E> {
    fn register_user(&self, email: &str) -> Result<(), String> {
        // ... registration logic ...

        self.email_service.send_email(
            email,
            "Welcome!",
            "Thanks for registering",
        )?;

        Ok(())
    }
}

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

    #[test]
    fn test_user_registration_sends_email() {
        let mock = MockEmailService::new();
        let service = UserService {
            email_service: &mock,
        };

        service.register_user("user@example.com").unwrap();
    }
}
```

```

        let emails = mock.emails_sent();
        assert_eq!(emails.len(), 1);
        assert_eq!(emails[0].0, "user@example.com");
        assert_eq!(emails[0].1, "Welcome!");
    }
}

```

This pattern is powerful: the real code uses `EmailService` trait, tests use `MockEmailService`, production uses `SmtpEmailService`. No mocking framework needed.

Example: Using mockall for Advanced Mocking

For complex mocking needs, the `mockall` crate provides a powerful framework:

```

//=====
// Add to Cargo.toml:
//=====
// [dev-dependencies]
//=====
// mockall = "0.12"
//=====

use mockall::{automock, predicate::*;

#[automock]
trait Database {
    fn get_user(&self, id: i32) -> Option<User>;
    fn save_user(&mut self, user: User) -> Result<(), String>;
}

struct User {
    id: i32,
    name: String,
}

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

    #[test]
    fn test_with_mock() {
        let mut mock = MockDatabase::new();

        // Set expectations
        mock.expect_get_user()
            .with(eq(42))
            .times(1)
            .returning(|_| Some(User { id: 42, name: "Alice".to_string() }));

        mock.expect_save_user()
            .times(1)
    }
}

```

```

        .returning(|_| Ok(()));

    // Use the mock
    let user = mock.get_user(42).unwrap();
    assert_eq!(user.name, "Alice");

    mock.save_user(user).unwrap();

    // Automatically verifies expectations were met
}
}

```

mockall automatically generates mock implementations and verifies expectations, similar to mocking frameworks in other languages.

Example: Dependency Injection Patterns

Dependency injection makes testing easier by making dependencies explicit:

```

//=====
// Poor: Hard to test
//=====

struct PaymentProcessor {
    // Hard-coded dependency
}

impl PaymentProcessor {
    fn process_payment(&self, amount: f64) -> Result<(), String> {
        // Directly calls external API
        external_api::charge_card(amount)
    }
}

//=====
// Better: Dependency injection
//=====

trait PaymentGateway {
    fn charge(&self, amount: f64) -> Result<String, String>;
}

struct PaymentProcessor<G: PaymentGateway> {
    gateway: G,
}

impl<G: PaymentGateway> PaymentProcessor<G> {
    fn new(gateway: G) -> Self {
        PaymentProcessor { gateway }
    }

    fn process_payment(&self, amount: f64) -> Result<(), String> {
        let transaction_id = self.gateway.charge(amount)?;
        println!("Processed payment: {}", transaction_id);
    }
}

```

```

        Ok(())
    }
}

//=====
// Test with mock gateway
//=====

struct MockGateway {
    should_succeed: bool,
}

impl PaymentGateway for MockGateway {
    fn charge(&self, amount: f64) -> Result<String, String> {
        if self.should_succeed {
            Ok(format!("txn_{}", amount))
        } else {
            Err("Payment failed".to_string())
        }
    }
}

#[test]
fn test_successful_payment() {
    let gateway = MockGateway { should_succeed: true };
    let processor = PaymentProcessor::new(gateway);

    assert!(processor.process_payment(99.99).is_ok());
}

#[test]
fn test_failed_payment() {
    let gateway = MockGateway { should_succeed: false };
    let processor = PaymentProcessor::new(gateway);

    assert!(processor.process_payment(99.99).is_err());
}

```

This pattern—using traits for dependencies and injecting implementations—is idiomatic Rust and makes testing straightforward.

Example: Test Doubles for I/O

File system and network operations need special handling:

```

trait FileSystem {
    fn read_file(&self, path: &str) -> std::io::Result<String>;
    fn write_file(&self, path: &str, content: &str) -> std::io::Result<()>;
}

//=====
// Real implementation
//=====

```

```
struct RealFileSystem;

impl FileSystem for RealFileSystem {
    fn read_file(&self, path: &str) -> std::io::Result<String> {
        std::fs::read_to_string(path)
    }

    fn write_file(&self, path: &str, content: &str) -> std::io::Result<()> {
        std::fs::write(path, content)
    }
}

//=====
// In-memory fake for testing
//=====

use std::collections::HashMap;
use std::sync::Mutex;

struct FakeFileSystem {
    files: Mutex<HashMap<String, String>>,
}

impl FakeFileSystem {
    fn new() -> Self {
        FakeFileSystem {
            files: Mutex::new(HashMap::new()),
        }
    }
}

impl FileSystem for FakeFileSystem {
    fn read_file(&self, path: &str) -> std::io::Result<String> {
        self.files
            .lock()
            .unwrap()
            .get(path)
            .cloned()
            .ok_or_else(|| std::io::Error::new(
                std::io::ErrorKind::NotFound,
                "File not found"
            ))
    }

    fn write_file(&self, path: &str, content: &str) -> std::io::Result<()> {
        self.files
            .lock()
            .unwrap()
            .insert(path.to_string(), content.to_string());
        Ok(())
    }
}

#[test]
```

```

fn test_file_operations() {
    let fs = FakeFileSystem::new();

    fs.write_file("/test.txt", "hello").unwrap();
    let content = fs.read_file("/test.txt").unwrap();

    assert_eq!(content, "hello");
}

```

This fake is fast, deterministic, and doesn't touch the actual file system.

Pattern 7: Integration Testing

Problem: Unit tests hit internals but miss failures in public APIs, cross-component wiring, and real workflows.

Solution: Place integration tests under `tests/` so each file is its own crate using only the public surface, share setup via `tests/common`, and spin up real dependencies (DB transactions, HTTP servers, binaries).

Why It Matters: Validates that components cooperate as deployed, catches schema/serialization/API mismatches, and documents usage exactly as consumers experience it.

Use Cases: Web stacks (HTTP + DB + cache), CLI binaries, public libraries, migrations, multi-step business flows, and workspace crates that must interoperate.

Example: Integration Tests Structure

```

my_project/
├── src/
│   └── lib.rs
└── tests/
    ├── integration_test.rs
    └── common/
        └── mod.rs
Cargo.toml

```

Each file in `tests/` is compiled as a separate crate:

```

//=====
// tests/integration_test.rs
//=====

use my_project::*;

#[test]
fn test_public_api() {
    let result = public_function();

```

```
    assert_eq!(result, expected_value);
}
```

Integration tests only have access to your crate's public API, just like external users.

Example: Common Test Code

Shared test utilities go in `tests/common/`:

```
//=====
// tests/common/mod.rs
//=====
use my_project::*;

pub fn setup_test_database() -> Database {
    Database::new(":memory:")
}

pub fn create_test_user() -> User {
    User {
        id: 1,
        name: "Test User".to_string(),
        email: "test@example.com".to_string(),
    }
}
```

```
//=====
// tests/integration_test.rs
//=====

mod common;

#[test]
fn test_with_common_utilities() {
    let db = common::setup_test_database();
    let user = common::create_test_user();

    // Test using shared utilities
}
```

The `common/mod.rs` pattern prevents Rust from treating `common` as a test file.

Example: Testing Binary Crates

Binary crates can be tested by moving logic to a library:

```
my_binary/
├── src/
│   ├── main.rs      # Thin wrapper
│   └── lib.rs       # Business logic
```

```
└── tests/
    └── integration.rs
```

```
//=====
// src/lib.rs
//=====

pub fn run(args: Args) -> Result<(), Error> {
    // Application logic
}

//=====
// src/main.rs
//=====

fn main() {
    let args = parse_args();
    if let Err(e) = my_binary::run(args) {
        eprintln!("Error: {}", e);
        std::process::exit(1);
    }
}

//=====
// tests/integration.rs
//=====

use my_binary::*;

#[test]
fn test_application_logic() {
    let args = Args { /* ... */ };
    assert!(run(args).is_ok());
}
```

This structure makes the binary testable while keeping `main.rs` simple.

Example: Database Integration Tests

Testing with real databases requires setup and teardown:

```
//=====
// tests/database_integration.rs
//=====

use sqlx::PgPool;

async fn setup_test_db() -> PgPool {
    let database_url = std::env::var("TEST_DATABASE_URL")
        .unwrap_or_else(|_| "postgresql://localhost/test".to_string());

    let pool = PgPool::connect(&database_url).await.unwrap();

    // Run migrations
    sqlx::migrate!("./migrations")
```

```

    .run(&pool)
    .await
    .unwrap();

    // Clear existing data
    sqlx::query("TRUNCATE TABLE users, posts CASCADE")
        .execute(&pool)
        .await
        .unwrap();

    pool
}

#[tokio::test]
async fn test_user_creation() {
    let pool = setup_test_db().await;

    // Test code
    let user = create_user(&pool, "test@example.com").await.unwrap();
    assert_eq!(user.email, "test@example.com");
}

```

For parallel tests, use separate test databases or transactions:

```

use sqlx::{PgPool, Postgres, Transaction};

async fn run_in_transaction<F, Fut>(pool: &PgPool, test: F)
where
    F: FnOnce(Transaction<Postgres>) -> Fut,
    Fut: std::future::Future<Output = ()>,
{
    let mut tx = pool.begin().await.unwrap();

    test(tx).await;

    // Always rollback - test changes never persist
    // (Transaction is dropped here, triggering rollback)
}

#[tokio::test]
async fn test_in_transaction() {
    let pool = setup_test_db().await;

    run_in_transaction(&pool, |mut tx| async move {
        sqlx::query("INSERT INTO users (email) VALUES ('test@example.com')")
            .execute(&mut *tx)
            .await
            .unwrap();

        let count: (i64,) = sqlx::query_as("SELECT COUNT(*) FROM users")
            .fetch_one(&mut *tx)
            .await
    })
}

```

```

    .unwrap();

    assert_eq!(count.0, 1);
}) .await;

// Database is unchanged - transaction was rolled back
}

```

This pattern runs tests in isolation without slow database cleanup.

Example: HTTP Integration Tests

Testing HTTP servers requires starting a test server:

```

//=====
// tests/http_integration.rs
//=====

use axum::Router, routing::get;
use hyper::StatusCode;

async fn create_test_app() -> Router {
    Router::new()
        .route("/", get(|| async { "Hello, World!" }))
        .route("/user/:id", get(get_user))
}

#[tokio::test]
async fn test_hello_endpoint() {
    let app = create_test_app().await;

    let response = app
        .oneshot(
            axum::http::Request::builder()
                .uri("/")
                .body(axum::body::Body::empty())
                .unwrap()
        )
        .await
        .unwrap();

    assert_eq!(response.status(), StatusCode::OK);

    let body = hyper::body::to_bytes(response.into_body()).await.unwrap();
    assert_eq!(&body[..], b"Hello, World!");
}

```

For end-to-end tests with a running server:

```

use tokio::net::TcpListener;

#[tokio::test]

```

```

async fn test_full_server() {
    let listener = TcpListener::bind("127.0.0.1:0").await.unwrap();
    let addr = listener.local_addr().unwrap();

    tokio::spawn(async move {
        axum::Server::from_tcp(listener.into_std().unwrap())
            .unwrap()
            .serve(create_test_app()).await.into_make_service()
            .await
            .unwrap();
    });

    // Wait for server to start
    tokio::time::sleep(tokio::time::Duration::from_millis(100)).await;

    // Make real HTTP request
    let client = reqwest::Client::new();
    let response = client
        .get(&format!("http://{}/", addr))
        .send()
        .await
        .unwrap();

    assert_eq!(response.status(), 200);
    assert_eq!(response.text().await.unwrap(), "Hello, World!");
}

```

Pattern 8: Criterion Benchmarking

Problem: Optimizing without measurements wastes time and hides regressions; single-run microbenchmarks are noisy and opaque about scaling.

Solution: Criterion automates statistical benchmarking, comparing implementations, varying input sizes, measuring throughput, and storing baselines while `black_box` thwarts dead-code elimination.

Why It Matters: Data-driven performance work avoids guesswork, flags slowdowns before release, and quantifies improvement with confidence intervals.

Use Cases: Algorithm comparisons, hot-path validation, throughput analysis, regression detection in CI, and picking between alternative data structures or implementations.

Example: Basic Criterion Benchmarks

```

// Add to Cargo.toml:
// [dev-dependencies]
// criterion = "0.5"
//
// [[bench]]
// name = "my_benchmark"
// harness = false

```

```
// benches/my_benchmark.rs
use criterion::{black_box, criterion_group, criterion_main, Criterion};

fn fibonacci(n: u64) -> u64 {
    match n {
        0 => 1,
        1 => 1,
        n => fibonacci(n - 1) + fibonacci(n - 2),
    }
}

fn criterion_benchmark(c: &mut Criterion) {
    c.bench_function("fib 20", |b| b.iter(|| fibonacci(black_box(20))));
}

criterion_group!(benches, criterion_benchmark);
criterion_main!(benches);
```

Run with `cargo bench`. Criterion runs the benchmark multiple times, detects and removes outliers, and reports statistics:

```
fib 20                      time: [26.029 µs 26.251 µs 26.509 µs]
```

The `black_box` function prevents the compiler from optimizing away the computation.

Example: Comparing Implementations

Benchmark multiple implementations to choose the best:

```
use criterion::{black_box, criterion_group, criterion_main, BenchmarkId, Criterion};

fn sum_loop(data: &[i32]) -> i32 {
    let mut sum = 0;
    for &x in data {
        sum += x;
    }
    sum
}

fn sum_iterator(data: &[i32]) -> i32 {
    data.iter().sum()
}

fn sum_fold(data: &[i32]) -> i32 {
    data.iter().fold(0, |acc, &x| acc + x)
}

fn benchmark_sum_implementations(c: &mut Criterion) {
    let mut group = c.benchmark_group("sum_implementations");

    let data: Vec<i32> = (0..1000).collect();
```

```

        group.bench_with_input(BenchmarkId::new("loop", data.len()), &data, |b, data| {
            b.iter(|| sum_loop(black_box(data)))
        });

        group.bench_with_input(BenchmarkId::new("iterator", data.len()), &data, |b, data| {
            b.iter(|| sum_iterator(black_box(data)))
        });

        group.bench_with_input(BenchmarkId::new("fold", data.len()), &data, |b, data| {
            b.iter(|| sum_fold(black_box(data)))
        });

        group.finish();
    }

criterion_group!(benches, benchmark_sum_implementations);
criterion_main!(benches);

```

Criterion generates comparative graphs showing which implementation is fastest.

Example: Parameterized Benchmarks

Benchmark across different input sizes:

```

use criterion::{black_box, criterion_group, criterion_main, BenchmarkId, Criterion};

fn sort_benchmark(c: &mut Criterion) {
    let mut group = c.benchmark_group("sort");

    for size in [10, 100, 1000, 10000].iter() {
        group.bench_with_input(BenchmarkId::from_parameter(size), size, |b, &size| {
            let mut data: Vec<i32> = (0..size).rev().collect();
            b.iter(|| {
                let mut d = data.clone();
                d.sort();
                black_box(d);
            });
        });
    }

    group.finish();
}

criterion_group!(benches, sort_benchmark);
criterion_main!(benches);

```

This reveals how performance scales with input size—crucial for understanding algorithmic complexity.

Example: Throughput Measurement

Measure operations per second or bytes per second:

```
use criterion::{black_box, criterion_group, criterion_main, Criterion, Throughput};

fn parse_numbers(data: &str) -> Vec<i32> {
    data.lines()
        .filter_map(|line| line.parse().ok())
        .collect()
}

fn throughput_benchmark(c: &mut Criterion) {
    let data = (0..10000).map(|i| i.to_string()).collect::<Vec<_>>().join("\n");
    let data_bytes = data.len();

    let mut group = c.benchmark_group("parse_throughput");
    group.throughput(Throughput::Bytes(data_bytes as u64));

    group.bench_function("parse", |b| {
        b.iter(|| parse_numbers(black_box(&data)))
    });

    group.finish();
}

criterion_group!(benches, throughput_benchmark);
criterion_main!(benches);
```

Output includes throughput:

```
parse_throughput/parse  time:  [1.2034 ms 1.2156 ms 1.2289 ms]
                        thrpt: [37.428 MiB/s 37.835 MiB/s 38.216 MiB/s]
```

Example: Profiling Integration

Criterion can integrate with profilers:

```
use criterion::{criterion_group, criterion_main, Criterion};
use pprof::criterion::{Output, PProfProfiler};

fn profiled_benchmark(c: &mut Criterion) {
    c.bench_function("expensive_function", |b| {
        b.iter(|| expensive_function())
    });
}

criterion_group! {
    name = benches;
    config = Criterion::default().with_profiler(PProfProfiler::new(100,
        Output::Flamegraph(None)));
}
```

```
    targets = profiled_benchmark
}
criterion_main!(benches);
```

This generates flamegraphs showing where time is spent.

Example: Regression Testing

Criterion saves baseline measurements:

```
# Save current performance as baseline
cargo bench -- --save-baseline master

# Make changes...

# Compare against baseline
cargo bench -- --baseline master
```

Criterion reports whether performance regressed, improved, or stayed the same.

Example: Best Practices for Benchmarking

1. **Benchmark realistic scenarios:** Synthetic microbenchmarks can be misleading
2. **Run benchmarks in isolation:** Close other programs, disable CPU scaling
3. **Use black_box:** Prevent compiler optimizations that wouldn't happen in real code
4. **Warm up before measuring:** Account for cache effects
5. **Benchmark multiple input sizes:** Understand scaling behavior
6. **Track historical performance:** Detect regressions early
7. **Profile before optimizing:** Benchmarks tell you what's slow, profilers tell you why

Summary

This chapter covered comprehensive testing and benchmarking patterns for Rust:

1. **Unit Test Patterns:** Built-in #[test] framework, assertion macros, error/panic testing, RAI^I cleanup
2. **Property-Based Testing:** proptest/quickcheck random input generation, shrinking, invariant verification
3. **Coverage-Guided Fuzzing:** cargo-fuzz/libFuzzer targets, sanitizers, corpus management
4. **Mutation Testing:** cargo-mutants/mutagen workflows to measure assertion strength
5. **Formal Verification:** Kani/Prusti proofs for critical invariants and `unsafe` code
6. **Mock and Stub Patterns:** Trait-based dependency injection, mockall expectations, fake implementations
7. **Integration Testing:** tests/ directory, public API testing, database transactions, HTTP servers

8. Criterion Benchmarking: Statistical analysis, implementation comparison, regression detection, throughput measurement

Key Takeaways: - Type system catches memory bugs; layered test techniques catch logic bugs—use both - Property tests and fuzzers explore input space automatically, surfacing edge-case crashers humans miss - Mutation testing proves your suite fails when behavior changes, preventing false confidence from raw coverage numbers - Formal verification tools offer mathematical guarantees for small, critical components - Mocking and integration tests provide fast feedback on both isolated components and end-to-end flows - Benchmark before optimizing—Criterion’s statistics prevent chasing noise

Testing Strategy: - Unit tests: Cover business logic, panics, and regression scenarios; run on every commit - Property tests: Apply to pure functions/data structures; run in CI with reasonable case limits - Fuzzers: Run locally for long sessions and in CI with `--max_total_time` budgets; persist corpora - Mutation tests: Schedule periodically (e.g., nightly) on core modules to detect assertion gaps - Formal proofs: Target `unsafe`, financial, or safety-critical code paths with `cargo kani` or Prusti - Mocks & integration tests: Exercise external interactions quickly, then confirm workflows end-to-end - Benchmarks: Track hot paths and compare implementations before shipping optimizations

Performance Guidelines: - Unit/property/mock tests run in milliseconds; keep them parallelizable - Fuzzing sessions often run minutes to hours—use timeouts for CI and longer runs locally - Mutation test suites can take minutes per module; narrow scope with `--mutate` filters - Formal proofs may take seconds-minutes depending on bounds; break proofs into small focused functions - Integration tests may need dedicated resources (DB, HTTP servers) and run in seconds - Benchmarks require multiple iterations for significance—expect minutes for complete suites

Best Practices: - Test error paths; use RAI^I for cleanup; group related tests with nested modules - Filter tests via `cargo test pattern`; tag slow ones with `#[ignore]` - Store fuzz corpora and crash reproducers; run fuzzers under sanitizers for maximum signal - Review surviving mutants immediately; they highlight missing assertions - Keep proofs small and composable; verify helper functions before large ones - Track benchmark baselines and compare across commits; use `black_box` to prevent dead-code elimination

CI/CD Integration: - `cargo test`: Run all tests (unit + integration); fail build on any failure - `cargo test --ignored`: Schedule expensive/slow suites nightly - `cargo fuzz run target --max_total_time=60`: Run fuzzers with capped time in CI; archive updated corpora - `cargo mutants --mutate src/core.rs --report`: Periodic mutation runs with HTML reports - `cargo kani proofs/add.rs`: Prove critical invariants before merging changes to safety-critical modules - `cargo bench --save-baseline main / --baseline main`: Track performance regressions - Coverage + lint tooling (`tarpaulin`, `llvm-cov`, `clippy`) + parallel test threads keep feedback fast (use `--test-threads=1` when shared fixtures demand serialization)

Performance Optimization

This chapter explores performance optimization: profiling to find bottlenecks, allocation reduction techniques, cache-friendly data structures, zero-cost abstractions, and compiler optimizations for maximum performance.

Pattern 1: Profiling Strategies

Problem: Guessing at bottlenecks leads to wasted optimization—without data you tweak the wrong code and still miss production hotspots.

Solution: Profile first: perf/cargo-flamegraph/Instruments for CPU, heaptrack/dhat for allocations, Criterion for microbenchmarks; always build in release with symbols and re-measure after each change.

Why It Matters: Profiling exposes the surprise 20% of code that burns 80% of time, shows call stacks responsible, and proves whether an optimization actually helped.

Use Cases: Locating hot functions, tracking allocations, validating optimizations, investigating production regressions, and comparing algorithm variants or scaling behavior.

Example: Which bottleneck

```
//=====
// Which is the bottleneck?
//=====

fn process_data(items: Vec<String>) -> Vec<String> {
    items.iter()
        .filter(|s| validate(s))           // Is it validation?
        .map(|s| transform(s))            // Is it transformation?
        .collect()                        // Is it allocation?
}

fn validate(s: &str) -> bool {
    s.len() > 10 && s.chars().all(|c| c.is_alphanumeric())
}

fn transform(s: &str) -> String {
    s.to_uppercase()
}
```

You might guess `transform` is slow because it allocates. Or maybe `validate` because it iterates characters. Only profiling tells you the truth. Maybe `collect()` dominates because the vector is huge. Or maybe `validate` is called millions of times with tiny strings, making the overhead of `chars()` matter.

Example: CPU Profiling with perf

On Linux, `perf` is the gold standard for CPU profiling:

```
# Build with debug symbols in release mode
# Add to Cargo.toml:
# [profile.release]
# debug = true

cargo build --release

# Record profiling data
```

```
perf record --call-graph dwarf ./target/release/myapp

# View report
perf report

# Generate flamegraph
perf script | stackcollapse-perf.pl | flamegraph.pl > flame.svg
```

The flamegraph visualizes where time is spent. Wide bars are expensive functions. The stack shows the call chain that led there.

Reading a flamegraph: - **X-axis**: Percentage of samples (wider = more time) - **Y-axis**: Call stack depth (bottom = entry point, top = current function) - **Color**: Meaningless, just for visibility

Look for wide bars at the top—those are your bottlenecks.

Example: Using cargo-flamegraph

For easier flamegraph generation:

```
# Install
cargo install flamegraph

# Generate flamegraph (requires root on Linux)
cargo flamegraph

# Or without root (less accurate)
cargo flamegraph --dev
```

This generates `flamegraph.svg` automatically.

Example: Profiling with Instruments (macOS)

On macOS, use Instruments:

```
# Build with debug symbols
cargo build --release

# Open in Instruments
instruments -t "Time Profiler" ./target/release/myapp
```

Instruments provides a GUI for exploring hotspots, viewing call trees, and drilling into specific functions.

Example: Profiling in Code with Benchmarks

Criterion benchmarks provide detailed performance data:

```
use criterion::{black_box, criterion_group, criterion_main, Criterion};

fn process_data(items: &[String]) -> Vec<String> {
```

```

    items.iter()
        .filter(|s| validate(s))
        .map(|s| transform(s))
        .collect()
}

fn validate(s: &str) -> bool {
    s.len() > 10 && s.chars().all(|c| c.is_alphanumeric())
}

fn transform(s: &str) -> String {
    s.to_uppercase()
}

fn benchmark(c: &mut Criterion) {
    let data: Vec<String> = (0..1000)
        .map(|i| format!("item_number_{}", i))
        .collect();

    c.bench_function("process_data", |b| {
        b.iter(|| process_data(black_box(&data)))
    });

    // Profile individual components
    c.bench_function("validate", |b| {
        b.iter(|| {
            for item in &data {
                black_box(validate(black_box(item)));
            }
        })
    });

    c.bench_function("transform", |b| {
        b.iter(|| {
            for item in &data {
                black_box(transform(black_box(item)));
            }
        })
    });
}

criterion_group!(benches, benchmark);
criterion_main!(benches);

```

Run with `cargo bench`. Criterion shows which component is slow.

Example: Memory Profiling with Valgrind

Find memory allocations and leaks:

```

# Install valgrind
# Ubuntu: sudo apt install valgrind

```

```
# macOS: brew install valgrind

# Profile memory usage
valgrind --tool=massif ./target/release/myapp

# Visualize with ms_print
ms_print massif.out.* > massif.txt
```

Or use `heaptrack` for more detailed allocation tracking:

```
# Install heaptrack
# Ubuntu: sudo apt install heaptrack

# Profile
heaptrack ./target/release/myapp

# View results
heaptrack_gui heaptrack.myapp.*.gz
```

Example: Profiling Allocations in Rust

Use `dhat` for Rust-specific allocation profiling:

```
// Add to Cargo.toml:
// [dependencies]
// dhat = "0.3"

#[cfg(feature = "dhat-heap")]
#[global_allocator]
static ALLOC: dhat::Alloc = dhat::Alloc;

fn main() {
    #[cfg(feature = "dhat-heap")]
    let _profiler = dhat::Profiler::new_heap();

    // Your code here
    run_application();
}
```

Run with:

```
cargo run --features dhat-heap --release
```

This generates `dhat-heap.json`, viewable in Firefox's DHAT viewer.

Example: Micro-Benchmarking Best Practices

When benchmarking specific code:

```

use criterion::black_box, Criterion;

fn benchmark_alternatives(c: &mut Criterion) {
    let data: Vec<i32> = (0..10000).collect();

    c.bench_function("sum_loop", |b| {
        b.iter(|| {
            let mut sum = 0;
            for &x in black_box(&data) {
                sum += black_box(x);
            }
            black_box(sum)
        })
    });

    c.bench_function("sum_iter", |b| {
        b.iter(|| {
            black_box(&data).iter().sum::<i32>()
        })
    });
}

c.bench_function("sum_fold", |b| {
    b.iter(|| {
        black_box(&data).iter().fold(0, |acc, &x| acc + x)
    })
});
}

```

Use `black_box` to prevent the optimizer from eliminating code. Without it, the compiler might optimize away the entire computation.

Pattern 2: Allocation Reduction

Problem: Heap allocations cost mutexes, cache misses, and copies; doing them inside hot loops or string builders dominates runtime and fragments memory.

Solution: Reuse buffers with `clear`, pre-size collections via `with_capacity`, lean on `SmallVec`, `Cow`, arenas, and `String::push_str` instead of repeated `format!`.

Why It Matters: Eliminating redundant allocations routinely yields multi-x speedups, keeps data cache-friendly, and avoids allocator contention.

Use Cases: Parsers, networking buffers, per-frame game loops, log formatting, small temporary collections, and any tight loop building strings or vectors.

Example: Allocation vs Stack Allocation

```

use std::time::Instant;

fn allocation_benchmark() {

```

```

let iterations = 1_000_000;

// Allocating
let start = Instant::now();
for _ in 0..iterations {
    let _v: Vec<i32> = vec![1, 2, 3, 4, 5];
}
println!("Allocating: {:?}", start.elapsed());

// Stack only
let start = Instant::now();
for _ in 0..iterations {
    let _arr = [1, 2, 3, 4, 5];
}
println!("Stack: {:?}", start.elapsed());
}

```

Allocating is often 10-100x slower than stack allocation. Reducing allocations can dramatically improve performance.

Example: Reusing Allocations

Instead of allocating repeatedly, reuse buffers:

```

//=====
// Bad: Allocates every iteration
//=====

fn process_bad(items: &[String]) -> Vec<String> {
    let mut results = Vec::new();
    for item in items {
        let mut buffer = String::new(); // Allocates!
        buffer.push_str("processed_");
        buffer.push_str(item);
        results.push(buffer);
    }
    results
}

//=====
// Good: Reuses buffer
//=====

fn process_good(items: &[String]) -> Vec<String> {
    let mut results = Vec::new();
    let mut buffer = String::new(); // Allocate once
    for item in items {
        buffer.clear(); // Reuse allocation
        buffer.push_str("processed_");
        buffer.push_str(item);
        results.push(buffer.clone()); // Still allocates, but see next example
    }
    results
}

```

```

//=====
// Better: Pre-allocate and avoid unnecessary work
//=====

fn process_better(items: &[String]) -> Vec<String> {
    let mut results = Vec::with_capacity(items.len()); // Pre-allocate
    for item in items {
        let processed = format!("processed_{}", item); // One allocation
        results.push(processed);
    }
    results
}

//=====
// Best for this case: Use iterators
//=====

fn process_best(items: &[String]) -> Vec<String> {
    items
        .iter()
        .map(|item| format!("processed_{}", item))
        .collect()
}

```

Example: SmallVec: Stack-Allocated Small Collections

`SmallVec` stores small collections on the stack:

```

// Add to Cargo.toml:
// smallvec = "1.11"

use smallvec::SmallVec;

//=====
// Stores up to 4 elements on stack, spills to heap if larger
//=====

type SmallVec4<T> = SmallVec<[T; 4]>;

fn process_items(items: &[i32]) -> SmallVec4<i32> {
    let mut result = SmallVec4::new();
    for &item in items {
        if item % 2 == 0 {
            result.push(item);
        }
        if result.len() >= 4 {
            break;
        }
    }
    result
}

//=====

```

```
// If result has ≤4 elements, no heap allocation!
//=====
```

Use `SmallVec` when collections are usually small. The stack storage avoids allocation in the common case.

Example: Cow: Clone-On-Write

`Cow` defers allocation until mutation:

```
use std::borrow::Cow;

fn process_string(input: &str) -> Cow<str> {
    if input.contains("bad") {
        // Must modify – allocates
        Cow::Owned(input.replace("bad", "good"))
    } else {
        // No modification needed – no allocation
        Cow::Borrowed(input)
    }
}

fn example() {
    let s1 = "good text";
    let s2 = "bad text";

    let r1 = process_string(s1); // No allocation
    let r2 = process_string(s2); // Allocates

    println!("{} , {}", r1, r2);
}
```

This pattern is common in APIs that sometimes need to modify data and sometimes don't.

Example: Arena Allocation

Arena as batch allocations for better performance:

```
//=====
// Add to Cargo.toml:
//=====
// typed-arena = "2.0"

use typed_arena::Arena;

struct Node<'a> {
    value: i32,
    children: Vec<&'a Node<'a>>,
}

fn build_tree<'a>(arena: &'a Arena<Node<'a>>) -> &'a Node<'a> {
```

```

let child1 = arena.alloc(Node {
    value: 1,
    children: vec![],
});

let child2 = arena.alloc(Node {
    value: 2,
    children: vec![],
});

arena.alloc(Node {
    value: 0,
    children: vec![child1, child2],
})
}

fn example() {
    let arena = Arena::new();
    let tree = build_tree(&arena);

    // All nodes allocated from arena
    // Deallocated together when arena drops
}

```

Arenas are fast because: 1. Allocation is a simple pointer bump 2. Individual deallocation is free (no-op) 3. Bulk deallocation is fast (drop the arena)

Example: String Interning

Deduplicate strings to save memory:

```

use std::collections::HashMap;

struct StringInterner {
    strings: HashMap<String, usize>,
    reverse: Vec<String>,
}

impl StringInterner {
    fn new() -> Self {
        StringInterner {
            strings: HashMap::new(),
            reverse: Vec::new(),
        }
    }

    fn intern(&mut self, s: &str) -> usize {
        if let Some(&id) = self.strings.get(s) {
            id
        } else {
            let id = self.reverse.len();
            self.reverse.push(s.to_string());
            self.strings.insert(s.to_string(), id);
            id
        }
    }
}

```

```

        self.strings.insert(s.to_string(), id);
    id
    }

fn get(&self, id: usize) -> &str {
    &self.reverse[id]
}

fn example() {
    let mut interner = StringInterner::new();

    // Same strings map to same ID
    let id1 = interner.intern("hello");
    let id2 = interner.intern("hello");

    assert_eq!(id1, id2); // No duplicate allocation

    println!("{}", interner.get(id1));
}

```

Use interning when you have many duplicate strings (like identifiers in a compiler).

Pattern 3: Cache-Friendly Data Structures

Problem: Pointer-chasing data structures thrash caches—RAM misses are $\sim 100\times$ slower than L1 hits and false sharing stalls multi-threaded code.

Solution: Favor contiguous storage ([Vec](#), SoA layouts, arenas), group hot fields together, pad or align to avoid false sharing, and prefetch predictable strides.

Why It Matters: Cache-friendly layouts let hardware prefetch and keep hot data in L1, yielding 2–10 \times faster loops with less coherency traffic.

Use Cases: ECS/game data, parsers/ASTs, graph and numerical kernels, big data scans, multi-threaded counters, and storage engines choosing row vs column layouts.

Example: Array-of-Structs vs Struct-of-Arrays

```

//=====
// Array of Structs (AoS) - bad for cache
//=====

struct ParticleAoS {
    x: f32,
    y: f32,
    z: f32,
    vx: f32,
    vy: f32,
    vz: f32,
}

```

```

fn update_positions_aos(particles: &mut [ParticleAoS], dt: f32) {
    for p in particles {
        // Loads entire particle (24 bytes)
        // But we only need x, y, z (12 bytes)
        // Wastes cache bandwidth
        p.x += p.vx * dt;
        p.y += p.vy * dt;
        p.z += p.vz * dt;
    }
}

//=====
// Struct of Arrays (SoA) - cache-friendly
//=====

struct ParticlesSoA {
    x: Vec<f32>,
    y: Vec<f32>,
    z: Vec<f32>,
    vx: Vec<f32>,
    vy: Vec<f32>,
    vz: Vec<f32>,
}

fn update_positions_soa(particles: &mut ParticlesSoA, dt: f32) {
    for i in 0..particles.x.len() {
        // Loads contiguous x, y, z values
        // Much better cache usage
        particles.x[i] += particles.vx[i] * dt;
        particles.y[i] += particles.vy[i] * dt;
        particles.z[i] += particles.vz[i] * dt;
    }
}

```

SoA can be 2-3x faster for this access pattern because it uses cache lines efficiently.

Example: Cache Line Awareness

Cache lines are typically 64 bytes. Accessing any byte in a cache line loads the entire line:

```

#[repr(C, align(64))]
struct CacheLineAligned {
    value: i64,
    padding: [u8; 56],
}

//=====
// Bad: False sharing
//=====

struct CounterBad {
    thread1_counter: i64, // Same cache line
    thread2_counter: i64, // Same cache line
}

```

```

}

//=====
// Good: No false sharing
//=====

#[repr(C, align(64))]
struct CounterGood {
    thread1_counter: i64,
    _padding: [u8; 56],
}

#[repr(C, align(64))]
struct CounterGood2 {
    thread2_counter: i64,
    _padding: [u8; 56],
}

```

False sharing occurs when two threads write to different variables in the same cache line, causing cache invalidation and performance degradation.

Example: Prefetching and Sequential Access

Sequential access is dramatically faster than random access:

```

use std::time::Instant;

fn sequential_access(data: &[i32]) -> i64 {
    let mut sum = 0i64;
    for &x in data {
        sum += x as i64;
    }
    sum
}

fn random_access(data: &[i32], indices: &[usize]) -> i64 {
    let mut sum = 0i64;
    for &idx in indices {
        sum += data[idx] as i64;
    }
    sum
}

fn benchmark() {
    let data: Vec<i32> = (0..1_000_000).collect();
    let mut indices: Vec<usize> = (0..data.len()).collect();

    // Shuffle for random access
    use rand::seq::SliceRandom;
    indices.shuffle(&mut rand::thread_rng());

    let start = Instant::now();
    let sum1 = sequential_access(&data);

```

```

    println!("Sequential: {:?}", start.elapsed());

    let start = Instant::now();
    let sum2 = random_access(&data, &indices);
    println!("Random: {:?}", start.elapsed());

    // Random access is often 5-10x slower!
}

```

Design data structures for sequential access when possible.

Example: Linked Lists vs Vectors

Linked lists are almost always slower than vectors due to poor cache behavior:

```

//=====
// Bad: Linked list
//=====
struct LinkedList<T> {
    head: Option<Box<Node<T>>,
}

struct Node<T> {
    value: T,
    next: Option<Box<Node<T>>,
}

//=====
// Good: Vector
//=====
struct VecList<T> {
    items: Vec<T>,
}

//=====
// Iterating a vector loads contiguous cache lines
//=====
// Iterating a linked list causes a cache miss per node

```

Use vectors unless you need O(1) insertion in the middle (rare).

HashMap Optimization

```

use std::collections::HashMap;
use rustc_hash::FxHashMap; // Faster hash for integer keys

fn compare_hashmaps() {
    let mut std_map = HashMap::new();
    let mut fx_map = FxHashMap::default();

    // FxHashMap is faster for integer keys
}

```

```

    for i in 0..10000 {
        std_map.insert(i, i * 2);
        fx_map.insert(i, i * 2);
    }

    // FxHashMap: ~30% faster for integer keys
    // But less secure (predictable hashes)
}

//=====
// Pre-size HashMaps
//=====

fn presized_hashmap() {
    let items = vec![(1, "a"), (2, "b"), (3, "c")];

    // Bad: Allocates multiple times as it grows
    let mut map1 = HashMap::new();
    for (k, v) in &items {
        map1.insert(*k, *v);
    }

    // Good: Allocates once
    let mut map2 = HashMap::with_capacity(items.len());
    for (k, v) in &items {
        map2.insert(*k, *v);
    }

    // Better: Use FromIterator
    let map3: HashMap<, > = items.iter().copied().collect();
}

```

Pattern 4: Zero-Cost Abstractions

Problem: Abstractions look expensive—iterator chains, generics, and newtypes seem slower than hand-written loops or raw types.

Solution: Trust the optimizer: iterators inline to the same machine code, generics monomorphize per type, `#[inline]` removes tiny call overhead, and newtypes share representation with their inner type.

Why It Matters: Zero-cost abstractions let you write clear, reusable APIs without leaving performance on the table; release builds routinely match or beat manual code.

Use Cases: Iterator-heavy pipelines, generic data structures, type-safe ID wrappers, compile-time computation (const fn/generics), and layered DSL-style APIs.

Example: Understanding Branch Misprediction

```

use std::time::Instant;

fn with_unpredictable_branch(data: &[i32]) -> i32 {
    let mut sum = 0;

```

```

for &x in data {
    // Unpredictable - depends on data
    if x % 2 == 0 {
        sum += x;
    }
}
sum
}

fn without_branch(data: &[i32]) -> i32 {
let mut sum = 0;
for &x in data {
    // Branchless - uses arithmetic
    sum += x * (x % 2 == 0) as i32;
}
sum
}

fn benchmark() {
// Random data - unpredictable branches
let random_data: Vec<i32> = (0..1_000_000)
    .map(|_| rand::random::<i32>())
    .collect();

let start = Instant::now();
let sum1 = with_unpredictable_branch(&random_data);
println!("With branch: {:?}", start.elapsed());

let start = Instant::now();
let sum2 = without_branch(&random_data);
println!("Branchless: {:?}", start.elapsed());

// Branchless can be 2x faster with random data
}

```

Example: Sorting for Branch Prediction

```

fn sum_if_positive_unsorted(data: &[i32]) -> i32 {
let mut sum = 0;
for &x in data {
    if x > 0 { // Unpredictable branches
        sum += x;
    }
}
sum
}

fn sum_if_positive_sorted(data: &mut [i32]) -> i32 {
    data.sort_unstable(); // Cost: O(n log n)

    let mut sum = 0;
    for &x in data {

```

```

        if x > 0 { // Predictable after sorting
            sum += x;
        }
    }
    sum
}

// If you iterate many times, sorting once can be faster

```

Example: Branch-Free Code with Bitwise Operations

```

//=====
// With branch
//=====

fn max_with_branch(a: i32, b: i32) -> i32 {
    if a > b { a } else { b }
}

//=====
// Branchless
//=====

fn max_branchless(a: i32, b: i32) -> i32 {
    let diff = a - b;
    let sign = diff >> 31; // -1 if negative, 0 if positive
    a - (diff & sign)
}

//=====
// Or use LLVM's select (compiler does this optimization)
//=====

fn max_select(a: i32, b: i32) -> i32 {
    if a > b { a } else { b } // LLVM converts to select instruction
}

```

The compiler often optimizes simple `if` expressions to branchless code automatically.

Example: Likely and Unlikely Hints

```

//=====
// Unstable feature - requires nightly
//=====

#![feature(core_intrinsics)]

use std::intrinsics::{likely, unlikely};

fn process_with_hints(data: &[i32]) -> i32 {
    let mut sum = 0;
    for &x in data {
        unsafe {
            if likely(x > 0) { // Hint: usually true
                sum += x;
            }
        }
    }
    sum
}

```

```

        }
    }
    sum
}

//=====
// Stable alternative using cold
//=====

#[cold]
#[inline(never)]
fn handle_error() {
    eprintln!("Error occurred!");
}

fn process_stable(x: i32) {
    if x < 0 {
        handle_error(); // Compiler knows this is rare
    }
}

```

The `#[cold]` attribute tells the compiler this code is rarely executed, improving branch prediction for the common path.

Example: Pattern Matching Optimization

```

enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(u8, u8, u8),
}

//=====
// Less optimizable - many possible branches
//=====

fn process_message_bad(msg: &Message) -> String {
    match msg {
        Message::Quit => "quit".to_string(),
        Message::Move { x, y } => format!("move {} {}", x, y),
        Message::Write(s) => s.clone(),
        Message::ChangeColor(r, g, b) => format!("color {} {} {}", r, g, b),
    }
}

//=====
// More optimizable - common case first
//=====

fn process_message_good(msg: &Message) -> String {
    match msg {
        Message::Write(s) => s.clone(), // Assume this is most common
        Message::Move { x, y } => format!("move {} {}", x, y),
    }
}

```

```

        Message::ChangeColor(r, g, b) => format!("color {} {} {}", r, g, b),
        Message::Quit => "quit".to_string(),
    }
}

```

Put common cases first in match statements to improve branch prediction.

Pattern 5: Compiler Optimizations

Problem: Leaving builds at debug defaults or generic CPU targets forfeits huge speedups; many teams never flip LTO, PGO, or codegen knobs because the impact seems opaque.

Solution: Ship release builds with `opt-level=3`, enable LTO (and PGO when feasible), target the actual CPU (`target-cpu=native`), reduce `codegen-units` for deeper optimization, and tune for size with `opt-level="z"/panic="abort"` when needed.

Why It Matters: The right flags routinely deliver 10–30× faster binaries or much smaller artifacts by letting LLVM inline across crates, specialize for hot paths, and emit SIMD instructions your hardware already supports.

Use Cases: Production binaries, benchmarking harnesses, SIMD-heavy workloads, embedded/WASM targets chasing size, and CI pipelines that produce optimized artifacts via PGO/LTO combinations.

Example: Const Functions

Const functions execute at compile time when possible:

```

const fn factorial(n: u32) -> u32 {
    match n {
        0 => 1,
        _ => n * factorial(n - 1),
    }
}

const FACTORIAL_10: u32 = factorial(10); // Computed at compile time!

fn example() {
    println!("{}", FACTORIAL_10); // Just loads the constant
}

```

Example: Const Generics for Compile-Time Values

```

struct Matrix<const N: usize> {
    data: [[f64; N]; N],
}

impl<const N: usize> Matrix<N> {
    const fn zeros() -> Self {
        Matrix {
            data: [[0.0; N]; N],
        }
    }
}

```

```

        }

    }

    const fn identity() -> Self {
        let mut data = [[0.0; N]; N];
        let mut i = 0;
        while i < N {
            data[i][i] = 1.0;
            i += 1;
        }
        Matrix { data }
    }

const IDENTITY_4X4: Matrix<4> = Matrix::identity();

fn example() {
    // Matrix size known at compile time
    // Enables better optimization
    let m = Matrix::<3>::zeros();
}

```

Example: Lookup Tables

Pre-compute lookup tables at compile time:

```

const fn generate_sin_table() -> [f64; 360] {
    let mut table = [0.0; 360];
    let mut i = 0;
    while i < 360 {
        // Simplified – actual sin computation would use series expansion
        table[i] = 0.0; // Placeholder
        i += 1;
    }
    table
}

const SIN_TABLE: [f64; 360] = generate_sin_table();

fn fast_sin(degrees: usize) -> f64 {
    SIN_TABLE[degrees % 360] // O(1) lookup vs expensive calculation
}

```

Example: Static Assertions

Verify invariants at compile time:

```

const fn is_power_of_two(n: usize) -> bool {
    n != 0 && (n & (n - 1)) == 0
}

```

```

struct RingBuffer<T, const N: usize> {
    data: [Option<T>; N],
    head: usize,
}

impl<T, const N: usize> RingBuffer<T, N> {
    const VALID_SIZE: () = assert!(is_power_of_two(N), "Size must be power of two");

    fn new() -> Self {
        let _ = Self::VALID_SIZE; // Force compile-time check
        RingBuffer {
            data: std::array::from_fn(|_| None),
            head: 0,
        }
    }
}

// This won't compile:
// let buffer = RingBuffer::<i32, 7>::new();

// This compiles:
let buffer = RingBuffer::<i32, 8>::new();

```

Example: Build-Time Code Generation

Use build scripts to generate code:

```

//=====
// build.rs
//=====

fn main() {
    println!("cargo:rerun-if-changed=build.rs");

    let out_dir = std::env::var("OUT_DIR").unwrap();
    let dest_path = std::path::Path::new(&out_dir).join("generated.rs");

    // Generate lookup table
    let mut code = String::from("const PRIMES: &[u32] = &[\n");
    for i in 2..10000 {
        if is_prime(i) {
            code.push_str(&format!("    {},\n", i));
        }
    }
    code.push_str("];\n");

    std::fs::write(dest_path, code).unwrap();
}

fn is_prime(n: u32) -> bool {
    if n < 2 { return false; }
    for i in 2..=(n as f64).sqrt() as u32 {
        if n % i == 0 { return false; }
    }
}

```

```
    }
    true
}
```

```
// src/lib.rs
include!(concat!(env!("OUT_DIR"), "/generated.rs"));

fn is_prime_fast(n: u32) -> bool {
    PRIMES.binary_search(&n).is_ok()
}
```

Example: Compile-Time String Processing

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

const fn const_concat_len(s1: &str, s2: &str) -> usize {
    s1.len() + s2.len()
}

const HELLO_LEN: usize = const_strlen("Hello, World!");

fn example() {
    // Length computed at compile time
    let mut buffer = [0u8; HELLO_LEN];
}
```

Type-Level Computation

Use the type system for compile-time computation:

```
use std::marker::PhantomData;

struct Succ<N>(PhantomData<N>);
struct Zero;

type One = Succ<Zero>;
type Two = Succ<One>;
type Three = Succ<Two>;

trait NatNum {
    const VALUE: usize;
}

impl NatNum for Zero {
    const VALUE: usize = 0;
}

impl<N: NatNum> NatNum for Succ<N> {
```

```

    const VALUE: usize = N::VALUE + 1;
}

fn example() {
    assert_eq!(Three::VALUE, 3); // Computed at compile time
}

```

Advanced Optimization Techniques

SIMD (Single Instruction Multiple Data)

Process multiple values simultaneously:

```

// Requires nightly and target features
// Add to .cargo/config.toml:
// [build]
// rustflags = ["-C", "target-cpu=native"]

#[cfg(target_arch = "x86_64")]
use std::arch::x86_64::*;

#[inline]
fn add_arrays_scalar(a: &[f32], b: &[f32], result: &mut [f32]) {
    for i in 0..a.len() {
        result[i] = a[i] + b[i];
    }
}

#[cfg(target_arch = "x86_64")]
#[inline]
unsafe fn add_arrays_simd(a: &[f32], b: &[f32], result: &mut [f32]) {
    let chunks = a.len() / 8;

    for i in 0..chunks {
        let offset = i * 8;

        // Load 8 floats at once
        let a_vec = _mm256_loadu_ps(a.as_ptr().add(offset));
        let b_vec = _mm256_loadu_ps(b.as_ptr().add(offset));

        // Add 8 floats in one instruction
        let result_vec = _mm256_add_ps(a_vec, b_vec);

        // Store 8 floats at once
        _mm256_storeu_ps(result.as_mut_ptr().add(offset), result_vec);
    }

    // Handle remainder
    for i in (chunks * 8)..a.len() {
        result[i] = a[i] + b[i];
    }
}

```

```
}
```

```
// SIMD can be 4-8x faster for this operation
```

Inline Assembly

For absolute control (rarely needed):

```
#![feature(asm)]
```

```
unsafe fn cpuid(eax: u32) -> (u32, u32, u32, u32) {
    let mut ebx: u32;
    let mut ecx: u32;
    let mut edx: u32;

    std::arch::asm!(
        "cpuid",
        inout("eax") eax,
        out("ebx") ebx,
        out("ecx") ecx,
        out("edx") edx,
    );
    (eax, ebx, ecx, edx)
}
```

Link-Time Optimization

Enable in Cargo.toml:

```
[profile.release]
lto = "fat"          # Full LTO
codegen-units = 1    # Single codegen unit for better optimization
```

LTO enables cross-crate inlining and optimization, often yielding 10-20% speedup at the cost of longer compile times.

Summary

This chapter covered performance optimization patterns for maximizing Rust code performance:

- 1. Profiling Strategies:** CPU profiling (perf, flamegraph), memory profiling (dhat, valgrind), Criterion benchmarks
- 2. Allocation Reduction:** Reuse buffers, SmallVec, Cow, pre-allocation, arena allocation
- 3. Cache-Friendly Data Structures:** Contiguous memory, struct-of-arrays, arena allocation, inline data
- 4. Zero-Cost Abstractions:** Iterators = loops, generics monomorphize, inline functions, newtype pattern
- 5. Compiler Optimizations:** Release mode, LTO, PGO, target-cpu=native, codegen-units=1

Key Takeaways: - Measure first: intuition about bottlenecks usually wrong, profiling reveals truth - Allocation reduction: 2-10x speedup by reusing buffers, pre-allocating, using SmallVec - Cache-friendly: cache miss = 100x slower than hit, contiguous memory = prefetching - Zero-cost abstractions: iterators as fast as loops, generics free, newtypes free - Compiler optimization: release 10-100x faster than debug, LTO + PGO + target-cpu = maximum speed

Optimization Workflow: 1. Profile to find hotspots (perf, flamegraph, Criterion) 2. Understand why slow (allocations? cache misses? branches?) 3. Optimize (reduce allocations, improve locality, eliminate branches) 4. Verify with benchmarks (did it actually help?) 5. Repeat for next hotspot

Profiling Commands:

```
# Flamegraph (all platforms)
cargo install flamegraph
cargo flamegraph

# Criterion benchmarks
cargo bench

# Memory profiling with dhat
cargo run --features dhat-heap --release

# Release mode with debug symbols
[profile.release]
debug = true
```

Performance Guidelines: - Allocation: 10-100x slower than stack, mutex contention in multi-threaded - Cache: L1 = 1 cycle, L2 = 10 cycles, L3 = 40 cycles, RAM = 200 cycles - Branch misprediction: 10-20 cycles penalty - Function call: inlined = free, not inlined = ~5 cycles - SIMD: 4-8x speedup for data-parallel operations

Anti-Patterns: - Premature optimization (measure first!) - Optimizing cold code (focus on hotspots) - Sacrificing readability for negligible gains (5% not worth complexity) - Ignoring allocations (often biggest win) - Not benchmarking changes (did it actually help?) - Using debug mode for benchmarks (10-100x slower) - Assuming cache doesn't matter (it's the bottleneck)

Optimization Priority (by typical impact): 1. Algorithmic complexity ($O(N^2) \rightarrow O(N \log N)$) 2. Reduce allocations (reuse buffers, SmallVec) 3. Cache-friendly data layout (contiguous, SoA) 4. Compiler flags (release, LTO, target-cpu) 5. Branch prediction (predictable branches) 6. Micro-optimizations (last resort, measure first)

Embedded & Real-Time Patterns

Rust on microcontrollers, bare metal SoCs, and hard real-time workloads demands strict control over memory, timing, and side effects. Without an OS, you must replace the standard library with `#![no_std]`, carefully manage interrupts, and keep predictable execution. This chapter assembles patterns that combine HAL abstractions, interrupt coordination, and deterministic scheduling so experienced Rustaceans can bring the language's safety guarantees to embedded constraints.

Modern embedded stacks typically follow a split architecture: 1. Board Support Package (BSP) initializes clocks, peripherals, and pin mappings. 2. Hardware Abstraction Layer (HAL) provides portable traits (`embedded-hal`, `embedded-io`). 3. Application logic plugs drivers together using zero-allocation data structures, lightweight schedulers, and strict error handling.

Development Setups: Raspberry Pi vs. STM32

Working Directly on a Raspberry Pi

Linux-based SBCs like Raspberry Pi let you run `cargo` natively, which is great for rapid iteration before jumping to bare metal. - Install Rust with `rustup` on the Pi (`curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh`), then add needed targets such as `armv7-unknown-linux-gnueabihf`. - Use crates like `rppal` or `linux-embedded-hal` to access GPIO, SPI, and I2C without needing `no_std`. - For deterministic services, pin tasks to cores with `taskset` and use `systemd` units to manage startup; for realtime kernels enable `PREEMPT_RT`. - Deploy by copying binaries or using `cargo run --release --target armv7-unknown-linux-gnueabihf`, then supervise with `systemd`, `tmux`, or container runtimes. - You can mock out HAL traits on the Pi while your final firmware targets a microcontroller—this chapter's HAL patterns show how to keep code portable.

Cross-Compiling for STM32 Boards

Bare-metal STM32 development needs a `no_std` build, cross toolchain, and a flashing/debug story. - Install the `thumbv7em-none-eabihf` (or appropriate) target with `rustup target add thumbv7em-none-eabihf`. - Use `probe-rs` (`cargo install probe-run`) or `openocd + gdb` for flashing/debug; `cargo embed` automates logging via RTT/defmt. - HAL/BSP crates (e.g., `stm32f4xx-hal`, `stm32h7xx-hal`) provide clock setup and driver scaffolding—mirror your board layout there. - Configure `.cargo/config.toml` with runner `probe-run --chip STM32F401RETx` for seamless `cargo run --release`. - For CI, leverage `cargo xtask` scripts or `just` recipes to build both host-mock tests and firmware artifacts, ensuring determinism with `--locked --target`.

Pattern 1: Layered HAL Drivers

- **Problem:** Directly touching vendor registers makes code brittle and untestable. Porting across MCUs or even board revisions forces a rewrite.
- **Solution:** Build drivers against `embedded-hal`-style traits and keep board-specific code isolated in a BSP. This separates volatile register fiddling from reusable business logic.
- **Why It Matters:** HAL traits allow unit testing on the host, replaceable mocks, and reuse across Cortex-M, RISC-V, or even Linux-based targets.
- **Use Cases:** Sensor drivers, communication stacks, PWM motor control, portable display drivers.

Examples

Example: Board Support Layer

The BSP owns the device crate (here `stm32f4xx-hal`) and exports initialized peripherals using HAL traits.

```

#![no_std]
#![no_main]

use stm32f4xx_hal::{pac, prelude::*, timer::CounterHz};
use embedded_hal::digital::v2::OutputPin;

pub struct Board {
    pub led: impl OutputPin<Error = core::convert::Infallible>,
    pub timer: CounterHz<'static, pac::TIM2>,
}

pub fn init() -> Board {
    let dp = pac::Peripherals::take().unwrap();
    let rcc = dp.RCC.constrain();
    let clocks = rcc.cfgr.sysclk(84.MHz()).freeze();
    let gpioa = dp.GPIOA.split();

    let mut led = gpioa.pa5.into_push_pull_output();
    led.set_low().ok();

    let mut timer = dp.TIM2.counter_hz(&clocks);
    timer.start(1.Hz()).unwrap();

    Board { led, timer }
}

```

Example: Driver Consuming HAL Traits

Application code depends only on traits, so it can be tested with mocks.

```

use embedded_hal::digital::v2::OutputPin;
use embedded_hal::timer::CountDown;
use nb::block;

pub struct Heartbeat<P, T> {
    led: P,
    timer: T,
}

impl<P, T> Heartbeat<P, T>
where
    P: OutputPin<Error = core::convert::Infallible>,
    T: CountDown,
{
    pub fn new(led: P, timer: T) -> Self {
        Self { led, timer }
    }

    pub fn spin(&mut self) -> ! {
        loop {
            self.led.set_high().ok();

```

```
        block!(<self>.timer.wait()).ok();
        <self>.led.set_low().ok();
        block!(<self>.timer.wait()).ok();
    }
}
}
```

Example: Sensor Driver Abstracted Over SPI + Delay

Complex peripherals (IMUs, radios) often need multiple traits. By expressing the driver in terms of [embedded-hal](#) traits, the same code runs on STM32, Nordic, or on-host mocks.

```
use embedded_hal::digital::v2::OutputPin;
use embedded_hal::blocking::delay::DelayUs;
use embedded_hal::blocking::spi::Transfer;

pub struct ImuDriver<SPI, CS, DELAY> {
    spi: SPI,
    cs: CS,
    delay: DELAY,
}

impl<SPI, CS, DELAY> ImuDriver<SPI, CS, DELAY>
where
    SPI: Transfer<u8>,
    CS: OutputPin<Error = core::convert::Infallible>,
    DELAY: DelayUs<u16>,
{
    pub fn new(spi: SPI, cs: CS, delay: DELAY) -> Self {
        Self { spi, cs, delay }
    }

    pub fn read_whoami(&mut self) -> Result<u8, SPI::Error> {
        let mut buf = [0x75, 0];
        self.cs.set_low().ok()?;
        self.spi.transfer(&mut buf)?;
        self.cs.set_high().ok()?;
        Ok(buf[1])
    }

    pub fn configure(&mut self) -> Result<(), SPI::Error> {
        self.write_reg(0x6B, 0x00)?;
        self.delay.delay_us(50);
        self.write_reg(0x1C, 0x10)?;
        Ok(())
    }

    fn write_reg(&mut self, reg: u8, value: u8) -> Result<(), SPI::Error> {
        let mut buf = [reg & 0x7F, value];
        self.cs.set_low().ok()?;
        self.spi.transfer(&mut buf)?;
        self.cs.set_high().ok()?;
    }
}
```

```
    Ok(())
}
}
```

Example: Raspberry Pi HAL Wrapper

On boards like the Raspberry Pi 4 running Linux, you can still implement `embedded-hal` traits by delegating to crates such as `rppal`. That lets your application reuse the same drivers as an STM32 target.

```
use embedded_hal::digital::v2::OutputPin;
use rppal::gpio::{Gpio, OutputPin as PiPin};

pub struct PiLed {
    pin: PiPin,
}

impl PiLed {
    pub fn new(pin_id: u8) -> Self {
        let pin = Gpio::new().unwrap().get(pin_id).unwrap().into_output();
        Self { pin }
    }
}

impl OutputPin for PiLed {
    type Error = core::convert::Infallible;

    fn set_low(&mut self) -> Result<(), Self::Error> {
        self.pin.set_low();
        Ok(())
    }

    fn set_high(&mut self) -> Result<(), Self::Error> {
        self.pin.set_high();
        Ok(())
    }
}
```

You can now feed `PiLed` into the `Heartbeat` example and run the exact same logic on a Raspberry Pi for desktop prototyping.

Testing tip: replace `P` and `T` with fake implementations using `std` timers to coverage-test logic on the desktop.

Pattern 2: Static Allocation & Zero-Copy Buffers

- **Problem:** Dynamic allocation (`Vec`, `Box`) is often unavailable or banned in hard real-time systems. Yet peripherals require queues for DMA, networking, or logging.
- **Solution:** Use `heapless`, `arrayvec`, or custom `static mut` buffers guarded by safe wrappers. Favor compile-time capacity, placement in specific memory sections, and DMA-friendly alignment.

- **Why It Matters:** Static buffers make timing predictable and avoid allocator fragmentation. They also ease certification (MISRA, DO-178C) where dynamic memory is disallowed.
- **Use Cases:** UART ring buffers, telemetry queues, sensor fusion windows, DMA descriptors.

Examples

Example: Lock-Free Telemetry Queue

`heapless::spsc::Queue` provides a single-producer single-consumer buffer without heap allocations.

```
use heapless::spsc::Queue;
use core::sync::atomic::{AtomicU32, Ordering};

static mut Q: Queue<[u8; 32], 8> = Queue::new();
static NEXT_ID: AtomicU32 = AtomicU32::new(0);

fn producer_task() {
    // Safety: only called before RTOS start, so we get a unique splitter.
    let (mut prod, _) = unsafe { Q.split() };
    let mut packet = [0u8; 32];
    let id = NEXT_ID.fetch_add(1, Ordering::Relaxed);
    packet[..4].copy_from_slice(&id.to_le_bytes());
    prod.enqueue(packet).ok();
}

fn consumer_task() {
    let (_, mut cons) = unsafe { Q.split() };
    while let Some(pkt) = cons.dequeue() {
        process_packet(&pkt);
    }
}
```

Example: DMA Buffer Placement

Long-running transfers often require buffers in SRAM domains accessible to both DMA and CPU.

`#[link_section]` places the buffer without a linker script change.

```
#[link_section = ".dma_data"]
static mut ADC_BUFFER: [u16; 128] = [0; 128];

fn start_dmaadc: &mut AdcDma<'static>) {
    // Safety: DMA exclusively owns the buffer until transfer completes.
    unsafe { adc.start_dma(&mut ADC_BUFFER) }.unwrap();
}
```

Example: Fixed-Capacity Command Log

`heapless::Vec` provides a familiar `Vec` API with compile-time capacity. Store operational history without ever touching the heap.

```

use core::cell::RefCell;
use critical_section::Mutex;
use heapless::Vec;

#[derive(Clone, Copy)]
pub struct Command {
    opcode: u8,
    payload: [u8; 4],
}

static COMMAND_LOG: Mutex<RefCell<Vec<Command, 32>>> =
    Mutex::new(RefCell::new(Vec::new()));

pub fn append_command(cmd: Command) {
    critical_section::with(|cs| {
        let mut log = COMMAND_LOG.borrow(cs).borrow_mut();
        log.push(cmd).ok(); // drop oldest silently when full
    });
}

pub fn latest() -> Option<Command> {
    critical_section::with(|cs| COMMAND_LOG.borrow(cs).borrow().last().copied())
}

```

Example: STM32 DMA Double Buffer

Some STM32 families (F7/H7) support double-buffered DMA streams. Pre-allocate both halves so high-rate peripherals (audio, SDR) never wait for allocation.

```

#[link_section = ".sram_d2"]
static mut AUDIO_BUFFERS: [[i16; 256]; 2] = [[0; 256]; 2];

fn start_audio_dma(dma: &mut stm32h7xx_hal::dma::StreamX<DMA1>) {
    let (buf_a, buf_b) = unsafe {
        (
            &mut AUDIO_BUFFERS[0] as *mut _,
            &mut AUDIO_BUFFERS[1] as *mut _,
        )
    };
    unsafe {
        dma.set_memory0(buf_a as *mut _);
        dma.set_memory1(buf_b as *mut _);
    }
    dma.enable_double_buffer();
    dma.start();
}

```

ISR handlers can then refill whichever half just completed without races or heap usage.

Pattern 3: Interrupt-Safe Shared State

- **Problem:** ISRs need to communicate with foreground tasks without data races. `static mut` variables are unsafe, and `RefCell` panics in interrupts.
- **Solution:** Use synchronization primitives tailored to bare metal: `cortex_m::interrupt::Mutex`, `critical_section::Mutex`, atomics, or lock-free queues. Disable interrupts only around the minimum critical section.
- **Why It Matters:** Predictable interrupt latency, no priority inversion, and analyzable execution times.
- **Use Cases:** Button debouncing, timer capture/compare, sensor event batching, cross-core mailboxes.

Examples

Example: Critical Section with Mutex

```
use core::cell::RefCell;
use cortex_m::interrupt::{free, Mutex};

static BUTTON_COUNT: Mutex<RefCell<u32>> = Mutex::new(RefCell::new(0));

#[interrupt]
fn EXTI0() {
    free(|cs| {
        let mut count = BUTTON_COUNT.borrow(cs).borrow_mut();
        *count += 1;
    });
}

fn read_count() -> u32 {
    free(|cs| *BUTTON_COUNT.borrow(cs).borrow())
}
```

Example: Atomic Flag for Wake-Ups

Use atomics for small pieces of state to avoid mutex overhead entirely.

```
use core::sync::atomic::{AtomicBool, Ordering};

static DATA_READY: AtomicBool = AtomicBool::new(false);

#[interrupt]
fn ADC1() {
    DATA_READY.store(true, Ordering::Release);
}

fn main_loop() {
    loop {
        if DATA_READY.swap(false, Ordering::AcqRel) {
```

```

        handle_sample();
    }
    cortex_m::asm::wfi(); // sleep until next interrupt
}

```

Example: Sharing Buses with `critical_section::Mutex`

When a driver must be callable from both interrupts and async tasks, wrap it in `critical_section::Mutex` to get `Send + Sync` access without global `unsafe`.

```

use core::cell::RefCell;
use critical_section::Mutex;

struct EnvSensor<I2C> {
    bus: I2C,
}

static SENSOR: Mutex<RefCell<Option<EnvSensor<I2cDriver>>> =
    Mutex::new(RefCell::new(None));

fn init_sensor(bus: I2cDriver) {
    critical_section::with(|cs| {
        *SENSOR.borrow(cs).borrow_mut() = Some(EnvSensor { bus });
    });
}

fn read_temperature() -> Option<i16> {
    critical_section::with(|cs| {
        let mut guard = SENSOR.borrow(cs).borrow_mut();
        guard.as_mut().and_then(|sensor| sensor.bus.read_temp().ok())
    })
}

```

Example: Raspberry Pi GPIO Interrupt Counter

On Raspberry Pi OS you can wire edge-triggered callbacks with `rppal`. Use atomics so the callback remains lock-free just like an ISR on bare metal.

```

use rppal::gpio::{Gpio, Trigger};
use core::sync::atomic::{AtomicU32, Ordering};

static BUTTON_COUNT: AtomicU32 = AtomicU32::new(0);

fn init_button(pin: u8) -> Result<(), rppal::gpio::Error> {
    let gpio = Gpio::new()?;
    let mut button = gpio.get(pin)?.into_input_pulldown();
    button.set_async_interrupt(Trigger::FallingEdge, |_|
        BUTTON_COUNT.fetch_add(1, Ordering::Relaxed));
}

```

```

    Ok(())
}

fn button_presses() -> u32 {
    BUTTON_COUNT.load(Ordering::Relaxed)
}

```

Pattern 4: Deterministic Scheduling with RTIC/Embassy

- **Problem:** Cooperative `loop {}` architectures make it hard to guarantee deadlines when peripherals compete for CPU time.
- **Solution:** Use a lightweight real-time framework (RTIC, Embassy) that models tasks as interrupt handlers with explicit priorities and resource locking.
- **Why It Matters:** Priority-based scheduling gives bounded latency, automatic critical sections, and eliminates the need for a traditional RTOS.
- **Use Cases:** Motor control loops, sensor fusion pipelines, industrial fieldbus stacks, battery management systems.

Examples

Example: RTIC Task Graph

`rtic::app` wires interrupts, priorities, and shared resources without a heap.

```

#![no_std]
#![no_main]

#[rtic::app(device = stm32f4xx_hal::pac, peripherals = true)]
mod app {
    use super::*;

    #[shared]
    struct Shared {
        rpm: u16,
    }

    #[local]
    struct Local {
        encoder: Encoder,
        pwm: PwmDriver,
    }

    #[init]
    fn init(ctx: init::Context) -> (Shared, Local) {
        // init hardware ...
        (Shared { rpm: 0 }, Local { encoder, pwm })
    }

    #[task(binds = TIM2, shared = [rpm], local = [encoder])]
    fn sample(mut ctx: sample::Context) {

```

```

    let rpm_measurement = ctx.local.encoder.capture();
    ctx.shared.rpm.lock(|rpm| *rpm = rpm_measurement);
}

#[task(priority = 2, shared = [rpm], local = [pwm])]
fn control(mut ctx: control::Context) {
    let rpm = *ctx.shared.rpm.lock(|rpm| rpm);
    let duty = pid_step(rpm);
    ctx.local.pwm.set_duty(duty);
}
}

```

Example: Embassy Async Driver

Embassy's async executors integrate timers and low-power WFI sleep automatically.

```

#[embassy_executor::main]
async fn main(spawner: Spawner) {
    let p = embassy_stm32::init(Default::default());
    let mut uart = Uart::new(p.USART2, p.PA2, p.PA3, Irqs, p.DMA1_CH6, p.DMA1_CH7);

    spawner.spawn(sample_task()).unwrap();

    loop {
        uart.write(b"ping\r\n").await.unwrap();
        Timer::after_secs(1).await;
    }
}

```

Example: Embassy Channels for Task Isolation

Use `embassy_sync::channel` to decouple fast sampling tasks from slower processing, keeping deadlines intact.

```

use embassy_executor::{Spawner, task};
use embassy_sync::channel::Channel;
use embassy_sync::blocking_mutex::raw::NoopRawMutex;
use embassy_time::{Duration, Timer};

static ADC_SAMPLES: Channel<NoopRawMutex, u16, 8> = Channel::new();

#[task]
async fn adc_sampler() {
    loop {
        let sample = read_adc_sample();
        ADC_SAMPLES.send(sample).await;
        Timer::after(Duration::from_micros(500)).await;
    }
}

#[task]

```

```

async fn filter_task() {
    loop {
        let sample = ADC_SAMPLES.recv().await;
        let filtered = low_pass(sample);
        publish(filtered).await;
    }
}

#[embassy_executor::main]
async fn main(spawner: Spawner) {
    spawner.spawn(adc_sampler()).unwrap();
    spawner.spawn(filter_task()).unwrap();
    embassy_time::Timer::after_secs(1).await;
}

```

Design tip: Keep ISR work minimal (capture timestamp, enqueue event) and defer heavy computation to lower-priority tasks to maintain deadlines.

Checklist for Embedded Rust Patterns - Compile with `#![no_std]` and `panic_probe/defmt` for meaningful crash info. - Keep unsafe code confined to BSP crates; expose safe APIs upward. - Measure worst-case execution times (WCET) per task and ensure they fit within interrupt budgets. - Use hardware timers for scheduling instead of busy loops to save power.

Appendix A: Comprehensive Standard Library Reference

Sections:

- [Type Conversion Cheatsheet](#)
- [Common Trait Implementations](#)
- [Iterator Combinators Reference](#)
- [Collections: Vec, HashMap, HashSet, and More](#)
- [String and Text Processing](#)
- [Cargo Commands Reference](#)

As you develop production Rust code, certain patterns recur constantly: converting between types, implementing common traits, chaining iterators, and managing projects with Cargo. While the main chapters explore these concepts in depth, this appendix provides a condensed reference for the patterns you'll reach for daily.

Think of this as your muscle memory guide. The type conversions you implement most frequently. The trait derivations that make your structs ergonomic. The iterator combinators that transform data pipelines. The Cargo commands that streamline your workflow. Each section distills practical knowledge into actionable patterns, organized for quick lookup when you know what you need but need a syntax reminder.

This reference is designed for experienced programmers who understand the underlying concepts and need fast access to implementation patterns. If you're new to a concept, follow the cross-references to the detailed chapters where we explore the why and when alongside the how.

Type Conversion Cheatsheet

Type conversions in Rust are explicit by design, preventing the subtle bugs that plague languages with implicit coercion. The standard library provides a hierarchy of conversion traits, each with different guarantees about safety, cost, and ownership.

The Conversion Hierarchy

Understanding when to use each conversion trait is crucial for API design. Here's the mental model:

Infallible conversions (always succeed) use `From` and `Into`. These represent transformations where the source type's entire value space maps cleanly into the target type. A `u8` can always become a `u32` because every 8-bit value fits in 32 bits. A `&str` can always become a `String` through allocation.

Fallible conversions (might fail) use `TryFrom` and `TryInto`. These handle narrowing conversions where the source type's value space exceeds the target. Converting `u32` to `u8` might fail for values above 255. Parsing strings into numbers might fail for invalid input.

Borrow conversions (zero-cost) use `AsRef` and `AsMut`. These provide cheap reference conversions, enabling generic functions to accept multiple concrete types through a shared borrowed view.

```
//=====
// The conversion trait landscape
//=====

use std::convert::{From, Into, TryFrom, TryInto, AsRef, AsMut};

//=====
// Pattern 1: Implementing From (Into comes free)
//=====

struct UserId(u64);
struct DatabaseId(u64);

impl From<DatabaseId> for UserId {
    fn from(db_id: DatabaseId) -> Self {
        UserId(db_id.0) // Infallible conversion
    }
}

//=====
// Now both directions work
//=====

let db_id = DatabaseId(42);
let user_id: UserId = db_id.into();           // Into is automatic
let user_id2 = UserId::from(DatabaseId(43)); // From is explicit

//=====
```

```

// Pattern 2: Implementing TryFrom for validated conversions
//=====
use std::num::TryFromIntError;

struct Port(u16);

impl TryFrom<u32> for Port {
    type Error = &'static str;

    fn try_from(value: u32) -> Result<Self, Self::Error> {
        if value <= 65535 {
            Ok(Port(value as u16))
        } else {
            Err("Port number exceeds maximum (65535)")
        }
    }
}

//=====
// Use with ? for error propagation
//=====

fn parse_port(input: u32) -> Result<Port, &'static str> {
    let port = Port::try_from(input)?;
    Ok(port)
}

//=====
// Pattern 3: AsRef for flexible APIs
//=====

fn log_message<S: AsRef<str>>(msg: S) {
    println!("{}", msg.as_ref());
}

//=====
// Works with many string types
//=====

log_message("literal");           // &str
log_message(String::from("owned")); // String
log_message("owned".to_string());  // String

//=====
// Pattern 4: AsMut for generic mutation
//=====

fn clear_buffer<T: AsMut<[u8]>>(buffer: &mut T) {
    for byte in buffer.as_mut() {
        *byte = 0;
    }
}

let mut vec_buffer = vec![1, 2, 3];
let mut array_buffer = [4, 5, 6];

```

```
clear_buffer(&mut vec_buffer);
clear_buffer(&mut array_buffer);
```

Common Conversion Patterns

String Conversions are the most frequent in real code. The ecosystem has converged on patterns that balance efficiency with ergonomics:

```
//=====
// &str → String (allocation required)
//=====

let owned: String = "borrowed".to_string();           // Uses Display
let owned: String = "borrowed".to_owned();            // Uses ToOwned
let owned: String = String::from("borrowed");         // Uses From
let owned: String = "borrowed".into();                // Uses Into (needs type hint)

//=====
// String → &str (cheap borrow)
//=====

let s = String::from("hello");
let borrowed: &str = &s;                            // Deref coercion
let borrowed: &str = s.as_str();                     // Explicit

//=====
// &str → Cow<str> (zero-copy when possible)
//=====

use std::borrow::Cow;

fn maybe_uppercase(s: &str, should_uppercase: bool) -> Cow<str> {
    if should_uppercase {
        Cow::Owned(s.to_uppercase()) // Allocates
    } else {
        Cow::Borrowed(s)          // Zero-copy
    }
}
```

Numeric Conversions require care because Rust prevents silent overflow. The patterns reflect whether you're widening (always safe) or narrowing (potentially lossy):

```
//=====
// Widening: use From/Into (infallible)
//=====

let x: u8 = 255;
let y: u32 = x.into();                  // Always succeeds
let z: u32 = u32::from(x);             // Equivalent

//=====
// Narrowing: use TryFrom/TryInto (fallible)
//=====

let big: u32 = 300;
let small: Result<u8, _> = big.try_into(); // Err for values > 255
```

```

//=====
// Lossy: use as for explicit truncation
//=====

let truncated = big as u8;           // Compiles but truncates to 44

//=====
// Floating point conversions (always explicit)
//=====

let precise: f64 = 3.14159;
let rough = precise as f32;        // Loses precision
let integer = precise as i32;      // Truncates to 3

```

Collection Conversions leverage `FromIterator` and `IntoIterator` to transform between collection types:

```

use std::collections::{HashSet, HashMap, BTreeSet};

//=====
// Vec → HashSet (deduplication)
//=====

let numbers = vec![1, 2, 2, 3, 3, 3];
let unique: HashSet<_> = numbers.into_iter().collect();

//=====
// Vec<(K, V)> → HashMap
//=====

let pairs = vec![("a", 1), ("b", 2)];
let map: HashMap<_, _> = pairs.into_iter().collect();

//=====
// HashSet → Vec (ordering unspecified)
//=====

let set: HashSet<_> = [3, 1, 2].iter().cloned().collect();
let vec: Vec<_> = set.into_iter().collect();

//=====
// HashSet → BTreeSet (ordered)
//=====

let hash_set: HashSet<_> = [3, 1, 2].iter().cloned().collect();
let tree_set: BTreeSet<_> = hash_set.into_iter().collect();

//=====
// Array → Vec (ownership transfer)
//=====

let array = [1, 2, 3, 4, 5];
let vec = array.to_vec();           // Clone
let vec = Vec::from(array);        // Also clone

```

Quick Reference Table

| From | To | Trait | Notes |
|-------------|-------------|------------------|--------------------------------|
| &str | String | Into/From | Allocates |
| String | &str | AsRef/Deref | Zero-cost borrow |
| &[T] | Vec<T> | Into/From | Clones elements |
| Vec<T> | &[T] | AsRef/Deref | Zero-cost borrow |
| [T; N] | Vec<T> | From | Moves or clones depending on T |
| u8 | u32 | Into/From | Widening (safe) |
| u32 | u8 | TryInto/TryFrom | Narrowing (may fail) |
| i32 | f64 | Into/From | Exact representation |
| f64 | i32 | as cast | Truncates, may overflow |
| Option<T> | Result<T,E> | ok_or/ok_or_else | Provide error for None case |
| Result<T,E> | Option<T> | ok() | Discards error |
| &Path | &OsStr | AsRef | Zero-cost |
| PathBuf | OsString | Into | Ownership transfer |

Common Trait Implementations

Rust's trait system enables code reuse through composition rather than inheritance. The standard library defines dozens of traits, but a handful appear repeatedly in production code. Understanding which traits to derive, which to implement manually, and how they interact is essential for ergonomic API design.

The Derivable Core

Most custom types should derive these traits unless they have specific reasons not to:

```
//=====
// The standard derive bundle for data types
//=====

#[derive(Debug, Clone, PartialEq, Eq, Hash)]
struct User {
    id: u64,
    username: String,
    email: String,
}

// Debug: Required for debugging, logging, and error messages
// Clone: Enables explicit duplication
// PartialEq/Eq: Enables == comparisons and use in HashSet/HashMap
// Hash: Enables use as HashMap keys
```

When to skip derives:

- **Skip Clone** for large structs where cloning is expensive and you want to make ownership transfers explicit
- **Skip PartialEq** for types where equality is ambiguous (floating point, time ranges)
- **Skip Hash** for types that shouldn't be used as keys (mutable state, large blobs)
- **Skip Eq** for types containing `f32/f64` (NaN breaks reflexivity)

```
//=====
// Example: Skipping Clone for large, move-only types
//=====

#[derive(Debug)]
struct LargeBuffer {
    data: Vec<u8>, // Intentionally move-only
}

//=====
// Example: Custom PartialEq for case-insensitive comparison
//=====

#[derive(Debug, Clone)]
struct CaseInsensitiveString(String);

impl PartialEq for CaseInsensitiveString {
    fn eq(&self, other: &Self) -> bool {
        self.0.to_lowercase() == other.0.to_lowercase()
    }
}
```

Ordering Traits: PartialOrd and Ord

Ordering enables sorting, binary search, and range operations. The distinction between `PartialOrd` and `Ord` reflects whether all values are comparable:

```
use std::cmp::Ordering;

//=====
// Ord: Total ordering (all values comparable)
//=====

#[derive(Debug, Clone, PartialEq, Eq, PartialOrd, Ord)]
struct Priority {
    level: u8, // Compared first (due to field order)
    timestamp: u64, // Tiebreaker
}

//=====
// Use in sorted collections
//=====

use std::collections::BTreeSet;
let mut tasks = BTreeSet::new();
tasks.insert(Priority { level: 1, timestamp: 100 });
tasks.insert(Priority { level: 2, timestamp: 50 });
```

```
// Automatically sorted by priority, then timestamp

//=====
// Custom Ord for reverse ordering
//=====

#[derive(Debug, Clone, PartialEq, Eq)]
struct ReverseScore(u32);

impl Ord for ReverseScore {
    fn cmp(&self, other: &Self) -> Ordering {
        // Reverse the natural ordering
        other.0.cmp(&self.0)
    }
}

impl PartialOrd for ReverseScore {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        Some(self.cmp(other))
    }
}
```

PartialOrd without Ord is necessary for types with incomparable values:

```
//=====
// FloatWrapper can't implement Ord because NaN isn't comparable
//=====

#[derive(Debug, Clone, PartialEq)]
struct FloatWrapper(f64);

impl PartialOrd for FloatWrapper {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        // Returns None for NaN comparisons
        self.0.partial_cmp(&other.0)
    }
}
```

Default: Zero-Cost Initialization

The **Default** trait provides canonical “zero” values, enabling concise initialization and builder patterns:

```
//=====
// Derive Default when all fields implement Default
//=====

#[derive(Debug, Default)]
struct Config {
    timeout_ms: u64,      // Defaults to 0
    retries: u32,         // Defaults to 0
    verbose: bool,        // Defaults to false
}

//=====
```

```

// Custom Default for better defaults
//=====
#[derive(Debug)]
struct Connection {
    host: String,
    port: u16,
    timeout_ms: u64,
}

impl Default for Connection {
    fn default() -> Self {
        Connection {
            host: "localhost".to_string(),
            port: 8080,
            timeout_ms: 5000,
        }
    }
}

//=====
// Use with struct update syntax
//=====

let conn = Connection {
    host: "api.example.com".to_string(),
    ..Default::default() // Fill remaining fields
};

//=====
// Use with Option::unwrap_or_default
//=====

fn get_config(maybe_config: Option<Config>) -> Config {
    maybe_config.unwrap_or_default()
}

```

Display and Debug: Human-Readable Output

`Debug` is for developers; `Display` is for users. The distinction guides how you present your types:

```

use std::fmt;

#[derive(Debug)] // Auto-derived for development
struct Timestamp {
    unix_seconds: i64,
}

//=====
// Manual Display for user-facing output
//=====

impl fmt::Display for Timestamp {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "Timestamp({})", self.unix_seconds)
    }
}

```

```

}

//=====
// Using both
//=====

let ts = Timestamp { unix_seconds: 1609459200 };
println!("{:?}", ts); // Debug: Timestamp { unix_seconds: 1609459200 }
println!("{}", ts); // Display: Timestamp(1609459200)

//=====
// Pretty-printing with {:#?}
//=====

#[derive(Debug)]
struct Nested {
    users: Vec<String>,
    config: Config,
}
// {:#?} formats with indentation for nested structures

```

Error: Making Errors First-Class

Types implementing `Error` integrate with Rust's error handling ecosystem, enabling `? propagation` and `error context`:

```

use std::error::Error;
use std::fmt;

//=====
// Minimal error type
//=====

#[derive(Debug)]
enum ApiError {
    NetworkFailure(String),
    InvalidResponse,
    Unauthorized,
}

impl fmt::Display for ApiError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            ApiError::NetworkFailure(msg) => write!(f, "Network error: {}", msg),
            ApiError::InvalidResponse => write!(f, "Invalid response from server"),
            ApiError::Unauthorized => write!(f, "Authentication required"),
        }
    }
}

impl Error for ApiError {}

//=====
// Use in Result types
//=====


```

```

fn fetch_data() -> Result<String, ApiError> {
    Err(ApiError::NetworkFailure("Connection timeout".to_string()))
}

//=====
// Chain with ? operator
//=====

fn process() -> Result<(), Box<dyn Error>> {
    let data = fetch_data()?;
    // Automatically converts
    Ok(())
}

```

Using thiserror for ergonomic error types:

```

//=====
// With thiserror crate (recommended for libraries)
//=====

use thiserror::Error;

#[derive(Error, Debug)]
enum DataError {
    #[error("I/O error: {0}")]
    Io(#[from] std::io::Error),

    #[error("Parse error at line {line}: {msg}")]
    Parse { line: usize, msg: String },

    #[error("Invalid format")]
    InvalidFormat,
}

// Derives Display, Error, and From conversions automatically

```

Iterator: Making Types Traversable

Implementing `Iterator` allows your types to work with for loops and iterator combinators:

```

//=====
// Simple iterator over a range
//=====

struct CountDown {
    count: u32,
}

impl Iterator for CountDown {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        if self.count > 0 {
            let current = self.count;
            self.count -= 1;
            Some(current)
        } else {
            None
        }
    }
}

```

```

        } else {
            None
        }
    }

//=====
// Use with for loops
//=====

for n in CountDown { count: 5 } {
    println!("{}", n); // 5, 4, 3, 2, 1
}

//=====
// Use with combinators
//=====

let countdown = CountDown { count: 5 };
let sum: u32 = countdown.filter(|&n| n % 2 == 0).sum(); // 4 + 2 = 6

```

Implementing IntoIterator for ergonomic iteration:

```

struct Playlist {
    songs: Vec<String>,
}

//=====
// Implement IntoIterator to enable for loops directly
//=====

impl IntoIterator for Playlist {
    type Item = String;
    type IntoIter = std::vec::IntoIter<String>;

    fn into_iter(self) -> Self::IntoIter {
        self.songs.into_iter()
    }
}

//=====
// Now works directly in for loops
//=====

let playlist = Playlist {
    songs: vec!["Song 1".to_string(), "Song 2".to_string()],
};

for song in playlist { // No need for .songs.into_iter()
    println!("{}", song);
}

```

Quick Reference Table

| Trait | Purpose | Derive? When to Implement Manually | |
|--------------|------------------------|------------------------------------|---|
| Debug | Developer debugging | Yes | Rarely; only for security-sensitive types |
| Clone | Explicit duplication | Yes | Skip for move-only types |
| Copy | Implicit duplication | Yes | Only for small, bitwise-copyable types |
| PartialEq | Equality comparison | Yes | Custom equality logic (case-insensitive, etc) |
| Eq | Reflexive equality | Yes | Skip for types with NaN (f32/f64) |
| PartialOrd | Partial ordering | Yes | Custom comparison logic |
| Ord | Total ordering | Yes | Reverse ordering, multi-field priority |
| Hash | Hash computation | Yes | Custom hash logic, skip for unhashable fields |
| Default | Zero/empty value | Yes | Better defaults than all-zeros |
| Display | User-facing output | No | Always manual |
| Error | Error type integration | No | Always manual (or use thiserror) |
| Iterator | Sequential traversal | No | Always manual |
| IntoIterator | Enable for loops | No | Collection-like types |
| From/Into | Type conversion | No | Always manual |
| Deref | Smart pointer behavior | No | Wrapper types needing transparent access |
| Drop | Custom cleanup | No | Resource management (files, locks, etc) |

Iterator Combinators Reference

Iterators are Rust's primary abstraction for sequential processing. They're lazy, composable, and often compile down to the same machine code as hand-written loops. Mastering iterator combinators transforms verbose imperative code into concise, declarative pipelines.

The key insight: iterators are **adapters and consumers**. Adapters transform iterators into new iterators (lazy). Consumers process iterators and return concrete values (eager). Chaining adapters builds up computation; calling a consumer executes it.

Creating Iterators

Every collection can produce iterators in three flavors, each with different ownership semantics:

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

//=====
// iter() - borrows elements immutably
//=====
for &item in data.iter() {
    println!("{}", item); // item is &i32, pattern &item extracts i32
}
//=====
// data is still valid
```

```

//=====
// =====
// // iter_mut() – borrows elements mutably
//=====

let mut data = vec![1, 2, 3];
for item in data.iter_mut() {
    *item *= 2; // item is &mut i32
}
// data is still valid, now [2, 4, 6]

//=====
// // into_iter() – takes ownership, consumes collection
//=====

for item in data.into_iter() {
    println!("{}", item); // item is i32
}
// data is now invalid (moved)

```

Manual iterator creation:

```

//=====
// Range iterators
//=====

(0..10)           // 0 to 9
(0..=10)          // 0 to 10 (inclusive)
(0...).take(100)  // Infinite iterator, take first 100

//=====
// From functions
//=====

std::iter::once(42)           // Single element
std::iter::repeat(7).take(5)   // [7, 7, 7, 7, 7]
std::iter::repeat_with(|| rand::random()) // Computed on each iteration
std::iter::empty::<i32>()      // Empty iterator

//=====
// From existing values
//=====

std::iter::from_fn(|| Some(42)) // Custom generation logic

```

Adapter Combinators: Transforming Iterators

Adapters are lazy—they don't compute anything until consumed. This enables efficient chaining without intermediate allocations.

Mapping: Transforming Elements

```

//=====
// map: Transform each element
//=====

```

```

let numbers = vec![1, 2, 3];
let doubled: Vec<_> = numbers.iter().map(|x| x * 2).collect();
// [2, 4, 6]

//=====
// map with complex transformations
//=====

let users = vec!["Alice", "Bob"];
let greetings: Vec<_> = users.iter()
    .map(|name| format!("Hello, {}!", name))
    .collect();
// ["Hello, Alice!", "Hello, Bob!"]

//=====
// filter_map: Map and filter in one pass
//=====

let inputs = vec!["42", "abc", "100"];
let numbers: Vec<i32> = inputs.iter()
    .filter_map(|s| s.parse().ok()) // Parse, keep only successes
    .collect();
// [42, 100]

//=====
// flat_map: Map and flatten
//=====

let words = vec!["hello", "world"];
let chars: Vec<_> = words.iter()
    .flat_map(|word| word.chars())
    .collect();
// ['h', 'e', 'l', 'l', 'o', 'w', 'o', 'r', 'l', 'd']

```

Filtering: Selecting Elements

```

//=====
// filter: Keep elements matching predicate
//=====

let numbers = vec![1, 2, 3, 4, 5, 6];
let evens: Vec<_> = numbers.iter()
    .filter(|&&x| x % 2 == 0)
    .copied() // Convert &i32 to i32
    .collect();
// [2, 4, 6]

//=====
// take: First N elements
//=====

let first_three: Vec<_> = (1..=100).take(3).collect();
// [1, 2, 3]

//=====
// take_while: Elements until predicate fails
//=====
```

```

let less_than_five: Vec<_> = (1..=10)
    .take_while(|&x| x < 5)
    .collect();
// [1, 2, 3, 4]

=====
// skip: Skip first N elements
=====
let skip_first: Vec<_> = vec![1, 2, 3, 4, 5].into_iter().skip(2).collect();
// [3, 4, 5]

=====
// skip_while: Skip until predicate fails
=====
let skip_small: Vec<_> = vec![1, 2, 3, 4, 5].into_iter()
    .skip_while(|&x| x < 3)
    .collect();
// [3, 4, 5]

```

Combining: Multiple Iterators

```

=====
// chain: Concatenate iterators
=====
let a = vec![1, 2];
let b = vec![3, 4];
let combined: Vec<_> = a.iter().chain(b.iter()).copied().collect();
// [1, 2, 3, 4]

=====
// zip: Pair elements from two iterators
=====
let names = vec!["Alice", "Bob"];
let ages = vec![30, 25];
let people: Vec<_> = names.iter().zip(ages.iter()).collect();
// [("Alice", 30), ("Bob", 25)]

=====
// Stops at shortest iterator
=====
let short = vec![1, 2];
let long = vec![10, 20, 30, 40];
let pairs: Vec<_> = short.iter().zip(long.iter()).collect();
// [(1, 10), (2, 20)]

=====
// enumerate: Add indices
=====
let letters = vec!['a', 'b', 'c'];

```

```
let indexed: Vec<_> = letters.iter().enumerate().collect();
// [(0, 'a'), (1, 'b'), (2, 'c')]
```

Inspection: Observing Elements

```
//=====
// inspect: Peek at elements without consuming
//=====

let sum: i32 = (1..=5)
    .inspect(|x| println!("Processing {}", x))
    .map(|x| x * 2)
    .inspect(|x| println!("Doubled to {}", x))
    .sum();
// Prints each step, then returns 30
```

Consumer Combinators: Producing Values

Consumers are eager—they process the entire iterator and return a result.

Collecting: Building Collections

```
use std::collections::{HashMap, HashSet, BTreeSet};

//=====
// collect into Vec
//=====

let vec: Vec<i32> = (1..=5).collect();

//=====
// collect into HashSet (deduplicates)
//=====

let set: HashSet<_> = vec![1, 2, 2, 3].into_iter().collect();
// {1, 2, 3}

//=====
// collect into HashMap from tuples
//=====

let map: HashMap<_, _> = vec![("a", 1), ("b", 2)].into_iter().collect();

//=====
// collect into String
//=====

let chars = vec!['h', 'e', 'l', 'l', 'o'];
let word: String = chars.into_iter().collect();
// "hello"

//=====
// partition: Split into two collections
//=====

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

```

let (evens, odds): (Vec<_>, Vec<_>) = numbers.into_iter()
    .partition(|&x| x % 2 == 0);
// evens: [2, 4], odds: [1, 3, 5]

```

Searching: Finding Elements

```

//=====
// find: First element matching predicate
//=====
let numbers = vec![1, 2, 3, 4];
let first_even = numbers.iter().find(|&x| x % 2 == 0);
// Some(&2)

//=====
// position: Index of first match
//=====
let pos = numbers.iter().position(|x| x == 3);
// Some(2)

//=====
// any: Check if any element matches
//=====
let has_even = numbers.iter().any(|x| x % 2 == 0);
// true

//=====
// all: Check if all elements match
//=====
let all_positive = numbers.iter().all(|x| x > 0);
// true

//=====
// nth: Get element at index
//=====
let third = numbers.iter().nth(2);
// Some(&3)

//=====
// last: Get last element
//=====
let last = numbers.iter().last();
// Some(&4)

```

Aggregating: Reducing to Single Values

```

//=====
// sum: Add all elements
//=====
let total: i32 = (1..=10).sum();
// 55

```

```

//=====
// product: Multiply all elements
//=====
let factorial: i32 = (1..=5).product();
// 120

//=====
// fold: Custom accumulation
//=====
let sum = (1..=5).fold(0, |acc, x| acc + x);
// 15

//=====
// fold for non-numeric types
//=====
let sentence = vec!["Hello", "world"];
let joined = sentence.into_iter().fold(String::new(), |mut acc, word| {
    if !acc.is_empty() {
        acc.push(' ');
    }
    acc.push_str(word);
    acc
});
// "Hello world"

//=====
// reduce: Like fold but uses first element as initial value
//=====
let max = vec![3, 1, 4, 1, 5].into_iter().reduce(|a, b| a.max(b));
// Some(5)

//=====
// max/min: Find extremes
//=====
let max = vec![3, 1, 4].into_iter().max();
// Some(4)

let min = vec![3, 1, 4].into_iter().min();
// Some(1)

//=====
// max_by/min_by: Custom comparison
//=====
let words = vec!["short", "longer", "longest"];
let longest = words.iter().max_by_key(|word| word.len());
// Some("longest")

```

Counting and Testing

```

//=====
// count: Number of elements

```

```
//=====
let count = (1..=100).filter(|x| x % 2 == 0).count();
// 50

//=====
// for_each: Side effects for each element
//=====
(1..=5).for_each(|x| println!("{}", x));
// Prints 1 through 5
```

Advanced Patterns: Real-World Pipelines

Iterator combinators shine in data processing pipelines where you transform, filter, and aggregate data in a single pass:

```
//=====
// Example: Process log lines
//=====

let log_lines = vec![
    "ERROR: Database connection failed",
    "INFO: Server started",
    "ERROR: Null pointer exception",
    "WARN: High memory usage",
];

let error_count = log_lines.iter()
    .filter(|line| line.starts_with("ERROR"))
    .count();
// 2

//=====
// Example: Transform and collect user data
//=====

struct User {
    name: String,
    age: u32,
    active: bool,
}

let users = vec![
    User { name: "Alice".to_string(), age: 30, active: true },
    User { name: "Bob".to_string(), age: 25, active: false },
    User { name: "Charlie".to_string(), age: 35, active: true },
];

let active_names: Vec<String> = users.into_iter()
    .filter(|user| user.active)
    .map(|user| user.name)
    .collect();
// ["Alice", "Charlie"]

//=====
```

```

// Example: Nested iteration with flat_map
//=====
let teams = vec![
    vec!["Alice", "Bob"],
    vec!["Charlie", "Dave", "Eve"],
];
// ["Alice", "Bob", "Charlie", "Dave", "Eve"]

//=====
// Example: Grouping with fold
//=====
use std::collections::HashMap;

let words = vec!["apple", "apricot", "banana", "blueberry"];
let grouped: HashMap<char, Vec<&str>> = words.into_iter()
    .fold(HashMap::new(), |mut map, word| {
        map.entry(word.chars().next().unwrap())
            .or_insert_with(Vec::new)
            .push(word);
        map
    });
// { 'a': ["apple", "apricot"], 'b': ["banana", "blueberry"] }

```

Quick Reference Table

| Combinator | Type | Signature | Use Case |
|------------|----------|-------------------------------------|-------------------------------|
| map | Adapter | map(f: T -> U) -> Iterator<U> | Transform each element |
| filter | Adapter | filter(f: T -> bool) -> Iterator<T> | Keep matching elements |
| filter_map | Adapter | filter_map(f: T -> Option<U>) | Map and filter simultaneously |
| flat_map | Adapter | flat_map(f: T -> Iterator<U>) | Map and flatten |
| take | Adapter | take(n: usize) -> Iterator<T> | First N elements |
| skip | Adapter | skip(n: usize) -> Iterator<T> | Skip first N elements |
| chain | Adapter | chain(other: Iterator) -> Iterator | Concatenate iterators |
| zip | Adapter | zip(other: Iterator<U>) -> (T, U) | Pair with another iterator |
| enumerate | Adapter | enumerate() -> (usize, T) | Add indices |
| inspect | Adapter | inspect(f: &T -> ()) -> Iterator<T> | Debug/log without consuming |
| collect | Consumer | collect() -> Collection | Build collection |
| sum | Consumer | sum() -> T | Add all elements |

| Combinator | Type | Signature | Use Case |
|------------|----------|--|----------------------------------|
| product | Consumer | <code>product() -> T</code> | Multiply all elements |
| fold | Consumer | <code>fold(init: B, f: (B, T) -> B) -> B</code> | Custom accumulation |
| reduce | Consumer | <code>reduce(f: (T, T) -> T) -> Option<T></code> | Accumulate without initial value |
| find | Consumer | <code>find(f: T -> bool) -> Option<T></code> | First matching element |
| any | Consumer | <code>any(f: T -> bool) -> bool</code> | Check if any match |
| all | Consumer | <code>all(f: T -> bool) -> bool</code> | Check if all match |
| count | Consumer | <code>count() -> usize</code> | Count elements |
| max/min | Consumer | <code>max() -> Option<T></code> | Find extreme values |
| partition | Consumer | <code>partition(f: T -> bool) -> (C, C)</code> | Split into two collections |

Collections: Vec, HashMap, HashSet, and More

Rust's standard library provides a rich set of collection types, each optimized for different access patterns. Understanding when to use each collection is crucial for writing efficient code. The core principle: **choose the collection whose performance characteristics match your usage pattern.**

Vec: Contiguous Growable Array

`Vec<T>` is the workhorse collection—use it as your default choice unless you have specific requirements for other types. It provides O(1) indexed access and amortized O(1) push/pop at the end.

```
use std::vec::Vec;

//=====
// Creating vectors
//=====

let v1: Vec<i32> = Vec::new();
let v2 = vec![1, 2, 3]; // vec! macro
let v3 = Vec::with_capacity(100); // Pre-allocate
let v4 = vec![0; 5]; // [0, 0, 0, 0, 0]

//=====
// Adding elements
//=====

let mut numbers = Vec::new();
numbers.push(1); // Add to end - O(1) amortized
numbers.extend([2, 3, 4]); // Add multiple
numbers.append(&mut vec![5, 6]); // Move elements from another vec
numbers.insert(0, 0); // Insert at index - O(n)

//=====
// Accessing elements
//=====
```

```

let first = numbers[0];                                // Panics if out of bounds
let second = numbers.get(1);                           // Returns Option<&T>
let last = numbers.last();                            // Option<&T>
let slice = &numbers[1..4];                           // Borrow a slice

=====
// Removing elements
=====
let last = numbers.pop();                            // Option<T> - O(1)
let removed = numbers.remove(0);                     // T - O(n), shifts elements
numbers.clear();                                    // Empty the vec

=====
// Iteration
=====
for num in &numbers {                                // Immutable borrow
    println!("{}", num);
}

for num in &mut numbers {                            // Mutable borrow
    *num *= 2;
}

for num in numbers {                               // Consumes the vec
    println!("{}", num);
}

=====
// Capacity management
=====
let mut v = Vec::with_capacity(10);
println!("len: {}", v.len(), "capacity: {}", v.capacity());
v.reserve(20);                                     // Ensure at least 20 more slots
v.shrink_to_fit();                                 // Release unused memory

=====
// Deduplication and sorting
=====
let mut data = vec![3, 1, 4, 1, 5, 9, 2, 6, 5];
data.sort();                                         // Sort in place - O(n log n)
data.dedup();                                       // Remove consecutive duplicates
// [1, 2, 3, 4, 5, 6, 9]

=====
// Binary search (requires sorted data)
=====
let sorted = vec![1, 2, 3, 4, 5];
match sorted.binary_search(&3) {
    Ok(index) => println!("Found at {}", index),
    Err(index) => println!("Not found, would insert at {}", index),
}

```

VecDeque: Double-Ended Queue

Use `VecDeque<T>` when you need efficient insertion/removal at both ends. It's implemented as a ring buffer.

```
use std::collections::VecDeque;

//=====
// Creating VecDeque
//=====

let mut deque = VecDeque::new();
let mut deque2 = VecDeque::from(vec![1, 2, 3]);

//=====
// Adding at both ends - O(1)
//=====

deque.push_back(1);                      // Add to back
deque.push_front(0);                     // Add to front
// [0, 1]

//=====
// Removing from both ends - O(1)
//=====

let back = deque.pop_back();             // Some(1)
let front = deque.pop_front();           // Some(0)

//=====
// Use cases
//=====

// Queue (FIFO)
let mut queue = VecDeque::new();
queue.push_back(1);
queue.push_back(2);
let first = queue.pop_front();          // FIFO order

// Stack (LIFO) - but Vec is better for this
let mut stack = VecDeque::new();
stack.push_back(1);
stack.push_back(2);
let last = stack.pop_back();            // LIFO order
```

HashMap: Hash-Based Key-Value Store

`HashMap<K, V>` provides $O(1)$ average-case insertion and lookup. Use it when you need fast key-based access and don't care about ordering.

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

//=====
// Creating HashMaps
//=====
```

```

let mut scores = HashMap::new();
let mut map: HashMap<String, i32> = HashMap::with_capacity(100);

//=====
// Inserting and updating
//=====

scores.insert("Alice".to_string(), 10);
scores.insert("Bob".to_string(), 20);

// Returns previous value if key existed
let old = scores.insert("Alice".to_string(), 15); // Some(10)

//=====
// Accessing values
//=====

let alice_score = scores.get("Alice");           // Option<&i32>
let bob_score = scores["Bob"];                   // Panics if missing

// Safe indexing with unwrap_or
let charlie_score = scores.get("Charlie").unwrap_or(&0);

//=====
// Checking existence
//=====

if scores.contains_key("Alice") {
    println!("Alice has a score");
}

//=====
// Removing entries
//=====

let removed = scores.remove("Bob");             // Option<V>

//=====
// Entry API (powerful pattern)
//=====

// Insert if missing
scores.entry("Charlie".to_string()).or_insert(0);

// Modify existing or insert default
let alice = scores.entry("Alice".to_string()).or_insert(0);
*alice += 5;

// Complex logic with entry
let word_counts = vec!["apple", "banana", "apple"];
let mut counts = HashMap::new();
for word in word_counts {
    let count = counts.entry(word).or_insert(0);
    *count += 1;
}
// {"apple": 2, "banana": 1}

//=====

```

```

// Iteration
//=====
for (key, value) in &scores {
    println!("{}: {}", key, value);
}

for key in scores.keys() {
    println!("{}", key);
}

for value in scores.values() {
    println!("{}", value);
}

//=====
// Convert from/to other types
//=====

let pairs = vec![("a", 1), ("b", 2)];
let map: HashMap<_, _> = pairs.into_iter().collect();

let vec_pairs: Vec<_> = map.into_iter().collect();

```

HashSet: Hash-Based Set

`HashSet<T>` stores unique values with O(1) average-case insertion and membership testing. Use it for deduplication and set operations.

```

use std::collections::HashSet;

//=====
// Creating HashSet
//=====

let mut set = HashSet::new();
let set2: HashSet<i32> = [1, 2, 3].iter().cloned().collect();

//=====
// Adding and removing - O(1)
//=====

set.insert(1);                                // true if inserted
set.insert(1);                                // false (already exists)
set.remove(&1);                               // true if removed

//=====
// Checking membership
//=====

if set.contains(&1) {
    println!("Set contains 1");
}

//=====
// Set operations
//=====


```

```

let set1: HashSet<_> = [1, 2, 3].iter().cloned().collect();
let set2: HashSet<_> = [2, 3, 4].iter().cloned().collect();

// Union: all elements from both sets
let union: HashSet<_> = set1.union(&set2).cloned().collect();
// {1, 2, 3, 4}

// Intersection: elements in both sets
let intersection: HashSet<_> = set1.intersection(&set2).cloned().collect();
// {2, 3}

// Difference: elements in first but not second
let diff: HashSet<_> = set1.difference(&set2).cloned().collect();
// {1}

// Symmetric difference: elements in either but not both
let sym_diff: HashSet<_> = set1.symmetric_difference(&set2).cloned().collect();
// {1, 4}

//=====
// Subset and superset
//=====
let small = HashSet::from([1, 2]);
let large = HashSet::from([1, 2, 3]);
assert!(small.is_subset(&large));
assert!(large.is_superset(&small));

//=====
// Deduplication pattern
//=====
let numbers = vec![1, 2, 2, 3, 3, 3];
let unique: HashSet<_> = numbers.into_iter().collect();
let deduped: Vec<_> = unique.into_iter().collect();

```

BTreeMap and BTreeSet: Ordered Collections

Use `BTreeMap<K, V>` and `BTreeSet<T>` when you need sorted keys. They provide $O(\log n)$ operations but maintain sorted order.

```

use std::collections::{BTreeMap, BTreeSet};

//=====
// BTreeMap: Sorted keys
//=====
let mut scores = BTreeMap::new();
scores.insert("Alice", 10);
scores.insert("Charlie", 30);
scores.insert("Bob", 20);

// Iteration is always sorted by key
for (name, score) in &scores {
    println!("{}: {}", name, score);
}

```

```

}

// Alice: 10
// Bob: 20
// Charlie: 30

//=====
// Range queries
//=====

let numbers: BTreeMap<i32, &str> = [
    (1, "one"),
    (5, "five"),
    (10, "ten"),
].iter().cloned().collect();

// Get all entries in range
for (key, value) in numbers.range(2..8) {
    println!("{}: {}", key, value);
}
// 5: five

//=====
// First and last entries
//=====

let first = scores.first_key_value();           // Option<(&K, &V)>
let last = scores.last_key_value();            // Option<(&K, &V)>

//=====
// BTreeSet: Sorted set
//=====

let mut set = BTreeSet::new();
set.insert(5);
set.insert(1);
set.insert(3);

// Always sorted
for num in &set {
    println!("{}", num);
}
// 1, 3, 5

// Range iteration
for num in set.range(2..=5) {
    println!("{}", num);
}
// 3, 5

```

BinaryHeap: Priority Queue

`BinaryHeap<T>` is a max-heap that provides $O(\log n)$ insertion and $O(\log n)$ removal of the largest element.

```
use std::collections::BinaryHeap;

//=====
// Creating a BinaryHeap
//=====

let mut heap = BinaryHeap::new();

//=====
// Adding elements - O(log n)
//=====

heap.push(3);
heap.push(1);
heap.push(5);
heap.push(2);

//=====
// Peeking at largest - O(1)
//=====

let largest = heap.peek(); // Some(&5)

//=====
// Removing largest - O(log n)
//=====

while let Some(max) = heap.pop() {
    println!("{}", max);
}

// 5, 3, 2, 1 (descending order)

//=====
// Min-heap using Reverse
//=====

use std::cmp::Reverse;
let mut min_heap = BinaryHeap::new();
min_heap.push(Reverse(3));
min_heap.push(Reverse(1));
min_heap.push(Reverse(5));

while let Some(Reverse(min)) = min_heap.pop() {
    println!("{}", min);
}

// 1, 3, 5 (ascending order)

//=====
// Priority queue use case
//=====

#[derive(Eq, PartialEq, Debug)]
struct Task {
    priority: u32,
    description: String,
}

impl Ord for Task {
```

```

fn cmp(&self, other: &Self) -> std::cmp::Ordering {
    self.priority.cmp(&other.priority)
}

impl PartialOrd for Task {
    fn partial_cmp(&self, other: &Self) -> Option<std::cmp::Ordering> {
        Some(self.cmp(other))
    }
}

let mut tasks = BinaryHeap::new();
tasks.push(Task { priority: 1, description: "Low".to_string() });
tasks.push(Task { priority: 5, description: "High".to_string() });
tasks.push(Task { priority: 3, description: "Medium".to_string() });

// Process tasks by priority
while let Some(task) = tasks.pop() {
    println!("Processing: {:?}", task.description);
}
// High, Medium, Low

```

LinkedList: Doubly-Linked List

`LinkedList<T>` provides O(1) insertion/removal anywhere if you have a cursor, but poor cache locality.
Rarely the best choice—use `Vec` or `VecDeque` instead in most cases.

```

use std::collections::LinkedList;

//=====
// Creating LinkedList
//=====

let mut list = LinkedList::new();
list.push_back(1);
list.push_back(2);
list.push_front(0);
// [0, 1, 2]

//=====
// Splitting and merging
//=====

let mut list1 = LinkedList::from([1, 2, 3]);
let mut list2 = LinkedList::from([4, 5, 6]);
list1.append(&mut list2);           // Moves all elements from list2
// list1: [1, 2, 3, 4, 5, 6], list2: []

//=====
// Note: LinkedList is rarely optimal!
//=====

// Vec is usually better due to:
// - Better cache locality

```

```
// - Lower memory overhead
// - Better performance for most operations
```

Collection Selection Guide

| Collection | Order | Key Access | Lookup | Insert | Remove | Use When |
|------------|------------|------------|----------|----------|----------|--|
| Vec | Insertion | Index | O(1) | O(1)* | O(n) | Default choice, indexed access |
| VecDeque | Insertion | Index | O(1) | O(1)** | O(1)** | Queue, double-ended operations |
| HashMap | Unordered | Key | O(1) | O(1) | O(1) | Fast key-value lookup, no order needed |
| HashSet | Unordered | Value | O(1) | O(1) | O(1) | Deduplication, membership testing |
| BTreeMap | Sorted | Key | O(log n) | O(log n) | O(log n) | Sorted keys, range queries |
| BTreeSet | Sorted | Value | O(log n) | O(log n) | O(log n) | Sorted set, range queries |
| BinaryHeap | Heap order | N/A | N/A | O(log n) | O(log n) | Priority queue |
| LinkedList | Insertion | N/A | O(n) | O(1)*** | O(1)*** | Rarely useful (prefer Vec/VecDeque) |

* Amortized O(1) at end ** At ends only *** With cursor, O(n) to find position

String and Text Processing

Rust distinguishes between owned strings (`String`) and borrowed string slices (`&str`). Understanding this distinction and the rich text processing capabilities is essential for working with text data.

String vs &str

```
//=====
// &str: Borrowed string slice
//=====

let literal: &str = "Hello, world!";           // String literals are &str
let slice: &str = &String::from("hello")[0..2]; // Slice of String

// &str is:
// - Immutable
// - Fixed size (known at compile time or stored as fat pointer)
// - Doesn't own its data
// - Cheap to pass around

//=====
// String: Owned, growable
//=====

let mut owned = String::from("Hello");
let mut owned2 = "Hello".to_string();
let mut owned3 = String::new();
```

```

// String is:
// - Mutable
// - Heap-allocated
// - Growable
// - UTF-8 encoded

//=====
// Converting between
//=====

let s = String::from("hello");
let slice: &str = &s;                      // String -> &str (cheap)
let slice: &str = s.as_str();               // Explicit conversion

let owned: String = slice.to_string();      // &str -> String (allocates)
let owned: String = slice.to_owned();        // Same
let owned: String = String::from(slice);    // Same

```

String Creation and Manipulation

```

//=====
// Creating strings
//=====

let s1 = String::new();
let s2 = String::from("hello");
let s3 = "hello".to_string();
let s4 = String::with_capacity(100);        // Pre-allocate

//=====
// Appending text
//=====

let mut s = String::from("Hello");
s.push_str(", world");                     // Append &str
s.push('!');                             // Append char
// "Hello, world!"

//=====
// Concatenation
//=====

let s1 = String::from("Hello");
let s2 = String::from(" world");

// Using + (takes ownership of left operand)
let s3 = s1 + &s2;                      // s1 moved, s2 borrowed
// s1 is now invalid!

// Using format! (doesn't take ownership)
let s1 = String::from("Hello");
let s2 = String::from(" world");
let s3 = format!("{}{}", s1, s2);         // Both still valid
let s4 = format!("{}{}");                 // Shorter syntax

```

```

//=====
// Inserting and removing
//=====

let mut s = String::from("Hello world");
s.insert(5, ',');                                // Insert char at byte position
s.insert_str(6, " beautiful");                  // Insert &str
// "Hello, beautiful world"

s.remove(5);                                     // Remove char at byte position
s.truncate(5);                                  // Cut off everything after index
// "Hello"

s.clear();                                       // Empty the string

//=====
// Replacing text
//=====

let s = "I like cats";
let s2 = s.replace("cats", "dogs");             // Returns new String
// "I like dogs"

let s = "aaabbccc";
let s2 = s.replace("a", "x", 2);                // Replace first n occurrences
// "xxabbccc"

```

String Inspection and Searching

```

let text = "Hello, world!";

//=====
// Basic properties
//=====

text.len();                                      // 13 (byte length, not char count!)
text.is_empty();                                 // false

//=====
// Checking contents
//=====

text.starts_with("Hello");                      // true
text.ends_with("!");                            // true
text.contains("world");                         // true

//=====
// Finding patterns
//=====

let pos = text.find("world");                   // Some(7) - byte position
let pos = text.find('w');                       // Some(7)
let rpos = text.rfind('o');                     // Some(8) - rightmost

//=====
// Checking predicates
//=====
```

```

let all_alpha = "hello".chars().all(|c| c.is_alphabetic());
let has_digit = "hello123".chars().any(|c| c.is_numeric());

//=====
// Splitting
//=====
let parts: Vec<&str> = "a,b,c,d".split(',').collect();
// ["a", "b", "c", "d"]

let parts: Vec<&str> = "a::b::c".split("::").collect();
// ["a", "b", "c"]

let parts: Vec<&str> = " a b c ".split_whitespace().collect();
// ["a", "b", "c"] - automatically trims

let (left, right) = "key=value".split_once('=').unwrap();
// ("key", "value")

let lines: Vec<&str> = "line1\nline2\nline3".lines().collect();
// ["line1", "line2", "line3"]

//=====
// Trimming whitespace
//=====
let trimmed = " hello ".trim();           // "hello"
let left = " hello ".trim_start();        // "hello "
let right = " hello ".trim_end();         // " hello"

let custom = "###hello###".trim_matches('#'); // "hello"

```

Character Iteration

Important: Strings are UTF-8 encoded. Never index directly into a string! Use iteration instead.

```

let text = "Hello 世界";

//=====
// Iterate over chars
//=====
for c in text.chars() {
    println!("{}", c);
}

// H, e, l, l, o, , 世, 界

//=====
// Iterate over bytes
//=====
for b in text.bytes() {
    println!("{}", b);
}

// 72, 101, 108, 108, 111, 32, 228, 184, 150, 231, 149, 140

```

```

//=====
// Get char at position (expensive!)
//=====

let third_char = text.chars().nth(2);           // Some('l')

//=====
// Count characters (not bytes)
//=====

let char_count = text.chars().count();          // 8 chars
let byte_count = text.len();                   // 13 bytes

//=====
// Character ranges
//=====

let chars: Vec<char> = ('a'..='z').collect(); // ['a', 'b', ..., 'z']

```

String Slicing (Use with Caution!)

```

let s = "Hello, 世界";

//=====
// Slicing at valid UTF-8 boundaries
//=====

let slice = &s[0..5];                         // "Hello"

//=====
// DANGER: Slicing at invalid boundaries
//=====

// let bad = &s[0..8];                         // Panics! Not a char boundary

//=====
// Safe slicing with get
//=====

let safe = s.get(0..5);                      // Some("Hello")
let bad = s.get(0..8);                        // None (invalid boundary)

//=====
// Finding character boundaries
//=====

if s.is_char_boundary(5) {
    let slice = &s[..5];
}

```

Parsing and Formatting

```

use std::fmt;

//=====
// Parsing from strings
//=====

let num: i32 = "42".parse().unwrap();

```

```

let num: Result<i32, _> = "not a number".parse(); // Err

let float: f64 = "3.14".parse().unwrap();
let boolean: bool = "true".parse().unwrap();

//=====
// Formatting with format!
//=====

let name = "Alice";
let age = 30;

let msg = format!("Name: {}, Age: {}", name, age);
let msg = format!("Name: {name}, Age: {age}"); // Named arguments

//=====
// Format specifiers
//=====

format!("{:>10}", "right"); // "      right" (right-align, width 10)
format!("{:<10}", "left"); // "left      " (left-align)
format!("{:^10}", "center"); // " center " (center)
format!("{:>5}", "42"); // "00042" (pad with zeros)

format!("{:.2}", 3.14159); // "3.14" (2 decimal places)
format!("{:e}", 1000.0); // "1e3" (scientific notation)
format!("{:#x}", 255); // "0xff" (hex with prefix)
format!("{:#b}", 10); // "0b1010" (binary with prefix)

//=====
// Custom Display implementation
//=====

struct Point { x: i32, y: i32 }

impl fmt::Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "({}, {})", self.x, self.y)
    }
}

let p = Point { x: 10, y: 20 };
let s = format!("{}", p); // "(10, 20)"

```

Case Conversion

```

let s = "Hello, World!";

//=====
// Case conversion
//=====

let lower = s.to_lowercase(); // "hello, world!"
let upper = s.to_uppercase(); // "HELLO, WORLD!"

//=====

```

```
// Unicode-aware (handles special cases)
//=====
let german = "Straße";
let upper = german.to_uppercase();           // "STRASSE" (ß → SS)

//=====
// Case checking
//=====
let all_lower = s.chars().all(|c| c.is_lowercase() || !c.is_alphabetic());
```

Regular Expressions (regex crate)

```
//=====
// Requires: regex = "1.0" in Cargo.toml
//=====

use regex::Regex;

//=====
// Creating and matching
//=====
let re = Regex::new(r"\d{4}-\d{2}-\d{2}").unwrap();
let is_match = re.is_match("2024-01-15"); // true

//=====
// Capturing groups
//=====
let re = Regex::new(r"(\d{4})-(\d{2})-(\d{2})").unwrap();
let text = "Date: 2024-01-15";

if let Some(caps) = re.captures(text) {
    let year = &caps[1];                      // "2024"
    let month = &caps[2];                     // "01"
    let day = &caps[3];                      // "15"
}

//=====
// Finding all matches
//=====
let re = Regex::new(r"\d+").unwrap();
for mat in re.find_iter("Numbers: 42, 100, 7") {
    println!("{}", mat.as_str());
}
// "42", "100", "7"

//=====
// Replacing text
//=====
let re = Regex::new(r"\d+").unwrap();
let result = re.replace_all("Id: 123, Code: 456", "XXX");
// "Id: XXX, Code: XXX"
```

Common String Patterns

```
//=====
// Joining strings
//=====

let words = vec!["Hello", "world"];
let sentence = words.join(" ");           // "Hello world"

let numbers = vec![1, 2, 3];
let csv = numbers.iter()
    .map(|n| n.to_string())
    .collect::<Vec<_>>()
    .join(",");                         // "1,2,3"

//=====
// Repeating strings
//=====

let repeated = "abc".repeat(3);          // "abcabcbc"

//=====
// Escaping special chars
//=====

let with_newlines = "Line 1\nLine 2\nLine 3";
let escaped = with_newlines.escape_default().to_string();
// "Line 1\nLine 2\nLine 3"

//=====
// Building strings efficiently
//=====

let mut s = String::with_capacity(100);   // Pre-allocate if you know size
for i in 0..10 {
    s.push_str(&i.to_string());
    s.push(' ');
}
```

Option and Result: Error Handling

Rust uses `Option<T>` for values that might be absent and `Result<T, E>` for operations that might fail. These types replace null pointers and exceptions, making error handling explicit and composable.

Option: Handling Optional Values

```
//=====
// Creating Option values
//=====

let some_value: Option<i32> = Some(42);
let no_value: Option<i32> = None;

//=====
```

```

// Pattern matching (most explicit)
//=====
match some_value {
    Some(x) => println!("Value: {}", x),
    None => println!("No value"),
}

//=====
// if let (single pattern)
//=====
if let Some(x) = some_value {
    println!("Value: {}", x);
}

//=====
// Unwrapping (use with caution!)
//=====
let value = some_value.unwrap();           // Panics if None
let value = some_value.expect("No value"); // Panics with custom message
let value = some_value.unwrap_or(0);        // Provides default
let value = some_value.unwrap_or_else(|| expensive_default());
let value = some_value.unwrap_or_default(); // Uses Default::default()

//=====
// Checking for presence
//=====
if some_value.is_some() {
    println!("Has value");
}

if no_value.is_none() {
    println!("No value");
}

//=====
// Transforming Option
//=====
let doubled = some_value.map(|x| x * 2);   // Some(84)
let none_doubled = no_value.map(|x| x * 2); // None

// and_then for chaining operations that return Option
fn divide(a: i32, b: i32) -> Option<i32> {
    if b == 0 { None } else { Some(a / b) }
}

let result = Some(10)
    .and_then(|x| divide(x, 2))           // Some(5)
    .and_then(|x| divide(x, 0));          // None

//=====
// Filtering
//=====
let value = Some(42);

```

```

let filtered = value.filter(|&x| x > 50); // None (failed predicate)
let kept = value.filter(|&x| x > 30);      // Some(42)

//=====
// Converting between types
//=====

let opt: Option<i32> = Some(42);
let result: Result<i32, &str> = opt.ok_or("No value"); // Ok(42)

let result: Result<i32, &str> = Err("Error");
let opt: Option<i32> = result.ok();           // None (discards error)

//=====
// Borrowing inner value
//=====

let value = Some(String::from("hello"));
let borrowed: Option<&str> = value.as_ref().map(|s| s.as_str());
let length: Option<usize> = value.as_ref().map(|s| s.len());

//=====
// Taking ownership
//=====

let mut value = Some(42);
let taken = value.take();                      // Some(42), value is now None

//=====
// Replacing value
//=====

let mut value = Some(42);
let old = value.replace(100);                  // old is Some(42), value is Some(100)

```

Result<T, E>: Handling Errors

```

use std::fs::File;
use std::io::{self, Read};

//=====
// Functions returning Result
//=====

fn divide(a: i32, b: i32) -> Result<i32, String> {
    if b == 0 {
        Err("Division by zero".to_string())
    } else {
        Ok(a / b)
    }
}

//=====
// Pattern matching
//=====

match divide(10, 2) {
    Ok(result) => println!("Result: {}", result),
}

```

```
    Err(e) => println!("Error: {}", e),
}

//=====
// The ? operator
//=====

fn read_file(path: &str) -> Result<String, io::Error> {
    let mut file = File::open(path)?;           // Returns early if Err
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    Ok(contents)
}

//=====
// Unwrapping (use sparingly!)
//=====

let value = divide(10, 2).unwrap();          // Panics on Err
let value = divide(10, 2).expect("Math error");

//=====
// Providing defaults
//=====

let value = divide(10, 0).unwrap_or(0);
let value = divide(10, 0).unwrap_or_else(|_| expensive_default());
let value = divide(10, 0).unwrap_or_default();

//=====
// Transforming Results
//=====

let doubled = divide(10, 2).map(|x| x * 2); // Ok(10)
let err_mapped = divide(10, 0)
    .map_err(|e| format!("Fatal: {}", e)); // Map error type

//=====
// Chaining operations
//=====

let result = divide(10, 2)
    .and_then(|x| divide(x, 2))           // Ok(2)
    .and_then(|x| divide(x, 0));          // Err

//=====
// Checking status
//=====

if divide(10, 2).is_ok() {
    println!("Success");
}

if divide(10, 0).is_err() {
    println!("Failed");
}

//=====
// Converting between Ok/Err and Option
```

```

//=====
let result: Result<i32, String> = Ok(42);
let opt: Option<i32> = result.ok();           // Some(42), discards error
let err_opt: Option<String> = result.err(); // None

//=====
// Combining multiple Results (all must succeed)
//=====

let r1: Result<i32, &str> = Ok(1);
let r2: Result<i32, &str> = Ok(2);
let combined: Result<Vec<i32>, &str> =
    vec![r1, r2].into_iter().collect();      // Ok(vec![1, 2])

//=====
// Early return on first error
//=====

fn process() -> Result<(), String> {
    divide(10, 2)?;                         // Continue if Ok
    divide(20, 4)?;                         // Continue if Ok
    divide(5, 0)?;                          // Returns Err immediately
    Ok(());                                // Never reached
}

```

Error Propagation Patterns

```

use std::fs::File;
use std::io::{self, Read};

//=====
// Manual error propagation (verbose)
//=====

fn read_username_v1() -> Result<String, io::Error> {
    let f = File::open("username.txt");
    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut s = String::new();
    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}

//=====
// With ? operator (idiomatic)
//=====

fn read_username_v2() -> Result<String, io::Error> {
    let mut f = File::open("username.txt")?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
}

```

```

    Ok(s)
}

//=====
// Chaining with ? (more concise)
//=====

fn read_username_v3() -> Result<String, io::Error> {
    let mut s = String::new();
    File::open("username.txt")?.read_to_string(&mut s)?;
    Ok(s)
}

//=====
// Converting error types with ? and From
//=====

use std::num::ParseIntError;

fn parse_and_double(s: &str) -> Result<i32, ParseIntError> {
    let num: i32 = s.parse()?;
    Ok(num * 2)
}

```

Custom Error Types

```

use std::fmt;
use std::error::Error;

//=====
// Simple error enum
//=====

#[derive(Debug)]
enum MathError {
    DivisionByZero,
    NegativeSquareRoot,
}

impl fmt::Display for MathError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            MathError::DivisionByZero => write!(f, "Division by zero"),
            MathError::NegativeSquareRoot => write!(f, "Square root of negative number"),
        }
    }
}

impl Error for MathError {}

fn safe_divide(a: f64, b: f64) -> Result<f64, MathError> {
    if b == 0.0 {
        Err(MathError::DivisionByZero)
    } else {
        Ok(a / b)
    }
}

```

```

    }

//=====
// Using thiserror crate (recommended)
//=====

// Cargo.toml: thiserror = "1.0"
use thiserror::Error;

#[derive(Error, Debug)]
enum DataError {
    #[error("I/O error: {0}")]
    Io(#[from] std::io::Error),

    #[error("Parse error: {0}")]
    Parse(#[from] std::num::ParseIntError),

    #[error("Invalid data at line {line}: {msg}")]
    Invalid { line: usize, msg: String },
}

```

Combinators Reference

```

//=====
// Option combinators
//=====

let opt = Some(42);

opt.map(|x| x * 2);                      // Some(84)
opt.and_then(|x| Some(x * 2));            // Some(84)
opt.or(Some(0));                          // Some(42)
opt.filter(|&x| x > 50);                 // None
opt.zip(Some(10));                        // Some((42, 10))

//=====
// Result combinators
//=====

let res: Result<i32, &str> = Ok(42);

res.map(|x| x * 2);                      // Ok(84)
res.and_then(|x| Ok(x * 2));             // Ok(84)
res.or(Ok(0));                           // Ok(42)
res.map_err(|e| format!("Error: {}", e)); // Transform error

//=====
// Early returns with ?
//=====

fn compute() -> Result<i32, String> {
    let a = divide(10, 2)?;
    let b = divide(a, 2)?;
}

```

```
Ok(b)  
}
```

Smart Pointers: Box, Rc, Arc, RefCell

Smart pointers provide ownership and borrowing patterns beyond simple references. They enable recursive data structures, shared ownership, and interior mutability while maintaining Rust's safety guarantees.

Box: Heap Allocation

`Box<T>` is the simplest smart pointer—it allocates data on the heap and provides unique ownership.

```
//=====  
// Creating boxed values  
//=====  
let boxed_int = Box::new(42);  
let boxed_string = Box::new(String::from("hello"));  
  
//=====  
// Use cases for Box  
//=====  
  
// 1. Recursive data structures (size must be known)  
#[derive(Debug)]  
enum List {  
    Cons(i32, Box<List>),  
    Nil,  
}  
  
use List::{Cons, Nil};  
  
let list = Cons(1, Box::new(Cons(2, Box::new(Cons(3, Box::new(Nil))))));  
  
// 2. Large values you want to avoid copying  
struct LargeStruct {  
    data: [u8; 1000000],  
}  
  
let large = Box::new(LargeStruct { data: [0; 1000000] });  
// Only the Box pointer is copied, not the data  
  
// 3. Trait objects (dynamic dispatch)  
trait Animal {  
    fn speak(&self);  
}  
  
struct Dog;  
impl Animal for Dog {  
    fn speak(&self) { println!("Woof!"); }  
}
```

```

}

let animal: Box<dyn Animal> = Box::new(Dog);
animal.speak();

//=====
// Accessing boxed values
//=====

let boxed = Box::new(42);
let value = *boxed;                                // Dereference to get value
println!("{}", boxed);                            // Auto-deref for Display

//=====
// Box provides unique ownership
//=====

let boxed = Box::new(42);
let moved = boxed;                                // Ownership transferred
// boxed is now invalid

//=====
// Converting to raw pointer
//=====

let boxed = Box::new(42);
let raw = Box::into_raw(boxed);                  // *mut i32
unsafe {
    println!("{}", *raw);
    let _ = Box::from_raw(raw);                // Must reconstruct to free
}

```

Rc: Reference Counted Shared Ownership

Rc<T> allows multiple owners of the same data through reference counting. **Single-threaded only.**

```

use std::rc::Rc;

//=====
// Creating Rc values
//=====

let rc1 = Rc::new(42);
let rc2 = Rc::clone(&rc1);                      // Increment ref count
let rc3 = rc1.clone();                          // Same as Rc::clone

//=====
// All point to same data
//=====

println!("{}, {}, {}", rc1, rc2, rc3);      // 42, 42, 42

//=====
// Checking reference count
//=====

println!("Count: {}", Rc::strong_count(&rc1)); // 3

```

```

//=====
// Drop decrements count, frees when 0
//=====
drop(rc2);
println!("Count: {}", Rc::strong_count(&rc1)); // 2

//=====
// Use case: Shared graph nodes
//=====
use std::rc::Rc;

struct Node {
    value: i32,
    children: Vec<Rc<Node>>,
}

let leaf = Rc::new(Node { value: 3, children: vec![] });
let node = Rc::new(Node {
    value: 5,
    children: vec![Rc::clone(&leaf)],
});
// leaf is shared between owners

//=====
// Weak references (break cycles)
//=====
use std::rc::{Rc, Weak};

struct Parent {
    children: Vec<Rc<Child>>,
}

struct Child {
    parent: Weak<Parent>,           // Weak doesn't increment count
}

let parent = Rc::new(Parent { children: vec![] });
let child = Rc::new(Child {
    parent: Rc::downgrade(&parent), // Create Weak from Rc
});

// Access weak reference
if let Some(parent_rc) = child.parent.upgrade() {
    // parent_rc is Rc<Parent>
}

```

Arc: Atomic Reference Counted (Thread-Safe)

`Arc<T>` is the thread-safe version of `Rc<T>`, using atomic operations for reference counting.

```

use std::sync::Arc;
use std::thread;

```

```
//=====
// Creating Arc values
//=====

let arc1 = Arc::new(42);
let arc2 = Arc::clone(&arc1);

//=====
// Sharing across threads
//=====

let data = Arc::new(vec![1, 2, 3, 4, 5]);

let handles: Vec<_> = (0..3)
    .map(|i| {
        let data_clone = Arc::clone(&data);
        thread::spawn(move || {
            println!("Thread {}: {:?}", i, data_clone);
        })
    })
    .collect();

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

//=====
// Checking reference count
//=====

println!("Count: {}", Arc::strong_count(&arc1));

//=====
// Use case: Shared immutable state
//=====

use std::sync::Arc;
use std::thread;

struct Config {
    max_connections: usize,
    timeout_ms: u64,
}

let config = Arc::new(Config {
    max_connections: 100,
    timeout_ms: 5000,
});

let config_clone = Arc::clone(&config);
thread::spawn(move || {
    println!("Max connections: {}", config_clone.max_connections);
});

println!("Timeout: {}", config.timeout_ms);
```

RefCell: Interior Mutability

RefCell<T> provides interior mutability—allows mutation through shared references. **Checks borrowing rules at runtime instead of compile time.** Single-threaded only.

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

//=====
// Creating RefCell values
//=====
let cell = RefCell::new(42);

//=====
// Borrowing mutably through shared ref
//=====
{
    let mut borrow = cell.borrow_mut();      // Runtime borrow check
    *borrow += 1;
}

let value = cell.borrow();                  // Immutable borrow
println!("{}", *value);                   // 43

//=====
// DANGER: Runtime panics on borrow violations
//=====
let cell = RefCell::new(42);
let borrow1 = cell.borrow_mut();
// let borrow2 = cell.borrow();           // Panics! Already mutably borrowed

//=====
// Checking borrow state
//=====
if let Ok(value) = cell.try_borrow() {
    println!("{}", *value);
} else {
    println!("Already borrowed mutably");
}

//=====
// Use case: Multiple owners with mutation
//=====
use std::rc::Rc;
use std::cell::RefCell;

struct SharedData {
    value: RefCell<i32>,
}

let data = Rc::new(SharedData {
    value: RefCell::new(0),
});
```

```

let data_clone = Rc::clone(&data);

*data.value.borrow_mut() += 1;
*data_clone.value.borrow_mut() += 1;

println!("{}", data.value.borrow());           // 2

```

Cell: Simple Interior Mutability

`Cell<T>` provides interior mutability for `Copy` types without runtime borrow checking.

```

use std::cell::Cell;

//=====
// Creating Cell values
//=====

let cell = Cell::new(42);

//=====
// Getting and setting
//=====

let value = cell.get();                      // 42 (Copy types only)
cell.set(100);
let new_value = cell.get();                   // 100

//=====
// Swapping and updating
//=====

let old = cell.replace(200);                  // Returns old value
cell.update(|x| x * 2);                     // 400

//=====
// Use case: Counters and flags
//=====

struct Counter {
    count: Cell<u32>,
}

impl Counter {
    fn increment(&self) {                    // Takes &self, not &mut self!
        self.count.set(self.count.get() + 1);
    }

    fn get(&self) -> u32 {
        self.count.get()
    }
}

let counter = Counter { count: Cell::new(0) };
counter.increment();

```

```
counter.increment();
println!("{}", counter.get()); // 2
```

Mutex and RwLock: Thread-Safe Interior Mutability

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

//=====
// Mutex: Exclusive access
//=====

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

for _ in 0..10 {
    let counter_clone = Arc::clone(&counter);
    let handle = thread::spawn(move || {
        let mut num = counter_clone.lock().unwrap();
        *num += 1;
    });
    handles.push(handle);
}

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

println!("Count: {}", *counter.lock().unwrap()); // 10

//=====
// RwLock: Multiple readers or one writer
//=====

let data = Arc::new(RwLock::new(vec![1, 2, 3]));

// Multiple readers
let data_clone1 = Arc::clone(&data);
let reader1 = thread::spawn(move || {
    let vec = data_clone1.read().unwrap();
    println!("{:?}", *vec);
});

let data_clone2 = Arc::clone(&data);
let reader2 = thread::spawn(move || {
    let vec = data_clone2.read().unwrap();
    println!("{:?}", *vec);
});

// One writer
let writer = thread::spawn(move || {
    let mut vec = data.write().unwrap();
    vec.push(4);
});
```

```
reader1.join().unwrap();
reader2.join().unwrap();
writer.join().unwrap();
```

Smart Pointer Selection Guide

| Pointer | Thread-Safe | Ownership | Mutability | Use When |
|------------|-------------|-----------|---------------------|--|
| Box<T> | Yes | Single | Through &mut | Heap allocation, recursive types |
| Rc<T> | No | Shared | Immutable | Multiple owners, single thread |
| Arc<T> | Yes | Shared | Immutable | Multiple owners, multiple threads |
| RefCell<T> | No | Single | Interior mutability | Mutation through shared ref, single thread |
| Cell<T> | No | Single | Interior mutability | Copy types, simple updates |
| Mutex<T> | Yes | Shared | Interior mutability | Shared mutable state across threads |
| RwLock<T> | Yes | Shared | Interior mutability | Many readers, few writers, across threads |

Common Combinations: - `Rc<RefCell<T>>`: Multiple owners with mutation (single-threaded) - `Arc<Mutex<T>>`: Multiple owners with mutation (multi-threaded) - `Arc<RwLock<T>>`: Multiple readers, occasional writers (multi-threaded)

Cargo Commands Reference

Cargo is Rust's build system and package manager, handling compilation, dependency management, testing, and publishing. Understanding Cargo's command structure transforms your development workflow from managing files to orchestrating projects.

Think of Cargo as your project lifecycle manager. It creates scaffolding, fetches dependencies, invokes the compiler, runs tests, generates documentation, and publishes crates. Every Rust project beyond a single-file script benefits from Cargo's conventions and automation.

Project Initialization

Starting a new project with Cargo creates the standard structure that the entire ecosystem expects:

```
# Create binary (application) project
cargo new my_app
# Creates:
# my_app/
#   ├── Cargo.toml      # Manifest file
#   └── src/
#       └── main.rs     # Binary entry point with fn main()

# Create library project
cargo new my_lib --lib
# Creates:
# my_lib/
```

```

# └── Cargo.toml
#   └── src/
#     └── lib.rs      # Library root

# Initialize in existing directory
cd existing_project
cargo init
cargo init --lib # For library

# Project naming conventions
cargo new snake_case_name      # Preferred: uses underscores
cargo new kebab-case-name       # Also works: converted to underscores

```

The `Cargo.toml` manifest is your project's metadata and dependency specification:

```

[package]
name = "my_app"
version = "0.1.0"
edition = "2021"          # Rust edition (2015, 2018, 2021)
authors = ["Your Name <you@example.com>"]
license = "MIT"
description = "A brief description"
repository = "https://github.com/user/my_app"

[dependencies]
# Dependencies from crates.io
serde = "1.0"
tokio = { version = "1.0", features = ["full"] }

# Git dependencies
my_utils = { git = "https://github.com/user/my_utils" }

# Local path dependencies
shared = { path = "../shared" }

[dev-dependencies]
# Test-only dependencies
proptest = "1.0"
criterion = "0.5"

[build-dependencies]
# Build script dependencies
cc = "1.0"

```

Building and Running

Cargo manages the compile-run cycle with sensible defaults for development and release builds:

```

# Check code for errors (fastest – no codegen)
cargo check
# Use this constantly during development

```

```

# Runs type checking and borrow checker without producing binaries

# Build in debug mode (default)
cargo build
# Produces unoptimized binary at target/debug/my_app
# Fast compilation, slow execution, includes debug symbols

# Build in release mode
cargo build --release
# Produces optimized binary at target/release/my_app
# Slow compilation, fast execution, strips debug info

# Run the binary (builds if needed)
cargo run
cargo run --release
cargo run -- arg1 arg2      # Pass arguments to binary
cargo run --bin other_binary # Run specific binary

# Build and run examples
cargo run --example my_example

# Build for specific target
cargo build --target x86_64-unknown-linux-musl

```

Understanding debug vs release:

- **Debug builds** (`cargo build`): Fast compile, slow runtime, large binaries
 - Optimizations: Off (opt-level = 0)
 - Debug symbols: Included
 - Overflow checks: Enabled
 - Use during development
- **Release builds** (`cargo build --release`): Slow compile, fast runtime, small binaries
 - Optimizations: Full (opt-level = 3)
 - Debug symbols: Stripped
 - Overflow checks: Disabled
 - Use for production, benchmarks, performance testing

Testing

Rust's testing is built into the language and integrated with Cargo:

```

# Run all tests
cargo test

# Run tests matching pattern
cargo test test_addition      # Runs tests with "test_addition" in name
cargo test integration::      # Runs tests in integration module

# Run tests in specific file
cargo test --test integration_tests

```

```

# Run doc tests only
cargo test --doc

# Run tests with output (show println!)
cargo test -- --nocapture

# Run tests single-threaded
cargo test -- --test-threads=1

# Run ignored tests
cargo test -- --ignored

# Run benchmarks
cargo bench

```

Test organization:

```

//=====
// Unit tests (in same file as code)
//=====

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

    #[test]
    fn test_addition() {
        assert_eq!(2 + 2, 4);
    }

    #[test]
    #[should_panic]
    fn test_panic() {
        panic!("Expected panic");
    }

    #[test]
    #[ignore] // Skip unless --ignored specified
    fn expensive_test() {
        // ...
    }
}

//=====
// Integration tests (in tests/ directory)
//=====

#[test]
fn test_public_api() {
    use my_lib::public_function;
    assert!(public_function());
}

```

Dependency Management

Cargo handles transitive dependencies, version resolution, and lock files automatically:

```
# Add dependency (modifies Cargo.toml)
cargo add serde
cargo add tokio --features full
cargo add serde_json --dev          # Dev dependency
cargo add cc --build               # Build dependency

# Update dependencies to latest compatible versions
cargo update
cargo update serde                # Update specific dependency

# Remove dependency
cargo remove serde

# Display dependency tree
cargo tree
cargo tree -i serde              # Show inverse dependencies (what needs serde)

# Check for outdated dependencies
cargo outdated # Requires cargo-outdated plugin
```

Version specification syntax:

```
[dependencies]
# Caret (default): Compatible updates
serde = "^1.2.3"    # >=1.2.3, <2.0.0
serde = "1.2.3"      # Same as above (^ is implicit)

# Tilde: Patch updates only
serde = "~1.2.3"     # >=1.2.3, <1.3.0

# Exact version
serde = "=1.2.3"     # Exactly 1.2.3

# Wildcard
serde = "1.2.*"       # >=1.2.0, <1.3.0

# Comparison operators
serde = ">1.2.0"
serde = ">=1.2.0, <2.0.0"
```

The `Cargo.lock` file pins exact versions for reproducible builds: - **Checked into version control** for binaries (ensures same build everywhere) - **Not checked in** for libraries (allows dependents to use newer compatible versions)

Documentation

Cargo generates HTML documentation from your doc comments:

```
# Generate and open documentation
cargo doc --open

# Include private items
cargo doc --document-private-items

# Generate without dependencies
cargo doc --no-deps
```

Writing doc comments:

```
/// Adds two numbers together.
///
/// # Examples
///
/// ```
/// assert_eq!(my_lib::add(2, 3), 5);
/// ```
///
/// # Panics
///
/// Panics if the sum overflows.
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

Publishing and Packaging

Cargo integrates with crates.io for publishing libraries:

```
# Login to crates.io (one-time setup)
cargo login YOUR_API_TOKEN

# Package for distribution (creates .crate file)
cargo package

# Dry-run publish (check for errors)
cargo publish --dry-run

# Publish to crates.io
cargo publish

# Yank a version (prevents new projects from using it)
cargo yank --vers 0.1.0
cargo yank --vers 0.1.0 --undo  # Un-yank
```

Workspaces: Multi-Crate Projects

Workspaces manage multiple related crates in a single repository:

```
# Workspace Cargo.toml (at repository root)
[workspace]
members = [
    "server",
    "client",
    "shared",
]

# Individual crates have their own Cargo.toml files
# server/Cargo.toml
[package]
name = "server"
version = "0.1.0"

[dependencies]
shared = { path = "../shared" }
```

```
# Build all workspace crates
cargo build --workspace

# Run specific workspace binary
cargo run -p server

# Test all workspace crates
cargo test --workspace
```

Advanced Commands

```
# Clean build artifacts
cargo clean

# Format code
cargo fmt
cargo fmt -- --check # Check formatting without modifying

# Lint with Clippy
cargo clippy
cargo clippy -- -D warnings # Fail on warnings

# Expand macros
cargo expand           # Requires cargo-expand

# View assembly
cargo asm             # Requires cargo-asm

# Security audit
cargo audit           # Requires cargo-audit

# Show why a dependency is included
cargo tree -i dependency_name
```

```
# Fix compiler warnings automatically
cargo fix

# Vendoring dependencies (bundle all dependencies)
cargo vendor
```

Quick Reference Table

| Command | Purpose | Common Options |
|--|-------------------------------|--|
| <code>cargo new <name></code> | Create new project | <code>--lib, --vcs none</code> |
| <code>cargo init</code> | Initialize in existing dir | <code>--lib</code> |
| <code>cargo build</code> | Compile project | <code>--release, --target</code> |
| <code>cargo run</code> | Build and run binary | <code>--release, -- <args></code> |
| <code>cargo check</code> | Check code without building | Fast feedback during development |
| <code>cargo test</code> | Run tests | <code>--test <name>, -- --nocapture</code> |
| <code>cargo bench</code> | Run benchmarks | Requires <code># [bench]</code> or criterion |
| <code>cargo doc</code> | Generate documentation | <code>--open, --no-deps</code> |
| <code>cargo add <crate></code> | Add dependency | <code>--dev, --build, --features</code> |
| <code>cargo update</code> | Update dependencies | Updates to latest compatible |
| <code>cargo tree</code> | Show dependency tree | <code>-i <crate></code> for inverse deps |
| <code>cargo clean</code> | Remove build artifacts | Frees disk space |
| <code>cargo fmt</code> | Format code | <code>-- --check</code> for CI |
| <code>cargo clippy</code> | Lint code | <code>-- -D warnings</code> to fail on warn |
| <code>cargo publish</code> | Publish to crates.io | <code>--dry-run</code> to test first |
| <code>cargo search <query></code> | Search crates.io | |
| <code>cargo install <crate></code> | Install binary crate globally | |

Environment Variables and Configuration

Cargo respects environment variables for customization:

```
# Increase parallelism
CARGO_BUILD_JOBS=8 cargo build

# Use offline mode (don't fetch updates)
CARGO_NET_OFFLINE=true cargo build

# Custom registry
CARGO_REGISTRY_DEFAULT=my-registry cargo build
```

```
# Target directory (useful for CI caching)
CARGO_TARGET_DIR=/tmp/target cargo build
```

Configuration in `~/.cargo/config.toml`:

```
[build]
jobs = 8
target-dir = "/tmp/cargo-target"

[term]
color = "always"

[net]
git-fetch-with-cli = true

[alias]
b = "build"
r = "run"
t = "test"
```

Conclusion

This quick reference captures the patterns you'll use daily in production Rust development. Type conversions establish clear boundaries between domains. Trait implementations make your types composable with the standard library. Iterator combinators transform data processing from imperative loops into declarative pipelines. Cargo commands orchestrate your entire project lifecycle.

Keep this appendix close as you write code. The patterns become muscle memory through repetition, but having a concise reference accelerates learning and prevents subtle mistakes. When you encounter unfamiliar territory, use these examples as starting points and refer back to the detailed chapters for deeper understanding.

Appendix B: Design Pattern Catalog

Creational Patterns:

- [Builder Pattern](#)
- [Factory Pattern](#)
- [Singleton Pattern](#)
- [Prototype Pattern](#)

Structural Patterns:

- [Adapter Pattern](#)
- [Decorator Pattern](#)
- [Facade Pattern](#)
- [Newtype Pattern](#)

Behavioral Patterns:

- [Strategy Pattern](#)
- [Observer Pattern](#)
- [Command Pattern](#)
- [Iterator Pattern](#)

Concurrency Patterns:

- [Thread Pool Pattern](#)
- [Producer-Consumer Pattern](#)
- [Fork-Join Pattern](#)
- [Actor Pattern](#)
- [Async/Await Pattern](#)

Design patterns are reusable solutions to common programming problems. They're not finished code you can copy, but templates for how to solve a problem in many different situations. In Rust, classic design patterns take on unique characteristics due to the language's ownership model, zero-cost abstractions, and powerful type system.

This catalog adapts traditional object-oriented patterns to Rust's paradigm while introducing patterns unique to systems programming and Rust's ecosystem. Some patterns that require runtime polymorphism in other languages can be implemented at compile-time in Rust through generics and traits. Others that rely on shared mutable state require different approaches due to Rust's borrowing rules.

The patterns are organized into four categories:

Creational patterns control object creation, managing complexity when instantiating objects requires more than simple construction. In Rust, these often leverage the type system to enforce invariants at compile-time.

Structural patterns organize relationships between entities, composing objects to form larger structures. Rust's trait system and zero-cost abstractions enable elegant implementations without runtime overhead.

Behavioral patterns focus on communication between objects, defining how they interact and distribute responsibility. Rust's ownership model influences how we implement delegation and message passing.

Concurrency patterns address the challenges of parallel and asynchronous execution. Rust's fearless concurrency model, with its compile-time race detection, enables patterns that would be unsafe in other languages.

Each pattern includes: - **Intent:** What problem does it solve? - **Motivation:** When and why would you use it? - **Implementation:** Idiomatic Rust code - **Trade-offs:** Advantages and limitations -

Variations: Common adaptations

Creational Patterns

Creational patterns abstract the instantiation process, making systems independent of how objects are created, composed, and represented. In Rust, these patterns often encode constraints in the type system, moving validation from runtime to compile-time.

Builder Pattern

Intent: Separate the construction of a complex object from its representation, allowing the same construction process to create different representations.

Motivation: When creating an object requires many optional parameters or complex initialization logic, constructors become unwieldy. The builder pattern provides a fluent interface for step-by-step construction, improving readability and enabling compile-time validation of required fields.

```
//=====
// Problem: Too many constructor parameters
//=====

struct HttpClient {
    base_url: String,
    timeout: Duration,
    user_agent: String,
    max_retries: u32,
    follow_redirects: bool,
    compression: bool,
}

//=====
// Unwieldy constructor
//=====

impl HttpClient {
    fn new(
        base_url: String,
        timeout: Duration,
        user_agent: String,
        max_retries: u32,
        follow_redirects: bool,
        compression: bool,
    ) -> Self {
        // ...
    }
}

//=====
// Solution: Builder pattern
//=====

struct HttpClientBuilder {
    base_url: String,
    timeout: Option<Duration>,
    user_agent: Option<String>,
    max_retries: Option<u32>,
```

```
    follow_redirects: bool,
    compression: bool,
}

impl HttpClientBuilder {
    fn new(base_url: impl Into<String>) -> Self {
        Self {
            base_url: base_url.into(),
            timeout: None,
            user_agent: None,
            max_retries: None,
            follow_redirects: true,
            compression: true,
        }
    }

    fn timeout(mut self, duration: Duration) -> Self {
        self.timeout = Some(duration);
        self
    }

    fn user_agent(mut self, agent: impl Into<String>) -> Self {
        self.user_agent = Some(agent.into());
        self
    }

    fn max_retries(mut self, retries: u32) -> Self {
        self.max_retries = Some(retries);
        self
    }

    fn follow_redirects(mut self, follow: bool) -> Self {
        self.follow_redirects = follow;
        self
    }

    fn build(self) -> HttpClient {
        HttpClient {
            base_url: self.base_url,
            timeout: self.timeout.unwrap_or(Duration::from_secs(30)),
            user_agent: self.user_agent.unwrap_or_else(|| "RustClient/1.0".to_string()),
            max_retries: self.max_retries.unwrap_or(3),
            follow_redirects: self.follow_redirects,
            compression: self.compression,
        }
    }
}

//=====
// Usage: Fluent and readable
//=====

let client = HttpClientBuilder::new("https://api.example.com")
    .timeout(Duration::from_secs(60))
```

```
.max_retries(5)
.user_agent("MyApp/2.0")
.build();
```

Advanced: Typestate pattern for compile-time validation

The typestate pattern uses phantom types to encode state in the type system, catching errors at compile-time:

```
use std::marker::PhantomData;

//=====
// State markers
//=====
struct Incomplete;
struct Complete;

struct ConfigBuilder<S> {
    host: Option<String>,
    port: Option<u16>,
    _state: PhantomData<S>,
}

impl ConfigBuilder<Incomplete> {
    fn new() -> Self {
        Self {
            host: None,
            port: None,
            _state: PhantomData,
        }
    }

    fn host(mut self, host: String) -> Self {
        self.host = Some(host);
        self
    }

    fn port(self, port: u16) -> ConfigBuilder<Complete> {
        ConfigBuilder {
            host: self.host,
            port: Some(port),
            _state: PhantomData,
        }
    }
}

//=====
// build() only available on Complete state
//=====

impl ConfigBuilder<Complete> {
    fn build(self) -> Config {
        Config {
```

```

        host: self.host.unwrap(),
        port: self.port.unwrap(),
    }
}

struct Config {
    host: String,
    port: u16,
}

//=====
// Compile-time error if required fields missing
//=====
// let config = ConfigBuilder::new().build(); // Error: no method `build`
let config = ConfigBuilder::new()
    .host("localhost".to_string())
    .port(8080)
    .build(); // OK

```

Trade-offs: - **Pros:** Ergonomic API, clear intent, optional parameters, compile-time validation with typestate - **Cons:** More boilerplate, separate builder type, consumes self (can't reuse builder)

When to use: Complex initialization, many optional parameters, validation requirements, library APIs

Factory Pattern

Intent: Define an interface for creating objects, but let subclasses or implementors decide which concrete type to instantiate.

Motivation: When the exact type of object to create isn't known until runtime, or when creation logic needs to be abstracted, factories encapsulate instantiation. In Rust, this typically uses trait objects or enums rather than inheritance.

```

//=====
// Abstract product
//=====
trait Button {
    fn render(&self) -> String;
    fn on_click(&self);
}

//=====
// Concrete products
//=====
struct WindowsButton;
impl Button for WindowsButton {
    fn render(&self) -> String {
        "Rendering Windows button".to_string()
    }
    fn on_click(&self) {

```

```

        println!("Windows click event");
    }
}

struct MacButton;
impl Button for MacButton {
    fn render(&self) -> String {
        "Rendering Mac button".to_string()
    }
    fn on_click(&self) {
        println!("Mac click event");
    }
}

//=====
// Factory interface
//=====

trait UIFactory {
    fn create_button(&self) -> Box<dyn Button>;
}

//=====
// Concrete factories
//=====

struct WindowsFactory;
impl UIFactory for WindowsFactory {
    fn create_button(&self) -> Box<dyn Button> {
        Box::new(WindowsButton)
    }
}

struct MacFactory;
impl UIFactory for MacFactory {
    fn create_button(&self) -> Box<dyn Button> {
        Box::new(MacButton)
    }
}

//=====
// Client code
//=====

fn render_ui(factory: &dyn UIFactory) {
    let button = factory.create_button();
    println!("{}", button.render());
    button.on_click();
}

//=====
// Usage
//=====

let factory: Box<dyn UIFactory> = if cfg!(target_os = "windows") {
    Box::new(WindowsFactory)
} else {

```

```
    Box::new(MacFactory)
};

render_ui(&factory);
```

Rust idiom: Enum-based factory (zero-cost)

When the set of types is closed, enums provide a zero-cost alternative:

```
enum Button {
    Windows(WindowsButton),
    Mac(MacButton),
}

impl Button {
    fn new(platform: Platform) -> Self {
        match platform {
            Platform::Windows => Button::Windows(WindowsButton),
            Platform::Mac => Button::Mac(MacButton),
        }
    }

    fn render(&self) -> String {
        match self {
            Button::Windows(btn) => btn.render(),
            Button::Mac(btn) => btn.render(),
        }
    }
}

enum Platform {
    Windows,
    Mac,
}

//=====
// No heap allocation, no dynamic dispatch
//=====

let button = Button::new(Platform::Windows);
```

Trade-offs: - **Trait objects:** Runtime polymorphism, heap allocation, open for extension - **Enums:** Compile-time dispatch, zero-cost, closed set of types

When to use: Plugin systems, platform abstraction, testing (mock factories), runtime type selection

Singleton Pattern

Intent: Ensure a class has only one instance and provide a global point of access to it.

Motivation: Some resources should exist only once: configuration, connection pools, logging systems. Rust's ownership model makes traditional singletons challenging, but several idiomatic alternatives exist.

Rust approach: Static with lazy initialization

```
use std::sync::OnceLock;

struct AppConfig {
    api_key: String,
    debug_mode: bool,
}

impl AppConfig {
    fn global() -> &'static AppConfig {
        static CONFIG: OnceLock<AppConfig> = OnceLock::new();
        CONFIG.get_or_init(|| {
            AppConfig {
                api_key: std::env::var("API_KEY").unwrap_or_default(),
                debug_mode: cfg!(debug_assertions),
            }
        })
    }
}

//=====
// Usage: Thread-safe, initialized once
//=====

let config = AppConfig::global();
println!("API Key: {}", config.api_key);
```

Alternative: Dependency injection (preferred)

Rather than global state, pass dependencies explicitly:

```
struct Database {
    connection_string: String,
}

impl Database {
    fn new(connection_string: String) -> Self {
        Self { connection_string }
    }
}

struct UserService<'a> {
    db: &'a Database,
}

impl<'a> UserService<'a> {
    fn new(db: &'a Database) -> Self {
        Self { db }
    }
}

//=====
```

```
// Explicit dependencies, testable, no global state
//=====
let db = Database::new("postgres://localhost".to_string());
let service = UserService::new(&db);
```

Trade-offs: - **OnceLock**: Simple, thread-safe, but global mutable state is discouraged in Rust -

Dependency injection: More flexible, testable, but requires passing dependencies - **Avoid**: Lazy static crates if OnceLock suffices (std library is preferred)

When to use: Truly global resources (logging, metrics), avoid when possible in favor of dependency injection

Prototype Pattern

Intent: Create new objects by cloning existing instances rather than constructing from scratch.

Motivation: When object creation is expensive (parsing, network calls, complex initialization), cloning an existing instance can be faster. Rust's **Clone** trait makes this pattern native to the language.

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

#[derive(Clone)]
struct TemplateEngine {
    templates: HashMap<String, String>,
    config: Config,
}

#[derive(Clone)]
struct Config {
    strict_mode: bool,
    cache_enabled: bool,
}

impl TemplateEngine {
    fn new() -> Self {
        let mut templates = HashMap::new();
        // Expensive initialization
        templates.insert("header".to_string(), load_template("header.html"));
        templates.insert("footer".to_string(), load_template("footer.html"));

        Self {
            templates,
            config: Config {
                strict_mode: true,
                cache_enabled: true,
            },
        }
    }

    fn with_different_config(&self, config: Config) -> Self {
        let mut cloned = self.clone();
```

```

        cloned.config = config;
        cloned // Reuses expensive template loading
    }
}

fn load_template(_name: &str) -> String {
    // Expensive operation
    String::from("template content")
}

//=====
// Usage
//=====
let base_engine = TemplateEngine::new(); // Expensive

//=====
// Cheap clones with variations
//=====
let dev_engine = base_engine.with_different_config(Config {
    strict_mode: false,
    cache_enabled: false,
});
let prod_engine = base_engine.clone(); // Reuses all data

```

Deep vs shallow cloning

```

use std::rc::Rc;

#[derive(Clone)]
struct SharedData {
    // Shallow clone: reference counted
    cache: Rc<Vec<u8>>,
    // Deep clone: clones the String
    user_id: String,
}

let original = SharedData {
    cache: Rc::new(vec![1, 2, 3]),
    user_id: "user123".to_string(),
};

let cloned = original.clone();
//=====
// cache is shared (reference count increased)
//=====
// user_id is copied
assert_eq!(Rc::strong_count(&original.cache), 2);

```

Trade-offs: - **Pros:** Reduces initialization cost, isolates complex object creation - **Cons:** Deep cloning can be expensive, shared state with Rc requires care

When to use: Expensive initialization, template objects, copy-on-write scenarios

Structural Patterns

Structural patterns compose objects and types to form larger structures while keeping the system flexible and efficient. Rust's trait system and zero-cost abstractions enable elegant structural compositions without runtime overhead.

Adapter Pattern

Intent: Convert the interface of a type into another interface that clients expect, allowing incompatible interfaces to work together.

Motivation: When integrating third-party libraries or legacy code, interfaces often don't match your requirements. Adapters wrap existing types to provide the interface you need without modifying the original.

```
//=====
// Target interface your code expects
//=====

trait MediaPlayer {
    fn play(&self, filename: &str);
}

//=====
// Existing third-party library with different interface
//=====

struct VlcPlayer;
impl VlcPlayer {
    fn play_vlc(&self, file_path: &str) {
        println!("Playing VLC: {}", file_path);
    }
}

struct Mp3Player;
impl Mp3Player {
    fn play_mp3(&self, file_name: &str) {
        println!("Playing MP3: {}", file_name);
    }
}

//=====
// Adapters to make them compatible
//=====

struct VlcAdapter {
    player: VlcPlayer,
}

impl MediaPlayer for VlcAdapter {
    fn play(&self, filename: &str) {
        self.player.play_vlc(filename);
    }
}
```

```

}

struct Mp3Adapter {
    player: Mp3Player,
}

impl MediaPlayer for Mp3Adapter {
    fn play(&self, filename: &str) {
        self.player.play_mp3(filename);
    }
}

//=====
// Client code works with uniform interface
//=====

fn play_media(player: &dyn MediaPlayer, file: &str) {
    player.play(file);
}

//=====
// Usage
//=====

let vlc = VlcAdapter { player: VlcPlayer };
let mp3 = Mp3Adapter { player: Mp3Player };
play_media(&vlc, "video.mp4");
play_media(&mp3, "song.mp3");

```

Zero-cost adapter with generics

```

struct GenericAdapter<T> {
    inner: T,
}

trait PlayVlc {
    fn play_vlc(&self, path: &str);
}

impl<T: PlayVlc> MediaPlayer for GenericAdapter<T> {
    fn play(&self, filename: &str) {
        self.inner.play_vlc(filename);
    }
}

//=====
// No trait object overhead, monomorphized at compile-time
//=====


```

Trade-offs: - **Pros:** Preserves single responsibility, enables interface compatibility - **Cons:** Additional layer of indirection, potential heap allocation with trait objects

When to use: Third-party integration, legacy code adaptation, interface standardization

Decorator Pattern

Intent: Attach additional responsibilities to an object dynamically, providing a flexible alternative to subclassing for extending functionality.

Motivation: When you need to add behavior to individual objects without affecting other instances, decorators wrap objects with new capabilities. In Rust, this often uses trait objects or generic wrappers.

```
trait DataSource {
    fn read(&self) -> String;
    fn write(&mut self, data: &str);
}

//=====
// Concrete component
//=====

struct FileDataSource {
    filename: String,
    contents: String,
}

impl DataSource for FileDataSource {
    fn read(&self) -> String {
        self.contents.clone()
    }

    fn write(&mut self, data: &str) {
        self.contents = data.to_string();
    }
}

//=====
// Decorator: Encryption
//=====

struct EncryptionDecorator {
    wrapped: Box,
}

impl DataSource for EncryptionDecorator {
    fn read(&self) -> String {
        let encrypted = self.wrapped.read();
        decrypt(&encrypted) // Add decryption behavior
    }

    fn write(&mut self, data: &str) {
        let encrypted = encrypt(data); // Add encryption behavior
        self.wrapped.write(&encrypted);
    }
}

//=====
// Decorator: Compression
//=====
```

```

//=====
struct CompressionDecorator {
    wrapped: Box<dyn DataSource>,
}

impl DataSource for CompressionDecorator {
    fn read(&self) -> String {
        let compressed = self.wrapped.read();
        decompress(&compressed)
    }

    fn write(&mut self, data: &str) {
        let compressed = compress(data);
        self.wrapped.write(&compressed);
    }
}

fn encrypt(data: &str) -> String { format!("encrypted({})", data) }
fn decrypt(data: &str) -> String {
    data.trim_start_matches("encrypted(").trim_end_matches(')').to_string() }
fn compress(data: &str) -> String { format!("compressed({})", data) }
fn decompress(data: &str) -> String {
    data.trim_start_matches("compressed(").trim_end_matches(')').to_string() }

//=====
// Usage: Stack decorators
//=====

let file = FileDataSource {
    filename: "data.txt".to_string(),
    contents: "sensitive data".to_string(),
};

let mut source: Box<dyn DataSource> = Box::new(file);
source = Box::new(EncryptionDecorator { wrapped: source });
source = Box::new(CompressionDecorator { wrapped: source });

source.write("secret");
//=====
// Writes: compressed(encrypted(secret))
//=====
```

Type-safe decorator with generics

```

struct Encrypted<T>(T);
struct Compressed<T>(T);

trait Read {
    fn read(&self) -> String;
}

impl Read for String {
    fn read(&self) -> String {
        self.clone()
```

```

    }

}

impl<T: Read> Read for Encrypted<T> {
    fn read(&self) -> String {
        decrypt(&self.0.read())
    }
}

impl<T: Read> Read for Compressed<T> {
    fn read(&self) -> String {
        decompress(&self.0.read())
    }
}

//=====
// Compile-time composition, zero-cost
//=====

let data = String::from("data");
let secure = Compressed(Encrypted(data));
println!("{}", secure.read());

```

Trade-offs: - **Trait objects:** Runtime flexibility, heap allocation - **Generic wrappers:** Compile-time composition, zero-cost, but fixed at compile-time

When to use: Logging wrappers, encryption/compression layers, middleware, cross-cutting concerns

Facade Pattern

Intent: Provide a unified, simplified interface to a complex subsystem, making the subsystem easier to use.

Motivation: Complex libraries or modules can have dozens of types and methods. A facade presents a clean, high-level API that hides the complexity, making common use cases simple while still allowing access to advanced features when needed.

```

//=====
// Complex subsystem
//=====

mod video_processing {
    pub struct VideoDecoder;
    impl VideoDecoder {
        pub fn decode(&self, _file: &str) -> Vec<u8> {
            println!("Decoding video...");
            vec![1, 2, 3]
        }
    }

    pub struct AudioExtractor;
    impl AudioExtractor {
        pub fn extract(&self, _data: &[u8]) -> Vec<u8> {

```

```

        println!("Extracting audio...");  

        vec![4, 5, 6]
    }  

}  
  

pub struct CodecManager;  

impl CodecManager {  

    pub fn configure(&self) {  

        println!("Configuring codecs...");  

    }
}  

}  
  

pub struct FormatConverter;  

impl FormatConverter {  

    pub fn convert(&self, _data: &[u8], _format: &str) -> String {  

        println!("Converting format...");  

        "output.mp4".to_string()
    }
}  

}  

}  
  

//=====  

// Facade: Simple interface  

//=====  

struct VideoConverter {  

    decoder: video_processing::VideoDecoder,  

    audio: video_processing::AudioExtractor,  

    codec: video_processing::CodecManager,  

    converter: video_processing::FormatConverter,
}  

  

impl VideoConverter {  

    fn new() -> Self {  

        Self {
            decoder: video_processing::VideoDecoder,  

            audio: video_processing::AudioExtractor,  

            codec: video_processing::CodecManager,  

            converter: video_processing::FormatConverter,
        }
    }
}  

  

// Simple API for common use case  

fn convert_to_mp4(&self, filename: &str) -> String {  

    self.codec.configure();  

    let video_data = self.decoder.decode(filename);  

    let audio_data = self.audio.extract(&video_data);  

    self.converter.convert(&audio_data, "mp4")
}
}  

}  
  

//=====  

// Client code: Simple and clean  

//=====

```

```
let converter = VideoConverter::new();
let output = converter.convert_to_mp4("input.avi");
println!("Converted: {}", output);
```

Trade-offs: - **Pros:** Simplifies complex APIs, reduces coupling, easier testing - **Cons:** May hide useful functionality, another layer of abstraction

When to use: Complex third-party libraries, legacy system integration, API simplification

Newtype Pattern

Intent: Create a distinct type by wrapping an existing type in a single-field struct, enabling type safety and trait implementation.

Motivation: Type aliases provide no safety—a `UserId` alias for `u64` can be accidentally mixed with `ProductId`. Newtypes create distinct types at compile-time with zero runtime cost, preventing bugs and enabling custom trait implementations.

```
//=====
// Problem: Type aliases don't prevent mixing
//=====

type Meters = f64;
type Feet = f64;

fn calculate_distance(m: Meters) -> Meters { m * 2.0 }

let feet: Feet = 10.0;
let result = calculate_distance(feet); // Compiles but wrong!

//=====
// Solution: Newtype pattern
//=====

#[derive(Debug, Clone, Copy, PartialEq)]
struct Meters(f64);

#[derive(Debug, Clone, Copy, PartialEq)]
struct Feet(f64);

impl Meters {
    fn new(value: f64) -> Self {
        Meters(value)
    }

    fn to_feet(self) -> Feet {
        Feet(self.0 * 3.28084)
    }
}

impl Feet {
    fn new(value: f64) -> Self {
        Feet(value)
    }
}
```

```

}

fn to_meters(self) -> Meters {
    Meters(self.0 / 3.28084)
}
}

fn calculate_distance_safe(m: Meters) -> Meters {
    Meters(m.0 * 2.0)
}

let feet = Feet::new(10.0);
//=====
// let result = calculate_distance_safe(feet); // Compile error!
//=====

let meters = feet.to_meters();
let result = calculate_distance_safe(meters); // OK

```

Implementing external traits

Rust's orphan rule prevents implementing external traits for external types, but newtypes provide a workaround:

```

use std::fmt;

//=====
// Can't implement Display for Vec<i32> directly (orphan rule)
//=====

// impl fmt::Display for Vec<i32> { } // Error!

//=====
// Newtype enables custom implementation
//=====

struct IntList(Vec<i32>);

impl fmt::Display for IntList {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "[")?;
        for (i, item) in self.0.iter().enumerate() {
            if i > 0 { write!(f, ", ")?; }
            write!(f, "{}", item)?;
        }
        write!(f, "]")
    }
}

let list = IntList(vec![1, 2, 3]);
println!("{}", list); // [1, 2, 3]

```

Trade-offs: - **Pros:** Zero-cost type safety, enables trait implementations, prevents mixing incompatible values - **Cons:** Requires explicit construction/extraction, more verbose than type aliases

When to use: Domain modeling (UserId, Email), unit types (Meters, Seconds), orphan rule workaround

Behavioral Patterns

Behavioral patterns define how objects interact and distribute responsibility. They describe not just patterns of objects or classes, but also the patterns of communication between them. Rust's ownership model influences these patterns significantly.

Strategy Pattern

Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Motivation: When multiple algorithms exist for a specific task, strategy pattern allows selecting the algorithm at runtime. In Rust, this uses trait objects for runtime polymorphism or generics for compile-time selection.

```
//=====
// Strategy trait
//=====

trait CompressionStrategy {
    fn compress(&self, data: &[u8]) -> Vec<u8>;
    fn decompress(&self, data: &[u8]) -> Vec<u8>;
}

//=====
// Concrete strategies
//=====

struct ZipCompression;
impl CompressionStrategy for ZipCompression {
    fn compress(&self, data: &[u8]) -> Vec<u8> {
        println!("ZIP compressing {} bytes", data.len());
        data.to_vec() // Simplified
    }

    fn decompress(&self, data: &[u8]) -> Vec<u8> {
        println!("ZIP decompressing {} bytes", data.len());
        data.to_vec()
    }
}

struct RarCompression;
impl CompressionStrategy for RarCompression {
    fn compress(&self, data: &[u8]) -> Vec<u8> {
        println!("RAR compressing {} bytes", data.len());
        data.to_vec()
    }

    fn decompress(&self, data: &[u8]) -> Vec<u8> {
        println!("RAR decompressing {} bytes", data.len());
    }
}
```

```

        data.to_vec()
    }
}

//=====
// Context
//=====
struct FileCompressor {
    strategy: Box<dyn CompressionStrategy>,
}

impl FileCompressor {
    fn new(strategy: Box<dyn CompressionStrategy>) -> Self {
        Self { strategy }
    }

    fn set_strategy(&mut self, strategy: Box<dyn CompressionStrategy>) {
        self.strategy = strategy;
    }

    fn compress_file(&self, data: &[u8]) -> Vec<u8> {
        self.strategy.compress(data)
    }
}

//=====
// Usage: Runtime strategy selection
//=====

let data = vec![1, 2, 3, 4, 5];
let mut compressor = FileCompressor::new(Box::new(ZipCompression));
compressor.compress_file(&data);

compressor.set_strategy(Box::new(RarCompression));
compressor.compress_file(&data);

```

Zero-cost strategy with generics

```

struct StaticCompressor<S> {
    strategy: S,
}

impl<S: CompressionStrategy> StaticCompressor<S> {
    fn new(strategy: S) -> Self {
        Self { strategy }
    }

    fn compress_file(&self, data: &[u8]) -> Vec<u8> {
        self.strategy.compress(data)
    }
}

//=====

```

```
// Compile-time strategy selection, no heap allocation
//=====
let compressor = StaticCompressor::new(ZipCompression);
compressor.compress_file(&data);
```

Functional strategy with closures

```
struct FunctionalCompressor<F>
where
    F: Fn(&[u8]) -> Vec<u8>,
{
    compress_fn: F,
}

impl<F> FunctionalCompressor<F>
where
    F: Fn(&[u8]) -> Vec<u8>,
{
    fn new(compress_fn: F) -> Self {
        Self { compress_fn }
    }

    fn compress(&self, data: &[u8]) -> Vec<u8> {
        (self.compress_fn)(data)
    }
}

//=====
// Strategy as closure
//=====
let zip_fn = |data: &[u8]| -> Vec<u8> {
    println!("ZIP compressing");
    data.to_vec()
};
let compressor = FunctionalCompressor::new(zip_fn);
```

Trade-offs: - **Trait objects:** Runtime flexibility, heap allocation, dynamic dispatch - **Generics:** Zero-cost, compile-time selection, monomorphization - **Closures:** Concise, captures environment, good for simple strategies

When to use: Multiple algorithms, runtime selection needed, testability (inject mock strategies)

Observer Pattern

Intent: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified automatically.

Motivation: GUIs, event systems, and reactive programming need to notify multiple listeners when state changes. Rust's ownership makes traditional observer patterns challenging, but channels and callbacks provide idiomatic solutions.

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

//=====
// Observer trait
//=====
trait Observer {
    fn update(&mut self, temperature: f32);
}

//=====
// Concrete observers
//=====
struct TemperatureDisplay {
    name: String,
}

impl Observer for TemperatureDisplay {
    fn update(&mut self, temperature: f32) {
        println!("{} display: {}°C", self.name, temperature);
    }
}

struct TemperatureLogger {
    log: Vec<f32>,
}

impl Observer for TemperatureLogger {
    fn update(&mut self, temperature: f32) {
        self.log.push(temperature);
        println!("Logged: {}°C (total: {} readings)", temperature, self.log.len());
    }
}

//=====
// Subject
//=====
struct WeatherStation {
    temperature: f32,
    observers: Vec<Arc<Mutex<dyn Observer + Send>>>,
}

impl WeatherStation {
    fn new() -> Self {
        Self {
            temperature: 0.0,
            observers: Vec::new(),
        }
    }

    fn attach(&mut self, observer: Arc<Mutex<dyn Observer + Send>>) {
        self.observers.push(observer);
    }
}
```

```

fn set_temperature(&mut self, temp: f32) {
    self.temperature = temp;
    self.notify();
}

fn notify(&self) {
    for observer in &self.observers {
        observer.lock().unwrap().update(self.temperature);
    }
}
}

//=====
// Usage
//=====
let mut station = WeatherStation::new();

let display = Arc::new(Mutex::new(TemperatureDisplay {
    name: "Main".to_string(),
})); 
let logger = Arc::new(Mutex::new(TemperatureLogger { log: Vec::new() })); 

station.attach(display);
station.attach(logger);

station.set_temperature(25.5);
station.set_temperature(26.0);

```

Channel-based observer (more idiomatic)

```

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

struct Event {
    temperature: f32,
}

//=====
// Publisher
//=====
struct Publisher {
    subscribers: Vec<mpsc::Sender<Event>>,
}

impl Publisher {
    fn new() -> Self {
        Self {
            subscribers: Vec::new(),
        }
    }
}

```

```

fn subscribe(&mut self) -> mpsc::Receiver<Event> {
    let (tx, rx) = mpsc::channel();
    self.subscribers.push(tx);
    rx
}

fn publish(&self, event: Event) {
    self.subscribers.retain(|tx| tx.send(event.clone()).is_ok());
}
}

#[derive(Clone)]
struct Event {
    temperature: f32,
}

//=====
// Usage
//=====

let mut publisher = Publisher::new();

let rx1 = publisher.subscribe();
let rx2 = publisher.subscribe();

thread::spawn(move || {
    for event in rx1 {
        println!("Observer 1: {}°C", event.temperature);
    }
});

thread::spawn(move || {
    for event in rx2 {
        println!("Observer 2: {}°C", event.temperature);
    }
});

publisher.publish(Event { temperature: 25.5 });

```

Trade-offs: - **Classic observer:** Familiar pattern, but requires Arc<Mutex<_>> for shared mutable state - **Channels:** More idiomatic in Rust, natural parallelism, no shared state - **Callbacks:** Simple for single-threaded scenarios

When to use: Event systems, GUIs, reactive programming, pub/sub architectures

Command Pattern

Intent: Encapsulate a request as an object, allowing you to parameterize clients with different requests, queue or log requests, and support undoable operations.

Motivation: When you need to decouple the object that invokes an operation from the one that performs it, commands encapsulate all information needed to execute an action. This enables undo/redo, macros, and request queuing.

```
//=====
// Command trait
//=====
trait Command {
    fn execute(&mut self);
    fn undo(&mut self);
}

//=====
// Receiver
//=====
struct TextEditor {
    content: String,
}

impl TextEditor {
    fn new() -> Self {
        Self {
            content: String::new(),
        }
    }

    fn write(&mut self, text: &str) {
        self.content.push_str(text);
    }

    fn delete_last(&mut self, count: usize) {
        let new_len = self.content.len().saturating_sub(count);
        self.content.truncate(new_len);
    }

    fn get_content(&self) -> &str {
        &self.content
    }
}

//=====
// Concrete commands
//=====
struct WriteCommand {
    editor: std::rc::Rc<std::cell::RefCell<TextEditor>>,
    text: String,
}

impl Command for WriteCommand {
    fn execute(&mut self) {
        self.editor.borrow_mut().write(&self.text);
    }
}
```

```

    fn undo(&mut self) {
        self.editor.borrow_mut().delete_last(self.text.len());
    }
}

struct DeleteCommand {
    editor: std::rc::Rc<std::cell::RefCell<TextEditor>>,
    deleted_text: String,
    count: usize,
}

impl Command for DeleteCommand {
    fn execute(&mut self) {
        let editor = self.editor.borrow();
        let content = editor.get_content();
        let start = content.len().saturating_sub(self.count);
        self.deleted_text = content[start..].to_string();
        drop(editor);

        self.editor.borrow_mut().delete_last(self.count);
    }

    fn undo(&mut self) {
        self.editor.borrow_mut().write(&self.deleted_text);
    }
}

//=====
// Invoker
//=====

struct CommandHistory {
    history: Vec<Box<dyn Command>>,
    current: usize,
}

impl CommandHistory {
    fn new() -> Self {
        Self {
            history: Vec::new(),
            current: 0,
        }
    }

    fn execute(&mut self, mut command: Box<dyn Command>) {
        command.execute();
        // Discard any undone commands
        self.history.truncate(self.current);
        self.history.push(command);
        self.current += 1;
    }

    fn undo(&mut self) {

```

```

        if self.current > 0 {
            self.current -= 1;
            self.history[self.current].undo();
        }
    }

    fn redo(&mut self) {
        if self.current < self.history.len() {
            self.history[self.current].execute();
            self.current += 1;
        }
    }
}

//=====
// Usage
//=====
use std::rc::Rc;
use std::cell::RefCell;

let editor = Rc::new(RefCell::new(TextEditor::new()));
let mut history = CommandHistory::new();

history.execute(Box::new(WriteCommand {
    editor: editor.clone(),
    text: "Hello ".to_string(),
})); 
history.execute(Box::new(WriteCommand {
    editor: editor.clone(),
    text: "World".to_string(),
})); 

println!("{}", editor.borrow().get_content()); // "Hello World"

history.undo(); 
println!("{}", editor.borrow().get_content()); // "Hello "

history.redo(); 
println!("{}", editor.borrow().get_content()); // "Hello World"

```

Functional command pattern

```

struct FunctionalCommand {
    execute_fn: Box<dyn FnMut()>,
    undo_fn: Box<dyn FnMut()>,
}

impl FunctionalCommand {
    fn new(execute_fn: Box<dyn FnMut()>, undo_fn: Box<dyn FnMut()>) -> Self {
        Self { execute_fn, undo_fn }
    }
}

```

```

fn execute(&mut self) {
    (self.execute_fn)();
}

fn undo(&mut self) {
    (self.undo_fn)();
}

}

```

Trade-offs: - **Pros:** Decouples invoker from receiver, enables undo/redo, command queuing, macros -
Cons: More objects, requires shared mutable state (Rc<RefCell<_>>)

When to use: Undo/redo systems, transaction systems, job queues, macro recording

Iterator Pattern

Intent: Provide a way to access elements of a collection sequentially without exposing the underlying representation.

Motivation: Rust has first-class support for the iterator pattern through the **Iterator** trait. This is the most idiomatic way to traverse collections and is deeply integrated into the language.

```

//=====
// Custom iterator
//=====

struct Fibonacci {
    current: u64,
    next: u64,
}

impl Fibonacci {
    fn new() -> Self {
        Self { current: 0, next: 1 }
    }
}

impl Iterator for Fibonacci {
    type Item = u64;

    fn next(&mut self) -> Option<Self::Item> {
        let current = self.current;
        self.current = self.next;
        self.next = current + self.next;
        Some(current)
    }
}

//=====
// Usage: Works with all iterator combinators
//=====

```

```
let fibs: Vec<u64> = Fibonacci::new().take(10).collect();
println!("{:?}", fibs); // [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Implementing IntoIterator for custom collections

```
struct MyCollection {
    items: Vec<String>,
}

impl MyCollection {
    fn new() -> Self {
        Self { items: Vec::new() }
    }

    fn add(&mut self, item: String) {
        self.items.push(item);
    }
}

//=====
// Owned iterator
//=====

impl IntoIterator for MyCollection {
    type Item = String;
    type IntoIter = std::vec::IntoIter<String>;

    fn into_iter(self) -> Self::IntoIter {
        self.items.into_iter()
    }
}

//=====
// Borrowed iterator
//=====

impl<'a> IntoIterator for &'a MyCollection {
    type Item = &'a String;
    type IntoIter = std::slice::Iter<'a, String>;

    fn into_iter(self) -> Self::IntoIter {
        self.items.iter()
    }
}

//=====
// Usage
//=====

let mut collection = MyCollection::new();
collection.add("Item 1".to_string());
collection.add("Item 2".to_string());

for item in &collection {
    println!("{}", item);
```

```

}

//=====
// collection still valid
//=====

for item in collection {
    println!("{}" , item);
}

//=====
// collection moved
//=====


```

Trade-offs: - **Pros:** Lazy evaluation, composable, integrates with language, zero-cost abstractions -

Cons: None—this is the idiomatic Rust approach

When to use: Always, for any sequential access. This is a fundamental Rust pattern.

Concurrency Patterns

Concurrency patterns address the challenges of parallel and asynchronous execution. Rust's ownership system prevents data races at compile-time, enabling fearless concurrency. These patterns leverage threads, `async/await`, and synchronization primitives.

Thread Pool Pattern

Intent: Manage a pool of worker threads to execute tasks efficiently, amortizing thread creation cost and limiting resource usage.

Motivation: Creating a thread per task is expensive. Thread pools maintain a fixed number of workers that process tasks from a queue, improving throughput and controlling concurrency.

```

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

type Job = Box;

struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Job>,
}

impl ThreadPool {
    fn new(size: usize) -> Self {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();
        let receiver = Arc::new(Mutex::new(receiver));

        let workers = (0..size)
            .map(|id| Worker::new(id, Arc::clone(&receiver)))

```

```

.collect();

    ThreadPool { workers, sender }
}

fn execute<F>(&self, job: F)
where
    F: FnOnce() + Send + 'static,
{
    self.sender.send(Box::new(job)).unwrap();
}
}

struct Worker {
    id: usize,
    thread: thread::JoinHandle<()>,
}

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Self {
        let thread = thread::spawn(move || loop {
            let job = receiver.lock().unwrap().recv();

            match job {
                Ok(job) => {
                    println!("Worker {} executing job", id);
                    job();
                }
                Err(_) => {
                    println!("Worker {} shutting down", id);
                    break;
                }
            }
        });
        Worker { id, thread }
    }
}

//=====
// Usage
//=====
let pool = ThreadPool::new(4);

for i in 0..10 {
    pool.execute(move || {
        println!("Task {} running", i);
        thread::sleep(std::time::Duration::from_millis(100));
    });
}

```

Using rayon for data parallelism

```

use rayon::prelude::*;

//=====
// Parallel iterator (much simpler than manual thread pool)
//=====

let numbers: Vec<i32> = (0..1000).collect();
let sum: i32 = numbers.par_iter().map(|&x| x * 2).sum();

```

Trade-offs: - **Manual thread pool:** Full control, custom scheduling, but complex implementation - **Rayon:** Simple, efficient, but less control over task scheduling - **Tokio runtime:** For async I/O tasks, not CPU-bound work

When to use: CPU-bound parallel tasks, limiting concurrency, long-running workers

Producer-Consumer Pattern

Intent: Decouple producers that generate data from consumers that process it, using a queue as a buffer.

Motivation: When production and consumption happen at different rates, a queue prevents blocking and enables concurrent processing. Rust's channels provide a natural implementation.

```

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

//=====
// Producer
//=====

fn producer(tx: mpsc::Sender<i32>) {
    for i in 0..10 {
        println!("Producing {}", i);
        tx.send(i).unwrap();
        thread::sleep(Duration::from_millis(100));
    }
    // Channel closes when tx is dropped
}

//=====
// Consumer
//=====

fn consumer(rx: mpsc::Receiver<i32>) {
    for item in rx {
        println!(" Consuming {}", item);
        thread::sleep(Duration::from_millis(200)); // Slower than producer
    }
}

//=====
// Usage: Single producer, single consumer

```

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

thread::spawn(move || producer(tx));
thread::spawn(move || consumer(rx));

thread::sleep(Duration::from_secs(3));
```

Multiple producers, single consumer

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

for id in 0..3 {
    let tx_clone = tx.clone();
    thread::spawn(move || {
        for i in 0..5 {
            tx_clone.send((id, i)).unwrap();
            thread::sleep(Duration::from_millis(50));
        }
    });
}
drop(tx); // Drop original to allow channel to close

thread::spawn(move || {
    for (id, item) in rx {
        println!("Producer {} sent {}", id, item);
    }
});
```

Bounded channel for backpressure

```
use std::sync::mpsc::sync_channel;

let (tx, rx) = sync_channel(3); // Buffer size of 3

//=====
// Producer blocks if queue is full
//=====

thread::spawn(move || {
    for i in 0..10 {
        println!("Sending {}", i);
        tx.send(i).unwrap(); // Blocks if buffer full
    }
});

thread::spawn(move || {
    thread::sleep(Duration::from_secs(1)); // Delay consumer
    for item in rx {
        println!("Received {}", item);
    }
});
```

```
});
```

Trade-offs: - **Unbounded channel:** No backpressure, can cause unbounded memory growth -
Bounded channel: Backpressure control, but producers may block - **Crossbeam:** More efficient channels, select! support

When to use: Pipeline architectures, async work queues, rate limiting, buffering

Fork-Join Pattern

Intent: Split a task into subtasks that can run in parallel, then join the results.

Motivation: Divide-and-conquer algorithms benefit from parallel execution. Fork-join splits work across threads and combines results.

```
use std::thread;

fn parallel_sum(data: &[i32]) -> i32 {
    const THRESHOLD: usize = 100;

    if data.len() <= THRESHOLD {
        // Base case: sequential sum
        data.iter().sum()
    } else {
        // Fork: split into two halves
        let mid = data.len() / 2;
        let (left, right) = data.split_at(mid);

        let left_data = left.to_vec();
        let handle = thread::spawn(move || parallel_sum(&left_data));

        let right_sum = parallel_sum(right);
        let left_sum = handle.join().unwrap();

        // Join: combine results
        left_sum + right_sum
    }
}

//=====
// Usage
//=====
let data: Vec<i32> = (1..=1000).collect();
let total = parallel_sum(&data);
println!("Sum: {}", total);
```

Using rayon for automatic fork-join

```

use rayon::prelude::*;

fn rayon_sum(data: &[i32]) -> i32 {
    data.par_iter().sum() // Automatically parallelizes
}

//=====
// Parallel recursion with rayon
//=====

fn quicksort<T: Ord + Send>(mut data: Vec<T>) -> Vec<T> {
    if data.len() <= 1 {
        return data;
    }

    let pivot = data.remove(0);
    let (mut less, mut greater): (Vec<_>, Vec<_>) = data
        .into_par_iter() // Parallel partition
        .partition(|x| x < &pivot);

    // Parallel recursive calls
    let (sorted_less, sorted_greater) = rayon::join(
        || quicksort(less),
        || quicksort(greater),
    );

    let mut result = sorted_less;
    result.push(pivot);
    result.extend(sorted_greater);
    result
}

```

Trade-offs: - **Manual fork-join:** Full control, but must manage thread creation - **Rayon:** Automatic work-stealing, simpler, efficient

When to use: Divide-and-conquer algorithms, parallel recursion, data parallelism

Actor Pattern

Intent: Encapsulate state and behavior in isolated actors that communicate only through message passing.

Motivation: Shared mutable state is the root of concurrency bugs. Actors eliminate shared state by giving each actor exclusive ownership of its data, communicating only via messages.

```

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

//=====
// Message types
//=====

enum AccountMessage {

```

```

Deposit(u64),
Withdraw(u64),
GetBalance(mpsc::Sender<u64>),
Shutdown,
}

//=====
// Actor
//=====
struct BankAccount {
    balance: u64,
    receiver: mpsc::Receiver<AccountMessage>,
}

impl BankAccount {
    fn new(initial: u64) -> (Self, mpsc::Sender<AccountMessage>) {
        let (tx, rx) = mpsc::channel();
        let account = BankAccount {
            balance: initial,
            receiver: rx,
        };
        (account, tx)
    }

    fn run(mut self) {
        thread::spawn(move || {
            for msg in self.receiver {
                match msg {
                    AccountMessage::Deposit(amount) => {
                        self.balance += amount;
                        println!("Deposited {}. Balance: {}", amount, self.balance);
                    }
                    AccountMessage::Withdraw(amount) => {
                        if self.balance >= amount {
                            self.balance -= amount;
                            println!("Withdrew {}. Balance: {}", amount, self.balance);
                        } else {
                            println!("Insufficient funds");
                        }
                    }
                    AccountMessage::GetBalance(reply) => {
                        reply.send(self.balance).unwrap();
                    }
                    AccountMessage::Shutdown => {
                        println!("Shutting down account");
                        break;
                    }
                }
            }
        });
    }
}

```

```

//=====
// Usage
//=====
let (account, handle) = BankAccount::new(100);
account.run();

handle.send(AccountMessage::Deposit(50)).unwrap();
handle.send(AccountMessage::Withdraw(30)).unwrap();

let (tx, rx) = mpsc::channel();
handle.send(AccountMessage::GetBalance(tx)).unwrap();
let balance = rx.recv().unwrap();
println!("Final balance: {}", balance);

handle.send(AccountMessage::Shutdown).unwrap();

```

Using Actix framework

```

//=====
// With actix-rt (external crate)
//=====
// Much more ergonomic for complex actor systems
/*
use actix::prelude::*;

struct BankAccount {
    balance: u64,
}

impl Actor for BankAccount {
    type Context = Context<Self>;
}

struct Deposit(u64);
impl Message for Deposit {
    type Result = ();
}

impl Handler<Deposit> for BankAccount {
    type Result = ();

    fn handle(&mut self, msg: Deposit, _ctx: &mut Context<Self>) {
        self.balance += msg.0;
    }
}
*/

```

Trade-offs: - **Pros:** No shared state, natural concurrency model, isolated failures - **Cons:** Message passing overhead, complexity for simple cases - **Frameworks:** Actix provides full-featured actor system

When to use: Stateful services, concurrent servers, distributed systems, isolation requirements

Async/Await Pattern

Intent: Write asynchronous code that looks like synchronous code, improving readability while maintaining non-blocking execution.

Motivation: Callback-based async code becomes nested and hard to follow. Async/await provides sequential syntax for asynchronous operations, compiling to efficient state machines.

```
use tokio;
use std::time::Duration;

async fn fetch_user(id: u64) -> String {
    // Simulated async operation
    tokio::time::sleep(Duration::from_millis(100)).await;
    format!("User {}", id)
}

async fn fetch_posts(user: &str) -> Vec<String> {
    tokio::time::sleep(Duration::from_millis(100)).await;
    vec![
        format!("{}'s post 1", user),
        format!("{}'s post 2", user),
    ]
}

async fn display_user_data(id: u64) {
    // Sequential async operations
    let user = fetch_user(id).await;
    println!("Fetched: {}", user);

    let posts = fetch_posts(&user).await;
    for post in posts {
        println!(" - {}", post);
    }
}

#[tokio::main]
async fn main() {
    display_user_data(1).await;
}
```

Parallel async operations with join!

```
use tokio;

async fn parallel_fetch() {
    // Wait for all to complete
    let (user1, user2, user3) = tokio::join!(
        fetch_user(1),
        fetch_user(2),
```

```

        fetch_user(3),
    );

    println!("{} , {} , {}", user1, user2, user3);
}

```

Select pattern: First to complete

```

async fn timeout_example() {
    let fetch = fetch_user(1);
    let timeout = tokio::time::sleep(Duration::from_millis(50));

    tokio::select! {
        user = fetch => println!("Got user: {}", user),
        _ = timeout => println!("Timeout!"),
    }
}

```

Trade-offs: - **Pros:** Readable async code, efficient (single-threaded event loop), scales to many connections - **Cons:** Runtime dependency (tokio/async-std), learning curve, colored functions - **vs Threads:** Async for I/O-bound, threads for CPU-bound

When to use: Web servers, network services, I/O-heavy applications, high concurrency with low CPU usage

Summary

Design patterns in Rust take unique forms due to the language's ownership model, type system, and zero-cost abstractions. Many patterns that require runtime polymorphism in object-oriented languages can be implemented at compile-time in Rust through generics and traits, eliminating overhead.

Key Takeaways

Creational patterns (Builder, Factory, Singleton, Prototype) benefit from Rust's type system: - Builders enable fluent APIs; typestate pattern enforces correctness at compile-time - Factories use traits or enums; enums provide zero-cost closed variants - Singletons use `OnceLock`; prefer dependency injection when possible - Prototypes leverage `Clone` trait, a first-class Rust concept

Structural patterns (Adapter, Decorator, Facade, Newtype) compose types efficiently: - Adapters bridge incompatible interfaces; generics eliminate runtime cost - Decorators add behavior; trait objects enable runtime composition, generics enable compile-time - Facades simplify complex subsystems - Newtypes provide zero-cost type safety and orphan rule workarounds

Behavioral patterns (Strategy, Observer, Command, Iterator) define interactions: - Strategies use traits; closures provide functional alternative - Observers use channels rather than callbacks; more idiomatic and thread-safe - Commands enable undo/redo; require shared mutable state (`Rc<RefCell<_>>`) - Iterators are first-class in Rust; the Iterator trait is ubiquitous

Concurrency patterns (Thread Pool, Producer-Consumer, Fork-Join, Actor, Async/Await) leverage Rust's fearless concurrency:

- Thread pools manage worker threads; rayon simplifies data parallelism
- Producer-consumer uses channels naturally; bounded channels provide backpressure
- Fork-join parallelizes divide-and-conquer; rayon automates work-stealing
- Actors eliminate shared state through message passing
- Async/await provides readable asynchronous code for I/O-bound tasks

Choosing the Right Pattern

When facing a design decision:

1. **Prefer zero-cost abstractions:** Use generics over trait objects when types are known at compile-time
2. **Embrace ownership:** Design with moves, borrows, and lifetimes rather than fighting them
3. **Use standard traits:** Implement `Iterator`, `From`, `Display` rather than custom abstractions
4. **Leverage the type system:** Encode invariants in types (newtype, typestate) rather than runtime checks
5. **Consider the async ecosystem:** For I/O-heavy code, async/await often beats threads
6. **Don't over-pattern:** Simple code beats clever patterns; only apply patterns when they solve real problems

Rust's patterns emphasize compile-time guarantees, zero-cost abstractions, and fearless concurrency. Master these patterns to write code that is both safe and performant, idiomatic and maintainable.

Appendix C: Anti-Patterns

Common Pitfalls:

- [Excessive Cloning](#)
- [Overusing Rc/Arc Without Need](#)
- [Ignoring Iterator Combinators](#)
- [Deref Coercion Abuse](#)
- [String vs &str Confusion](#)

Performance Anti-Patterns:

- [Collecting Iterators Unnecessarily](#)
- [Vec When \[\] Suffices](#)
- [HashMap for Small Key Sets](#)
- [Premature String Allocation](#)
- [Boxed Trait Objects Everywhere](#)

Safety Anti-Patterns:

- [Unsafe for Convenience](#)
- [Unwrap\(\) in Production Code](#)
- [RefCell/Mutex Without Consideration](#)
- [Ignoring Send/Sync Implications](#)

API Design Mistakes:

- [Stringly-Typed APIs](#)
- [Boolean Parameter Trap](#)
- [Leaky Abstractions](#)
- [Returning Owned When Borrowed Suffices](#)
- [Overengineered Generic APIs](#)

Overview

Anti-patterns are common solutions to recurring problems that initially seem reasonable but ultimately create more issues than they solve. Unlike design patterns, which represent best practices, anti-patterns represent pitfalls—seductive shortcuts that lead to bugs, performance degradation, or unmaintainable code.

In Rust, anti-patterns often emerge when developers apply patterns from other languages without adapting to Rust’s ownership model, or when they fight the compiler rather than understanding what it’s trying to prevent. The borrow checker is not your enemy—it’s preventing real-time bugs. When code feels like a battle against the type system, you’re usually approaching the problem incorrectly.

This catalog identifies common anti-patterns in four categories:

Common pitfalls are mistakes developers make when learning Rust, often from misunderstanding ownership, borrowing, or lifetime rules. These lead to unnecessary clones, reference counting where simple borrowing would suffice, or awkward code structure.

Performance anti-patterns sacrifice efficiency for convenience without understanding the cost. These include excessive allocations, missed optimization opportunities, and patterns that prevent the compiler from generating optimal code.

Safety anti-patterns undermine Rust’s guarantees, often through misuse of `unsafe`, inappropriate uses of interior mutability, or patterns that make code fragile and error-prone.

API design mistakes create poor interfaces—difficult to use correctly, easy to misuse, or inconsistent with Rust ecosystem conventions. These frustrate users and lead to adoption problems.

Each anti-pattern includes: - **The pattern:** What it looks like in code - **Why it’s problematic:** The issues it causes - **The solution:** The correct approach - **Real-world impact:** Consequences in production code

Learning to recognize and avoid these anti-patterns is as important as knowing design patterns. They represent the accumulated wisdom of the Rust community—lessons learned through bugs, performance issues, and maintenance headaches.

Common Pitfalls

These anti-patterns emerge from misunderstanding Rust’s fundamental concepts. They’re especially common among developers transitioning from garbage-collected languages or C++.

Anti-Pattern: Excessive Cloning

The Pattern: Cloning data unnecessarily to satisfy the borrow checker rather than understanding borrowing rules.

```
//=====
// ✗ ANTI-PATTERN: Clone to avoid borrow checker errors
//=====

fn process_data(data: Vec<String>) {
    let copy1 = data.clone(); // Unnecessary
    print_data(copy1);

    let copy2 = data.clone(); // Unnecessary
    transform_data(copy2);

    let copy3 = data.clone(); // Unnecessary
    save_data(copy3);
}

fn print_data(data: Vec<String>) {
    for item in &data {
        println!("{}", item);
    }
}

fn transform_data(data: Vec<String>) -> Vec<String> {
    data.iter().map(|s| s.to_uppercase()).collect()
}

fn save_data(data: Vec<String>) {
    // Save to database
}
```

Why It's Problematic: - Performance degradation: Each clone allocates and copies all data - Memory waste: Multiple copies of the same data - Indicates misunderstanding of ownership and borrowing - In large datasets, this can cause significant slowdowns

The Solution: Use references for read-only access, mutable references for modifications, or move ownership when appropriate.

```
//=====
// ✓ CORRECT: Use borrowing appropriately
//=====

fn process_data(data: Vec<String>) {
    // Borrow for read-only access
    print_data(&data);

    // Clone only when you need to modify and keep original
    let transformed = transform_data(&data);

    // Move ownership when done with original
}
```

```

        save_data(data);
    }

fn print_data(data: &[String]) {
    for item in data {
        println!("{}: {}", item);
    }
}

fn transform_data(data: &[String]) -> Vec<String> {
    data.iter().map(|s| s.to_uppercase()).collect()
}

fn save_data(data: Vec<String>) {
    // Takes ownership, no clone needed
}

```

Real-World Impact: A web service cloning request data at every processing step saw 3x memory usage and 40% latency increase. Switching to borrowing reduced memory by 66% and improved response times significantly.

Anti-Pattern: Overusing Rc/Arc Without Need

The Pattern: Using reference counting for shared ownership when simple borrowing or restructuring would work.

```

use std::rc::Rc;

//=====
// ✗ ANTI-PATTERN: Rc when not needed
//=====

struct DataProcessor {
    config: Rc<Config>,
    logger: Rc<Logger>,
    cache: Rc<Cache>,
}

impl DataProcessor {
    fn new(config: Config, logger: Logger, cache: Cache) -> Self {
        Self {
            config: Rc::new(config),
            logger: Rc::new(logger),
            cache: Rc::new(cache),
        }
    }

    fn process(&self, data: &str) {
        self.logger.log("Processing...");
        // Uses Rc for simple single-owner scenario
    }
}

```

```
}
```

Why It's Problematic: - Runtime overhead: Reference counting adds CPU cost - Heap allocation: Rc requires heap allocation - Complexity: Rc<RefCell> for mutation is unnecessarily complex - Hides ownership structure: Makes data flow unclear - Thread safety issues: Rc isn't Send/Sync

The Solution: Use references with explicit lifetimes, restructure ownership, or only use Rc/Arc when genuinely needed for shared ownership.

```
//=====
// ✅ CORRECT: Use references with lifetimes
//=====

struct DataProcessor<'a> {
    config: &'a Config,
    logger: &'a Logger,
    cache: &'a Cache,
}

impl<'a> DataProcessor<'a> {
    fn new(config: &'a Config, logger: &'a Logger, cache: &'a Cache) -> Self {
        Self { config, logger, cache }
    }

    fn process(&self, data: &str) {
        self.logger.log("Processing...");
    }
}

//=====
// Or restructure to owned data when sharing isn't needed
//=====

struct DataProcessor {
    config: Config, // Small configs can be copied/cloned
}

//=====
// Only use Rc/Arc when truly sharing across owners
//=====

use std::sync::Arc;
use std::thread;

fn multiple_threads_need_shared_data() {
    let config = Arc::new(Config::load());

    let config1 = Arc::clone(&config);
    let handle1 = thread::spawn(move || process_with_config(&config1));

    let config2 = Arc::clone(&config);
    let handle2 = thread::spawn(move || process_with_config(&config2));
}
```

```
// Genuine shared ownership across threads
}
```

Real-World Impact: A server using Rc throughout for “convenience” had 15% CPU overhead from reference counting. Refactoring to use references where possible eliminated the overhead.

Anti-Pattern: Ignoring Iterator Combinators

The Pattern: Using manual loops and mutable accumulators instead of iterator methods.

```
=====
// ✗ ANTI-PATTERN: Manual loops instead of iterators
=====

fn process_numbers(numbers: &[i32]) -> Vec<i32> {
    let mut result = Vec::new();
    for &num in numbers {
        if num % 2 == 0 {
            result.push(num * 2);
        }
    }
    result
}

fn find_first_large(numbers: &[i32]) -> Option<i32> {
    for &num in numbers {
        if num > 100 {
            return Some(num);
        }
    }
    None
}

fn sum_squares(numbers: &[i32]) -> i32 {
    let mut sum = 0;
    for &num in numbers {
        sum += num * num;
    }
    sum
}
```

Why It's Problematic: - More verbose and harder to read - Mutable state management is error-prone
- Misses optimization opportunities (compiler can optimize iterator chains better) - Doesn't compose well - Harder to parallelize (rayon works with iterators)

The Solution: Use iterator combinators for declarative, composable data processing.

```
=====
// ✅ CORRECT: Use iterator combinators
=====

fn process_numbers(numbers: &[i32]) -> Vec<i32> {
```

```

numbers.iter()
    .filter(|&&num| num % 2 == 0)
    .map(|&num| num * 2)
    .collect()
}

fn find_first_large(numbers: &[i32]) -> Option<i32> {
    numbers.iter()
        .find(|&&num| num > 100)
        .copied()
}

fn sum_squares(numbers: &[i32]) -> i32 {
    numbers.iter()
        .map(|&num| num * num)
        .sum()
}

//=====
// Easy to make parallel with rayon
//=====

use rayon::prelude::*;

fn parallel_sum_squares(numbers: &[i32]) -> i32 {
    numbers.par_iter() // Just change iter() to par_iter()
        .map(|&num| num * num)
        .sum()
}

```

Real-World Impact: Iterator chains often compile to the same or better assembly than manual loops, while being more maintainable. A data processing pipeline converted to iterators saw identical performance but 40% less code.

Anti-Pattern: Deref Coercion Abuse

The Pattern: Relying on Deref coercion for API design, making code implicit and confusing.

```

use std::ops::Deref;

//=====
// ❌ ANTI-PATTERN: Abusing Deref for inheritance-like behavior
//=====

struct Employee {
    name: String,
    id: u32,
}

struct Manager {
    employee: Employee,
    team_size: usize,
}

```

```

impl Deref for Manager {
    type Target = Employee;

    fn deref(&self) -> &Self::Target {
        &self.employee
    }
}

//=====
// Now Manager "inherits" Employee methods
//=====

fn print_employee_info(emp: &Employee) {
    println!("{}: {}", emp.id, emp.name);
}

let manager = Manager {
    employee: Employee { name: "Alice".to_string(), id: 1 },
    team_size: 5,
};

//=====
// Works due to Deref, but confusing
//=====

print_employee_info(&manager);

```

Why It's Problematic: - Violates principle of least surprise: implicit conversions hide intent - Not true inheritance: doesn't work with trait objects - Maintenance issues: changing Deref breaks code in non-obvious ways - Against Rust guidelines: Deref is for smart pointers, not emulating inheritance

The Solution: Use explicit methods or trait implementations for delegation.

```

//=====
// ✅ CORRECT: Explicit delegation
//=====

struct Manager {
    employee: Employee,
    team_size: usize,
}

impl Manager {
    fn employee(&self) -> &Employee {
        &self.employee
    }

    // Or delegate specific methods explicitly
    fn name(&self) -> &str {
        &self.employee.name
    }

    fn id(&self) -> u32 {
        self.employee.id
    }
}

```

```

    }

//=====
// Explicit conversion
//=====
fn print_employee_info(emp: &Employee) {
    println!("{}: {}", emp.id, emp.name);
}

let manager = Manager {
    employee: Employee { name: "Alice".to_string(), id: 1 },
    team_size: 5,
};

//=====
// Clear and explicit
//=====
print_employee_info(manager.employee());

```

Real-World Impact: A library using Deref for pseudo-inheritance confused users and broke when internal structure changed. Switching to explicit methods improved API clarity.

Anti-Pattern: String vs &str Confusion

The Pattern: Unnecessary string allocations and confusing conversions between String and &str.

```

//=====
// ✗ ANTI-PATTERN: Unnecessary allocations
//=====

fn greet(name: String) -> String {
    format!("Hello, {}", name)
}

fn process_names(names: Vec<&str>) {
    for name in names {
        let owned = name.to_string(); // Unnecessary allocation
        greet(owned);
    }
}

//=====
// Forces callers to own Strings
//=====
let name = "Alice".to_string(); // Allocation just to call function
greet(name);

```

Why It's Problematic: - Forces unnecessary allocations on callers - Less flexible API (can't use string literals without converting) - Performance cost from heap allocations - Doesn't follow Rust idioms

The Solution: Accept &str in function parameters, return String when ownership is transferred.

```

//=====
// ✅ CORRECT: Accept &str, return String when needed
//=====

fn greet(name: &str) -> String {
    format!("Hello, {}", name)
}

fn process_names(names: &[&str]) {
    for &name in names {
        let greeting = greet(name); // No unnecessary allocation
        println!("{}", greeting);
    }
}

//=====
// Flexible: works with literals and owned strings
//=====

greet("Alice");           // No allocation needed
let owned = String::from("Bob");
greet(&owned);           // Also works

//=====
// For more flexibility, use generic trait bounds
//=====

fn greet_generic<S: AsRef<str>>(name: S) -> String {
    format!("Hello, {}", name.as_ref())
}

//=====
// Works with &str, String, Cow<str>, etc.
//=====

greet_generic("Alice");
greet_generic(String::from("Bob"));

```

Real-World Impact: A web service accepting `String` parameters forced clients to allocate for every request. Changing to `&str` reduced allocations by 80% in typical workloads.

Performance Anti-Patterns

These patterns sacrifice performance without good reason, often from not understanding the cost model or missing optimization opportunities.

Anti-Pattern: Collecting Iterators Unnecessarily

The Pattern: Calling `.collect()` in the middle of iterator chains when the final consumer can work with iterators.

```

//=====
// ❌ ANTI-PATTERN: Unnecessary intermediate collections

```

```
//=====
fn process_data(numbers: &[i32]) -> i32 {
    let evens: Vec<i32> = numbers.iter()
        .filter(|&&x| x % 2 == 0)
        .copied()
        .collect(); // Unnecessary allocation

    let doubled: Vec<i32> = evens.iter()
        .map(|&x| x * 2)
        .collect(); // Another unnecessary allocation

    doubled.iter().sum()
}
```

Why It's Problematic: - Multiple heap allocations - Cache unfriendly: data copied multiple times - Breaks iterator fusion: compiler can't optimize across collect boundaries - Memory overhead from intermediate vectors

The Solution: Chain iterators without intermediate collections.

```
//=====
// ✅ CORRECT: Single iterator chain
//=====

fn process_data(numbers: &[i32]) -> i32 {
    numbers.iter()
        .filter(|&&x| x % 2 == 0)
        .map(|&x| x * 2)
        .sum()
    // Zero allocations, single pass through data
}

//=====
// Only collect when you need to reuse the result
//=====

fn process_data_reusable(numbers: &[i32]) -> Vec<i32> {
    numbers.iter()
        .filter(|&&x| x % 2 == 0)
        .map(|&x| x * 2)
        .collect() // Now justified: we return the collection
}
```

Benchmarking shows: Single iterator chain can be 10-100x faster than multiple collect calls for large datasets.

Anti-Pattern: Vec When rray Suffices

The Pattern: Using heap-allocated `Vec<T>` for fixed-size, small collections when stack-allocated arrays would work.

```

//=====
// ✗ ANTI-PATTERN: Heap allocation for fixed-size data
//=====

fn get_rgb_channels(pixel: u32) -> Vec<u8> {
    vec![
        ((pixel >> 16) & 0xFF) as u8,
        ((pixel >> 8) & 0xFF) as u8,
        (pixel & 0xFF) as u8,
    ]
}

fn multiply_3x3(a: Vec<Vec<f64>>, b: Vec<Vec<f64>>) -> Vec<Vec<f64>> {
    // Matrix multiplication with heap-allocated matrices
    // Multiple allocations for 9 numbers!
    unimplemented!()
}

```

Why It's Problematic: - Heap allocation overhead for small data - Pointer indirection reduces cache locality - Runtime size checks instead of compile-time guarantees - Can't take advantage of SIMD optimizations

The Solution: Use arrays for fixed-size data.

```

//=====
// ✓ CORRECT: Stack-allocated arrays
//=====

fn get_rgb_channels(pixel: u32) -> [u8; 3] {
    [
        ((pixel >> 16) & 0xFF) as u8,
        ((pixel >> 8) & 0xFF) as u8,
        (pixel & 0xFF) as u8,
    ]
}

fn multiply_3x3(a: [[f64; 3]; 3], b: [[f64; 3]; 3]) -> [[f64; 3]; 3] {
    let mut result = [[0.0; 3]; 3];
    for i in 0..3 {
        for j in 0..3 {
            for k in 0..3 {
                result[i][j] += a[i][k] * b[k][j];
            }
        }
    }
    result
    // All on stack, no allocations
}

//=====
// Use Vec only when size is truly dynamic
//=====

fn get_pixels(count: usize) -> Vec<[u8; 3]> {

```

```
    vec![0, 0, 0]; count]
}
```

Real-World Impact: Graphics code using `Vec<u8>` for RGB triplets spent 40% of time in allocator. Switching to `[u8; 3]` eliminated allocations and doubled throughput.

Anti-Pattern: HashMap for Small Key Sets

The Pattern: Using `HashMap` for collections with few items when linear search would be faster.

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

//=====
// ✗ ANTI-PATTERN: HashMap for 3 items
//=====

fn get_status_code(status: &str) -> u16 {
    let mut codes = HashMap::new();
    codes.insert("ok", 200);
    codes.insert("not_found", 404);
    codes.insert("error", 500);

    *codes.get(status).unwrap_or(&500)
}

//=====
// Recreating HashMap on every call!
//=====
```

Why It's Problematic: - Hash computation overhead for small collections - `HashMap` allocation and initialization cost - Linear search is faster for <10 items - Poor cache locality from hashing

The Solution: Use arrays or match statements for small, known key sets.

```
//=====
// ✓ CORRECT: Match for small known sets
//=====

fn get_status_code(status: &str) -> u16 {
    match status {
        "ok" => 200,
        "not_found" => 404,
        "error" => 500,
        _ => 500,
    }
    // Compiles to jump table or if-chain, no allocation
}

//=====
// Or array of tuples for linear search
//=====

fn get_status_code_array(status: &str) -> u16 {
```

```

const CODES: &[(&str, u16)] = &[
    ("ok", 200),
    ("not_found", 404),
    ("error", 500),
];

CODES.iter()
    .find(|(key, _)| *key == status)
    .map(|(_, code)| *code)
    .unwrap_or(500)
}

//=====
// Use HashMap only for larger, dynamic collections
//=====

use std::collections::HashMap;
use std::sync::LazyLock;

static LARGE_CODES: LazyLock<HashMap<&'static str, u16>> = LazyLock::new(|| {
    let mut map = HashMap::new();
    // Populate with many items
    for i in 0..1000 {
        // ...
    }
    map
});

```

Benchmarking: For 3-5 items, match is 10x faster than HashMap. HashMap becomes faster around 10-20 items.

Anti-Pattern: Premature String Allocation

The Pattern: Converting to String early when working with string data, causing unnecessary allocations.

```

//=====
// ❌ ANTI-PATTERN: Allocate early, use late
//=====

fn process_log_line(line: &str) -> Option<String> {
    let owned = line.to_string(); // Allocate immediately

    if !owned.starts_with("ERROR") {
        return None; // Wasted allocation
    }

    Some(owned.to_uppercase()) // Another allocation
}

fn extract_field(data: &str, field: &str) -> String {
    let owned = data.to_string(); // Unnecessary
    owned.split(',')
}

```

```

    .find(|s| s.starts_with(field))
    .unwrap_or("")
    .to_string() // Only this allocation needed
}

```

Why It's Problematic: - Allocates even when not needed (early returns) - Multiple allocations when one suffices - Doesn't leverage string slice efficiency

The Solution: Work with &str as long as possible, allocate only when necessary.

```

//=====
// ✅ CORRECT: Delay allocation
//=====

fn process_log_line(line: &str) -> Option<String> {
    if !line.starts_with("ERROR") {
        return None; // No allocation for filtered lines
    }

    Some(line.to_uppercase()) // Single allocation only when needed
}

fn extract_field<'a>(data: &'a str, field: &str) -> &'a str {
    data.split(',')
        .find(|s| s.starts_with(field))
        .unwrap_or("")
    // No allocations at all, returns slice of input
}

//=====
// If ownership needed, use Cow for conditional allocation
//=====

use std::borrow::Cow;

fn normalize<'a>(s: &'a str) -> Cow<'a, str> {
    if s.chars().any(|c| c.is_uppercase()) {
        Cow::Owned(s.to_lowercase()) // Allocate only if needed
    } else {
        Cow::Borrowed(s) // Zero-cost
    }
}

```

Real-World Impact: Log processing service allocating Strings for every line processed 10M strings/sec. 90% were filtered. Using &str until needed reduced allocations by 90% and improved throughput 3x.

Anti-Pattern: Boxed Trait Objects Everywhere

The Pattern: Using `Box<dyn Trait>` when static dispatch with generics would work.

```

//=====
// ✗ ANTI-PATTERN: Dynamic dispatch when static suffices
//=====

trait Processor {
    fn process(&self, data: &str) -> String;
}

fn pipeline(processors: Vec<Box<dyn Processor>>, data: &str) -> String {
    let mut result = data.to_string();
    for processor in processors {
        result = processor.process(&result); // Virtual call overhead
    }
    result
}

```

Why It's Problematic: - Heap allocation for each trait object - Virtual dispatch prevents inlining - No specialization possible - Dynamic dispatch has 2-10x overhead vs static

The Solution: Use generics for static dispatch when types are known at compile-time.

```

//=====
// ✓ CORRECT: Static dispatch with generics
//=====

fn pipeline<P1, P2, P3>(p1: P1, p2: P2, p3: P3, data: &str) -> String
where
    P1: Processor,
    P2: Processor,
    P3: Processor,
{
    let result = p1.process(data);
    let result = p2.process(&result);
    p3.process(&result)
    // All calls inlined, no heap allocations
}

//=====
// Or use impl Trait for flexibility
//=====

fn process_twice(data: &str, processor: impl Processor) -> String {
    let once = processor.process(data);
    processor.process(&once)
}

//=====
// Only use dyn when you truly need runtime polymorphism
//=====

fn dynamic_pipeline(processors: Vec<Box<dyn Processor>>, data: &str) -> String {
    // Justified: processors unknown at compile time
    let mut result = data.to_string();
    for processor in processors {
        result = processor.process(&result);
    }
}

```

```
    result  
}
```

Benchmarking: Static dispatch via generics can be 5-10x faster than dyn trait objects for simple operations due to inlining and specialization.

Safety Anti-Patterns

These patterns undermine Rust's safety guarantees, creating opportunities for bugs, undefined behavior, or security vulnerabilities.

Anti-Pattern: Unsafe for Convenience

The Pattern: Using `unsafe` to bypass borrow checker restrictions without genuine need or proper justification.

```
//=====  
// ✗ ANTI-PATTERN: Unsafe to "fix" borrow checker errors  
//=====  
  
struct Cache {  
    data: Vec<String>,  
}  
  
impl Cache {  
    fn get_mut_two(&mut self, i: usize, j: usize) -> (&mut String, &mut String) {  
        // "I know what I'm doing" famous last words  
        unsafe {  
            let ptr = self.data.as_mut_ptr();  
            (&mut *ptr.add(i), &mut *ptr.add(j))  
        }  
        // What if i == j? Undefined behavior!  
    }  
}
```

Why It's Problematic: - Creates undefined behavior (aliased mutable references)
- No bounds checking (can access out of bounds)
- Defeats Rust's safety guarantees
- Hard to audit and maintain - Usually indicates misunderstanding of safe alternatives

The Solution: Use safe alternatives or properly validate unsafe code.

```
//=====  
// ✓ CORRECT: Safe solution  
//=====  
  
impl Cache {  
    fn get_mut_two(&mut self, i: usize, j: usize) -> Option<(&mut String, &mut String)> {  
        if i == j {  
            return None; // Can't return two mutable refs to same element  
        }  
    }  
}
```

```

// Safe split_at_mut
if i < j {
    let (left, right) = self.data.split_at_mut(j);
    Some((&mut left[i], &mut right[0]))
} else {
    let (left, right) = self.data.split_at_mut(i);
    Some((&mut right[0], &mut left[j]))
}
}

//=====
// Or use indexing if you're sure indices are valid
//=====

impl Cache {
    fn get_mut_two_unchecked(&mut self, i: usize, j: usize) -> (&mut String, &mut String) {
        assert!(i != j);
        assert!(i < self.data.len());
        assert!(j < self.data.len());

        // Still use safe split_at_mut
        if i < j {
            let (left, right) = self.data.split_at_mut(j);
            (&mut left[i], &mut right[0])
        } else {
            let (left, right) = self.data.split_at_mut(i);
            (&mut right[0], &mut left[j])
        }
    }
}

```

Real-World Impact: A library using `unsafe` for “convenience” had a bug where indices could be equal, causing memory corruption. The safe solution using `split_at_mut` would have prevented this.

Anti-Pattern: `unwrap()` in Production Code

The Pattern: Using `.unwrap()` or `.expect()` liberally without proper error handling.

```

//=====
// ✗ ANTI-PATTERN: Unwrap everywhere
//=====

fn load_config(path: &str) -> Config {
    let contents = std::fs::read_to_string(path).unwrap(); // Panics if file missing
    let config: Config = serde_json::from_str(&contents).unwrap(); // Panics if invalid JSON
    config
}

fn get_user_age(users: &HashMap<String, User>, id: &str) -> u32 {
    users.get(id).unwrap().age // Panics if user not found
}

```

```

fn divide(a: i32, b: i32) -> i32 {
    a.checked_div(b).unwrap() // Panics on division by zero
}

```

Why It's Problematic: - Crashes on unexpected input - Poor user experience (panic messages are cryptic) - No recovery opportunity - Makes code fragile - Acceptable in examples, not production

The Solution: Handle errors properly with Result/Option or document why unwrap is safe.

```

//=====
// ✅ CORRECT: Proper error handling
//=====

use std::io;
use serde_json;

#[derive(Debug)]
enum ConfigError {
    Io(io::Error),
    Parse(serde_json::Error),
}

impl From<io::Error> for ConfigError {
    fn from(err: io::Error) -> Self {
        ConfigError::Io(err)
    }
}

impl From<serde_json::Error> for ConfigError {
    fn from(err: serde_json::Error) -> Self {
        ConfigError::Parse(err)
    }
}

fn load_config(path: &str) -> Result<Config, ConfigError> {
    let contents = std::fs::read_to_string(path)?;
    let config: Config = serde_json::from_str(&contents)?;
    Ok(config)
}

fn get_user_age(users: &HashMap<String, User>, id: &str) -> Option<u32> {
    users.get(id).map(|user| user.age)
}

fn divide(a: i32, b: i32) -> Option<i32> {
    a.checked_div(b)
}

//=====
// If unwrap is genuinely safe, document why
//=====

fn get_first_line(text: &str) -> &str {
    text.lines()
}

```

```

    .next()
    .unwrap() // Safe: lines() always yields at least one line (possibly empty)
}

```

Real-World Impact: A web service using unwrap() crashed on malformed requests. Proper error handling returned 400 errors instead of crashing.

Anti-Pattern: RefCell/Mutex Without Consideration

The Pattern: Using interior mutability as a first resort rather than last resort.

```

use std::cell::RefCell;

//=====
// ✗ CORRECT: Proper mutability
//=====

struct Application {
    state: RefCell<AppState>,
    config: RefCell<Config>,
    users: RefCell<Vec<User>>,
}

impl Application {
    fn process(&self) {
        let mut state = self.state.borrow_mut();
        let config = self.config.borrow();
        let mut users = self.users.borrow_mut();

        // Runtime borrow checking overhead
        // Can panic if borrowing rules violated
        // Hidden mutation through shared reference
    }
}

```

Why It's Problematic: - Runtime cost of borrow checking - Can panic if borrowing rules violated - Hides mutability in API (& self but mutates) - Indicates fighting the type system - Makes code harder to reason about

The Solution: Design with proper ownership and mutability; use RefCell only when genuinely needed.

```

//=====
// ✗ CORRECT: Proper mutability
//=====

struct Application {
    state: AppState,
    config: Config,
    users: Vec<User>,
}

impl Application {

```

```

fn process(&mut self) { // Honest about mutation
    // Compile-time borrow checking
    // Clear ownership and mutation
}

fn read_config(&self) -> &Config {
    &self.config // No runtime overhead
}
}

//=====
// Use RefCell only when necessary (e.g., graph structures, caching)
//=====
use std::cell::RefCell;

struct Node {
    value: i32,
    children: RefCell<Vec<Node>>, // Justified: allows mutation during traversal
}

//=====
// For shared ownership with mutation, use Arc<Mutex<T>>
//=====
use std::sync::{Arc, Mutex};

fn concurrent_modification() {
    let data = Arc::new(Mutex::new(vec![1, 2, 3]));

    let data_clone = Arc::clone(&data);
    std::thread::spawn(move || {
        data_clone.lock().unwrap().push(4);
    });
}

```

Real-World Impact: Code using RefCell throughout had runtime panics from double borrows and 10% overhead from runtime checks. Restructuring with proper &mut eliminated both issues.

Anti-Pattern: Ignoring Send/Sync Implications

The Pattern: Using thread-unsafe types across threads without understanding Send/Sync bounds.

```

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

//=====
// ✗ ANTI-PATTERN: Thread-unsafe types in threads
//=====

fn share_across_threads() {
    let data = Rc::new(RefCell::new(vec![1, 2, 3]));
    let data_clone = Rc::clone(&data);
}

```

```

// Compile error: Rc and RefCell aren't Send/Sync
// thread::spawn(move || {
//     data_clone.borrow_mut().push(4);
// });

// "Fix" with unsafe (WRONG!)
let ptr = Rc::into_raw(data_clone);
thread::spawn(move || {
    unsafe {
        let rc = Rc::from_raw(ptr);
        rc.borrow_mut().push(4);
        // Data race! Undefined behavior!
    }
});
}

```

Why It's Problematic: - Data races cause undefined behavior - Rc uses non-atomic reference counting (race condition) - RefCell uses non-atomic borrow tracking (race condition) - Circumventing Send/Sync with unsafe defeats safety

The Solution: Use thread-safe alternatives (Arc, Mutex) or restructure to avoid sharing.

```

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

//=====
// ✅ CORRECT: Thread-safe types
//=====

fn share_across_threads_safely() {
    let data = Arc::new(Mutex::new(vec![1, 2, 3]));
    let data_clone = Arc::clone(&data);

    let handle = thread::spawn(move || {
        data_clone.lock().unwrap().push(4);
    });

    handle.join().unwrap();

    let final_data = data.lock().unwrap();
    println!("{:?}", *final_data);
}

//=====
// Or use message passing (often better)
//=====

use std::sync::mpsc;

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

    thread::spawn(move || {

```

```

        tx.send(vec![1, 2, 3, 4]).unwrap();
    });

    let data = rx.recv().unwrap();
    println!("{}: {:?}", data);
}

```

Real-World Impact: Code using unsafe to share Rc across threads had intermittent crashes from data races. Switching to Arc<Mutex> eliminated the undefined behavior.

API Design Mistakes

These anti-patterns create poor interfaces that are hard to use correctly or inconsistent with Rust ecosystem conventions.

Anti-Pattern: Stringly-Typed APIs

The Pattern: Using strings to represent structured data that should have proper types.

```

//=====
// ✗ ANTI-PATTERN: String-based API
//=====

fn set_log_level(level: &str) {
    match level {
        "debug" | "info" | "warn" | "error" => { /* ... */ }
        _ => panic!("Invalid log level"),
    }
}

fn parse_color(color: &str) -> (u8, u8, u8) {
    match color {
        "red" => (255, 0, 0),
        "green" => (0, 255, 0),
        "blue" => (0, 0, 255),
        _ => panic!("Unknown color"),
    }
}

//=====
// Caller must know valid strings
//=====

set_log_level("degub"); // Typo! Runtime panic

```

Why It's Problematic: - No compile-time validation - Easy to make typos - Runtime errors for invalid values - No IDE autocomplete - Poor discoverability

The Solution: Use enums for fixed sets of values.

```

//=====
// ✓ CORRECT: Type-safe enums
//=====

```

```

//=====
#[derive(Debug, Clone, Copy)]
enum LogLevel {
    Debug,
    Info,
    Warn,
    Error,
}

fn set_log_level(level: LogLevel) {
    match level {
        LogLevel::Debug => { /* ... */ }
        LogLevel::Info => { /* ... */ }
        LogLevel::Warn => { /* ... */ }
        LogLevel::Error => { /* ... */ }
    }
}

#[derive(Debug, Clone, Copy)]
enum Color {
    Red,
    Green,
    Blue,
    Rgb(u8, u8, u8),
}

impl Color {
    fn to_rgb(self) -> (u8, u8, u8) {
        match self {
            Color::Red => (255, 0, 0),
            Color::Green => (0, 255, 0),
            Color::Blue => (0, 0, 255),
            Color::Rgb(r, g, b) => (r, g, b),
        }
    }
}

//=====
// Compile-time checked, IDE autocomplete
//=====

set_log_level(LogLevel::Debug);
//=====

// set_log_level(LogLevel::Degub); // Compile error!
//=====

let color = Color::Red;
let custom = Color::Rgb(128, 128, 128);

```

Real-World Impact: A database library using string-based query types had 30% of user issues from typos. Switching to enums eliminated these errors.

Anti-Pattern: Boolean Parameter Trap

The Pattern: Using boolean parameters where the meaning is unclear at call sites.

```
//=====
// ✗ ANTI-PATTERN: Unclear boolean parameters
//=====

fn connect(host: &str, encrypted: bool, persistent: bool, verbose: bool) {
    // ...
}

//=====
// What do these booleans mean?
//=====

connect("localhost", true, false, true); // Unclear!
connect("localhost", false, true, false); // What?
```

Why It's Problematic: - Unclear intent at call sites - Easy to swap arguments - Hard to extend (what if you need 4 modes, not 2?) - Poor readability

The Solution: Use enums or builder pattern for clarity.

```
//=====
// ✓ CORRECT: Explicit enum parameters
//=====

enum Encryption {
    Encrypted,
    Plaintext,
}

enum Connection {
    Persistent,
    Transient,
}

enum Verbosity {
    Verbose,
    Quiet,
}

fn connect(
    host: &str,
    encryption: Encryption,
    connection: Connection,
    verbosity: Verbosity,
) {
    // ...
}

//=====
// Clear intent at call site
//=====
```

```
connect(  
    "localhost",  
    Encryption::Encrypted,  
    Connection::Transient,  
    Verbosity::Verbose,  
)  
  
//=====  
// Or use builder pattern  
//=====  
struct ConnectionBuilder {  
    host: String,  
    encrypted: bool,  
    persistent: bool,  
    verbose: bool,  
}  
  
impl ConnectionBuilder {  
    fn new(host: impl Into<String>) -> Self {  
        Self {  
            host: host.into(),  
            encrypted: false,  
            persistent: false,  
            verbose: false,  
        }  
    }  
  
    fn encrypted(mut self) -> Self {  
        self.encrypted = true;  
        self  
    }  
  
    fn persistent(mut self) -> Self {  
        self.persistent = true;  
        self  
    }  
  
    fn verbose(mut self) -> Self {  
        self.verbose = true;  
        self  
    }  
  
    fn connect(self) -> Connection {  
        // ...  
        unimplemented!()  
    }  
}  
  
//=====  
// Clear and fluent  
//=====  
let conn = ConnectionBuilder::new("localhost")  
    .encrypted()
```

```
.verbose()  
.connect();
```

Real-World Impact: A configuration API with 5 boolean parameters had numerous bugs from swapped arguments. Builder pattern eliminated the errors.

Anti-Pattern: Leaky Abstractions

The Pattern: Exposing implementation details in public APIs.

```
//=====  
// ✗ ANTI-PATTERN: Exposing internal details  
//=====  
  
pub struct Database {  
    pub connection_pool: Vec<Connection>, // Internal detail exposed  
    pub cache: HashMap<String, Vec<u8>>, // Implementation leaked  
}  
  
impl Database {  
    pub fn get_connection(&mut self) -> &mut Connection {  
        &mut self.connection_pool[0] // Caller can manipulate pool  
    }  
  
    pub fn query(&self, sql: &str) -> Vec<u8> {  
        self.cache.get(sql).cloned().unwrap_or_else(|| {  
            // Actually query database  
            vec![]  
        })  
    }  
}
```

Why It's Problematic: - Can't change implementation without breaking users - Exposes internal invariants that must be maintained - No encapsulation - Hard to evolve API

The Solution: Hide implementation details, expose only necessary interfaces.

```
//=====  
// ✓ CORRECT: Proper encapsulation  
//=====  
  
pub struct Database {  
    connection_pool: Vec<Connection>, // Private  
    cache: HashMap<String, Vec<u8>>, // Private  
}  
  
impl Database {  
    pub fn new(connection_string: &str) -> Result<Self, Error> {  
        Ok(Self {  
            connection_pool: vec![Connection::new(connection_string)?],  
            cache: HashMap::new(),  
        })  
    }  
}
```

```

    }

    pub fn query(&mut self, sql: &str) -> Result<Vec<Row>, Error> {
        // Returns proper type, not implementation detail (Vec<u8>)
        if let Some(cached) = self.cache.get(sql) {
            return Ok(deserialize_rows(cached));
        }

        let conn = self.get_connection_internal()?;
        let result = conn.execute(sql)?;
        let serialized = serialize_rows(&result);
        self.cache.insert(sql.to_string(), serialized);
        Ok(result)
    }

    // Private helper
    fn get_connection_internal(&mut self) -> Result<&mut Connection, Error> {
        self.connection_pool.first_mut()
            .ok_or(Error::NoConnections)
    }
}

struct Row {
    // Proper abstraction
}

fn serialize_rows(rows: &[Row]) -> Vec<u8> {
    unimplemented!()
}

fn deserialize_rows(data: &[u8]) -> Vec<Row> {
    unimplemented!()
}

```

Real-World Impact: A library exposing internal Vec was locked into using Vec even when a different structure would be better. Proper encapsulation would have allowed evolution.

Anti-Pattern: Returning Owned When Borrowed Suffices

The Pattern: Returning owned `String` or `Vec<T>` when a borrowed reference would work.

```

//=====
// ✗ ANTI-PATTERN: Unnecessary ownership transfer
//=====

struct User {
    name: String,
    email: String,
}

impl User {
    fn get_name(&self) -> String {

```

```

        self.name.clone() // Allocates on every call
    }

    fn get_email(&self) -> String {
        self.email.clone() // Unnecessary clone
    }
}

fn format_user(user: &User) -> String {
    format!("{} <{}>", user.get_name(), user.get_email())
    // Two allocations just to borrow the data
}

```

Why It's Problematic: - Forces allocation on callers - Wasteful when data is just read - Less flexible (can't pattern match on borrowed data efficiently)

The Solution: Return references for data you already own.

```

//=====
// ✅ CORRECT: Return references
//=====

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

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

fn format_user(user: &User) -> String {
    format!("{} <{}>", user.name(), user.email())
    // No extra allocations
}

//=====
// Only return owned when you create new data
//=====

impl User {
    fn display_name(&self) -> String {
        format!("{} ({})", self.name, self.email)
        // Creating new data, ownership transfer makes sense
    }
}

```

Real-World Impact: An ORM returning cloned Strings for every field access had 10x memory overhead. Switching to references eliminated the waste.

Anti-Pattern: Overengineered Generic APIs

The Pattern: Using complex generic bounds when concrete types would suffice.

```
//=====
// ✗ ANTI-PATTERN: Overly generic for no benefit
//=====

fn print_items<I, T>(items: I)
where
    I: IntoIterator<Item = T>,
    T: std::fmt::Display + std::fmt::Debug + Clone + Send + Sync + 'static,
{
    for item in items {
        println!("{}", item);
    }
}

//=====
// Confusing signature for simple functionality
//=====
```

Why It's Problematic: - Harder to understand - Confusing error messages - Unnecessary complexity -
Most callers use concrete types anyway

The Solution: Use concrete types or simple generics that provide real value.

```
//=====
// ✓ CORRECT: Simple and clear
//=====

fn print_items(items: &[impl std::fmt::Display]) {
    for item in items {
        println!("{}", item);
    }
}

//=====
// Or be generic only where it adds value
//=====

fn print_items_generic<T: std::fmt::Display>(items: &[T]) {
    for item in items {
        println!("{}", item);
    }
}

//=====
// Reserve complex bounds for when truly needed
//=====

use std::hash::Hash;
use std::collections::HashMap;

fn count_occurrences<T>(items: &[T]) -> HashMap<&T, usize>
where
```

```

T: Hash + Eq, // Actually needed for HashMap
{
    let mut counts = HashMap::new();
    for item in items {
        *counts.entry(item).or_insert(0) += 1;
    }
    counts
}

```

Real-World Impact: A library with overly generic APIs confused users and had poor compile times. Simplifying to concrete types where appropriate improved both.

Summary

Anti-patterns represent accumulated wisdom from mistakes made across the Rust ecosystem. Recognizing these patterns helps you write better code from the start and refactor problematic code when you encounter it.

Key Principles to Avoid Anti-Patterns

Ownership and borrowing: - Clone only when semantically necessary, not to satisfy the borrow checker - Use references with lifetimes; resort to Rc/Arc only when genuinely sharing ownership - Understand why the borrow checker complains before “fixing” with workarounds

Performance: - Chain iterators without intermediate collections - Use arrays for fixed-size data, Vec for dynamic - Prefer static dispatch (generics) over dynamic (trait objects) when types known - Work with &str until you need ownership

Safety: - Use unsafe only when necessary and document invariants - Handle errors with Result/Option, not unwrap() - Use proper mutability (&mut) rather than interior mutability by default - Respect Send/Sync bounds; don’t circumvent with unsafe

API design: - Use types (enums) instead of strings/booleans for clarity - Hide implementation details, expose minimal necessary interface - Return references when data already owned, own when creating new data - Keep generics simple unless complexity provides clear value

Learning from Anti-Patterns

When you find yourself: - **Adding .clone() to fix errors:** Understand borrowing first - **Wrapping everything in Rc/Arc:** Reconsider ownership structure - **Fighting the borrow checker:** The design might be wrong - **Using unsafe to “fix” safety errors:** Safe solution usually exists - **Creating complex generic APIs:** Simplicity often better

The Rust compiler is your ally. When it rejects code, it’s usually protecting you from real bugs. Instead of fighting it with workarounds, understand what it’s preventing and design accordingly. The patterns that feel natural in Rust—ownership, borrowing, iterators, enums—exist because they align with the language’s guarantees.

Master these anti-patterns not to never make mistakes, but to recognize and fix them quickly when they appear in your code.