Part I: Core Language Mechanics

- memory-ownership-patterns: Master ownership, borrowing, and lifetimes as the foundation for safe Rust.
- struct-enum-patterns: Model complex data with expressive structs, enums, and pattern-centric APIs.
- **trait-design-patterns**: Compose reusable behavior through carefully scoped traits and idioms.
- **generics-polymorphism**: Use generics and trait bounds to build flexible, zero-cost abstractions.
- **builder-api-design**: Design fluent builder types that validate configuration before construction.
- **lifetime-patterns**: Reason about lifetime annotations to uphold reference safety without clutter.
- **functional-programming**: Adopt iterator, closure, and declarative styles for clearer logic.
- **pattern-matching-destructuring**: Leverage `match` and destructuring to unpack data cleanly.
- **iterator-patterns-combinators**: Chain iterator adapters and combinators for elegant data pipelines.
- **reference-binding**: Handle borrowing, references, and iterator lifetimes in tandem.
- **error-handling-architecture**: Structure `Result` flows, bubbling, and context-rich error types.

Part II: Collections & Data Structures

- **vec-slice-manipulation**: Work efficiently with contiguous buffers via `Vec` , slices, and views.
- **string-processing**: Parse, transform, and construct UTF-8 strings without needless copies.

- **hashmap-hashset-patterns**: Choose and tune hash-based collections for fast lookups.
- **advanced-collections**: Employ specialized containers such as `BTreeMap` and `BinaryHeap`.

Part III: Concurrency & Parallelism

- **threading-patterns**: Launch threads, share state, and coordinate via channels and locks.
- **async-runtime-patterns**: Drive async tasks atop executors while keeping latency predictable.
- **atomic-lock-free**: Write lock-free data paths using atomics and correct memory ordering.
- **parallel-algorithms**: Apply Rayon-style abstractions for scalable data-parallel workloads.

Part IV: Smart Pointers & Memory

- **smart-pointer-patterns**: Employ `Box`, `Rc`, `Arc`, and custom pointers for ownership control.
- **unsafe-rust-patterns**: Encapsulate unsafe code responsibly with airtight invariants.

Part V: I/O & Serialization

- **synchronous-io**: Build blocking I/O services with the standard library's stream traits.
- **async-io-patterns**: Structure non-blocking I/O stacks using async/await primitives.
- **serialization-patterns**: Encode and decode data via Serde and custom formats.

Part VI: Macros & Metaprogramming

- **declarative-macros**: Author `macro_rules!` DSLs that expand ergonomically.
- **procedural-macros**: Craft derive and attribute macros with `syn` and `quote`.

Part VII: Systems Programming

- **ffi-c-interop**: Bridge Rust with C interfaces while honoring safety contracts.
- **network-programming**: Implement network clients and servers with std or Tokio.
- **database-patterns**: Integrate SQL/NoSQL backends through Diesel, SQLx, or lower-level APIs.
- **testing-benchmarking**: Build resilient test suites, benches, and property checks.
- **performance-optimization**: Profile, measure, and tune for predictable performance wins.
- **embedded-realtime-patterns**: Apply Rust in `no_std`, RTIC, and real-time control scenarios.

Appendices

- **appendix-a-quick-reference**: Keep a handy cheat sheet of syntax, commands, and patterns.
- **appendix-b-design-patterns**: Browse a catalog of reusable Rust design templates.
- **appendix-c-anti-patterns**: Recognize and avoid common pitfalls and code smells.

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.replacen("~", "/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<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 `disarm`ed to prevent the cleanup from running.

```rust
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
```

```rust
#[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.

```rust
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.

```rust
// 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.

```rust
// 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.

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

```rust
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.

```
// 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)