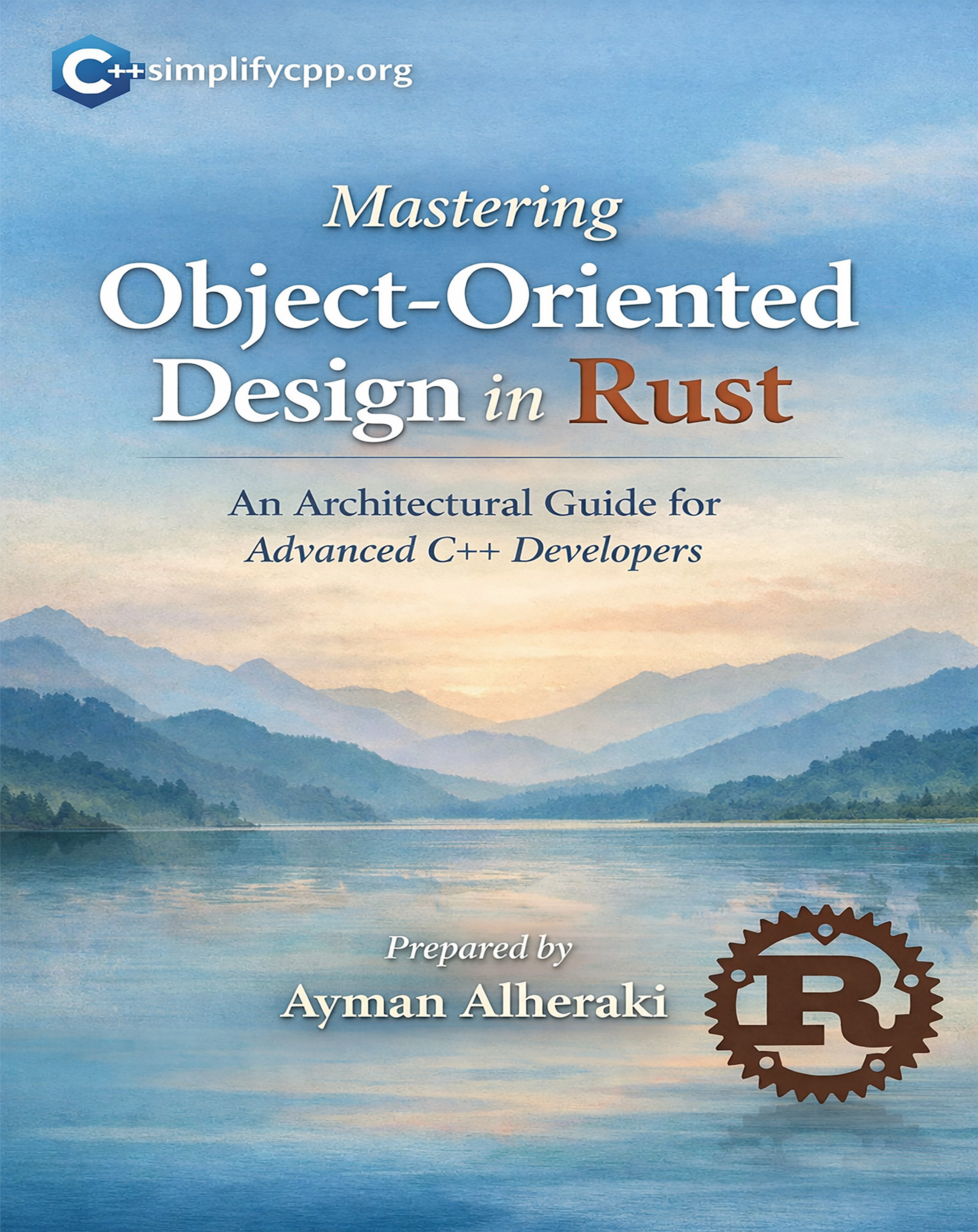

simplifycpp.org

Mastering Object-Oriented Design *in* Rust

An Architectural Guide for
Advanced C++ Developers

Prepared by
Ayman Alheraki

Mastering Object-Oriented Design in Rust

An Architectural Guide for Advanced C++ Developers

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

Prepared by Ayman Alheraki

February 2026

Contents

Author’s Introduction

Object-Oriented Programming (OOP) has long been one of the foundational pillars of modern software engineering. It is not merely a technique for organizing code; it is an architectural mindset that enables engineers to build scalable, maintainable, well-structured, and responsibility-driven systems. Misunderstanding OOP does not simply produce poor code — it produces fragile systems that are difficult to evolve, extend, or trust.

Throughout decades of working with C and C++, I came to understand that the real power of OOP does not lie in inheritance or polymorphism alone. Its true strength lies in architectural discipline: clear separation of responsibilities, protection of internal invariants, well-defined interfaces, and prevention of misuse.

Yet even with all that expressive power, traditional systems have carried a persistent and serious vulnerability: memory safety.

Rust represents a fundamental shift in engineering thinking. It does not merely provide an alternative object model; it introduces a rigorous ownership and borrowing system that elevates memory safety to a compile-time guarantee. Entire classes of runtime failures are transformed into early, deterministic compiler errors.

When sound OOP principles are combined with Rust’s ownership model, borrowing rules, and strict aliasing guarantees, the result is a design environment that:

- Preserves architectural clarity.
- Prevents incorrect usage by construction.

- Eliminates common memory errors such as use-after-free and double-free.
- Reduces data race risks in concurrent systems.
- Strengthens confidence in security-critical applications.

For these reasons, I have dedicated this booklet to exploring Object-Oriented Design in Rust from an architectural perspective — specifically for experienced C++ developers. The goal is not to discard familiar concepts, but to reinterpret and refine them within a safer, more disciplined, and more predictable system.

If OOP gives you expressive power, Rust gives you structural enforcement. When expressive power is combined with enforced safety, the outcome is a robust, high-confidence architecture suitable for systems where correctness and security are non-negotiable.

This booklet is not a language tutorial. It is a conceptual reconstruction of object-oriented architecture under the guarantees and constraints of Rust.

Ayman Alheraki

Preface

Why Rust OOP is Not C++ OOP

Rust supports many *object-oriented outcomes* (encapsulation, polymorphism, modular design), but it intentionally avoids the core mechanism that dominates classical C++ OOP: **implementation inheritance**. In Rust, the center of gravity is:

- **Composition** over inheritance (build types out of other types, not base classes),
- **Traits** for shared behavior (interfaces/contracts),
- **Algebraic data types** (`enum`) + exhaustive `match` for robust modeling,
- **Ownership and borrowing** as part of the design contract (APIs that prevent misuse).

In Modern C++, OOP often means:

- class hierarchies,
- virtual dispatch,
- `protected` state shared across inheritance layers,
- managing aliasing and lifetime risks via discipline, reviews, and tooling.

In Rust, the language forces architecture to be explicit:

- you decide between **static dispatch** (generics) and **dynamic dispatch** (trait objects),
- you model state transitions explicitly (often with `enum`),
- you choose ownership (`T`), borrowing (`&T`), mutation (`&mut T`), or shared ownership (`Rc`/`Arc`),
- thread-safety is expressed in the type system (`Send`/`Sync`).

A Minimal Example: "Interface" Behavior Without Inheritance

```
trait Renderer {
    fn render(&self, input: &str) -> String;
}

struct Html;
impl Renderer for Html {
    fn render(&self, input: &str) -> String {
        format!("<p>{}</p>", input)
    }
}

struct Markdown;
impl Renderer for Markdown {
    fn render(&self, input: &str) -> String {
        format!("**{}**", input)
    }
}
```

No base class, no inheritance chain. The polymorphism decision comes next: generics (static) or trait objects (dynamic).

Static Dispatch: Faster, More "Template-like"

```rust
fn print_rendered<R: Renderer>(r: &R, s: &str) {
    println!("{}", r.render(s));
}

fn main() {
    let h = Html;
    let m = Markdown;
    print_rendered(&h, "Hello");
    print_rendered(&m, "Hello");
}
```

Dynamic Dispatch: Runtime Polymorphism (Virtual-like)

```rust
fn print_rendered_dyn(r: &dyn Renderer, s: &str) {
    println!("{}", r.render(s));
}

fn main() {
    let renderers: Vec<Box<dyn Renderer>> = vec![Box::new(Html), Box::new(Markdown)];
    for r in &renderers {
        print_rendered_dyn(r.as_ref(), "Hello");
    }
}
```

The design is explicit: you *choose* the dispatch strategy, and the compiler enforces the consequences.

Mental Model Shift Required

This booklet assumes you already understand how OOP is used in real C++ systems: interfaces, factories, plugin models, ownership, RAII, and performance tradeoffs. The shift in Rust is mainly **architectural**:

1. **From inheritance trees to composition graphs.**
   In Rust, “is-a” inheritance is replaced by “has-a” composition and “can-do” traits.

2. **From “I will be careful” to “the type system will prevent it”.**
   Lifetimes and borrowing turn many runtime bugs into compile-time design decisions.

3. **From “object identity everywhere” to “data modeling first”.**
   Many OOP designs in C++ are simpler and safer as Rust `enum`-based models.

4. **From ad-hoc thread safety to type-driven thread safety.**
   Shared mutable state is *not the default*; you must opt into it explicitly.

Replacing a Class Hierarchy with an `enum`

Instead of:

- `class Expr { virtual eval() };`
- `class Add : Expr {...}; class Mul : Expr {...};`

You often model with:

```
#[derive(Clone)]
enum Expr {
    Num(i64),
    Add(Box<Expr>, Box<Expr>),
    Mul(Box<Expr>, Box<Expr>),
}

impl Expr {
    fn eval(&self) -> i64 {
        match self {
            Expr::Num(n) => *n,
            Expr::Add(a, b) => a.eval() + b.eval(),
            Expr::Mul(a, b) => a.eval() * b.eval(),
        }
    }
}

fn main() {
    let e = Expr::Add(Box::new(Expr::Num(2)), Box::new(Expr::Mul(Box::new(Expr::Num(3)),
    ↪  Box::new(Expr::Num(4)))));
    println!("{}", e.eval()); // 2 + (3*4) = 14
}
```

This replaces dynamic dispatch with exhaustive modeling, eliminates downcasts, and makes extension rules explicit (add a variant *and* handle it everywhere).

Ownership as an API Contract (Not Just Memory Management)

```
struct Cache {
    // Internal invariant: keys are unique, values are owned by the cache.
    items: std::collections::HashMap<String, String>,
}

impl Cache {
    fn new() -> Self {
        Self { items: std::collections::HashMap::new() }
    }
```

```rust
    // Borrowing: caller keeps ownership of the key/value data sources.
    fn insert(&mut self, key: &str, value: &str) {
        self.items.insert(key.to_string(), value.to_string());
    }

    // Borrowing return: caller cannot outlive the cache.
    fn get(&self, key: &str) -> Option<&str> {
        self.items.get(key).map(|s| s.as_str())
    }
}

fn main() {
    let mut c = Cache::new();
    c.insert("lang", "Rust");
    if let Some(v) = c.get("lang") {
        println!("{}", v);
    }
}
```

In Rust, method signatures communicate ownership and lifetime constraints precisely, making many “illegal states” unrepresentable.

What This Book Assumes You Already Know

This booklet is written for an experienced C++ engineer. It does **not** teach:

- basic syntax, variables, loops,
- what OOP is at an introductory level,
- general software engineering definitions.

It assumes you already have:

- strong OOP intuition: encapsulation, interfaces, polymorphism, cohesion/coupling,
- experience with RAII and resource ownership in C++,
- comfort with templates and performance reasoning,
- familiarity with multi-threaded design and race conditions.

Rust prerequisites assumed (minimal):

- basic understanding of **ownership/borrowing** and why `&mut` is exclusive,
- basic use of **traits** and **generics**,
- the idea of **Option/Result** as explicit control-flow types.

When needed, we will restate these concepts *only* as they affect OOP architecture and API design.

What You Will Master by the End

By the end of this booklet, you will be able to design Rust systems with the same confidence you have in Modern C++ OOP, but using Rust-native mechanisms. Specifically, you will master:

1. **Rust "class design" using `struct` + `impl`**
   Stable encapsulation boundaries, invariants, and ergonomic APIs.
2. **Trait-based abstraction (interfaces) at an expert level**
   Trait bounds, associated types, and clean architectural layering.
3. **Polymorphism tradeoffs: generics vs `dyn`**
   Knowing when to pick static dispatch, dynamic dispatch, or enum modeling.

4. **Replacing inheritance correctly**

   Composition, delegation, newtype isolation, and extension traits.

5. **Enum-first modeling for correctness**

   State machines, protocol models, command models, and AST-style representations.

6. **Ownership-driven API design**

   Using `T` vs `&T` vs `&mut T` vs `Rc/Arc` intentionally.

7. **Interior mutability and controlled shared state**

   Safe patterns using `RefCell`, `Mutex`, `RwLock` with clear tradeoffs.

8. **Thread-safe object architectures on Windows**

   Correct usage of `Arc` + synchronization primitives for concurrent designs.

A Final Snapshot: Thread-Safe "Service Object" Pattern (Windows-Friendly)

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

trait Logger: Send + Sync {
    fn log(&self, msg: &str);
}

struct MemLogger {
    buf: Mutex<Vec<String>>,
}

impl MemLogger {
    fn new() -> Self {
        Self { buf: Mutex::new(Vec::new()) }
    }
}

impl Logger for MemLogger {
    fn log(&self, msg: &str) {
        let mut g = self.buf.lock().unwrap();
        g.push(msg.to_string());
    }
}

fn main() {
    let logger: Arc<dyn Logger> = Arc::new(MemLogger::new());

    let t1 = {
        let l = Arc::clone(&logger);
        thread::spawn(move || l.log("from t1"))
    };
    let t2 = {
        let l = Arc::clone(&logger);
        thread::spawn(move || l.log("from t2"))
    };

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

    // Downcasting is not the default in Rust OOP; if you need inspection APIs, design them
    ↪  explicitly.
    // Here we just demonstrate concurrency-safe shared behavior through a trait object.
}
```

This is the Rust OOP promise: explicit abstraction + explicit ownership + explicit concurrency guarantees.

Part I

Core Object Modeling in Rust

Structs, `impl` Blocks, and Encapsulation

1.1 Structs as Class Equivalents

In Rust, `struct` + `impl` is the closest practical equivalent to a C++ class: it packages *state* (fields) and *behavior* (methods) with strong encapsulation defaults. Unlike C++, Rust does not use implementation inheritance; object design tends to be *composition-first* with explicit interfaces (traits) when polymorphism is needed (covered later). For core modeling, think:

- **State:** `struct` fields.
- **Behavior:** methods inside `impl`.
- **Construction:** associated functions (commonly `new`) and factories.
- **Resource safety:** deterministic cleanup via RAII/`Drop`.

```rust
pub struct Point {
    x: i32,
    y: i32,
}

impl Point {
    pub fn new(x: i32, y: i32) -> Self {
        Self { x, y }
    }

    pub fn x(&self) -> i32 { self.x }
    pub fn y(&self) -> i32 { self.y }

    pub fn translate(&mut self, dx: i32, dy: i32) {
        self.x += dx;
        self.y += dy;
    }
}

fn main() {
    let mut p = Point::new(10, 20);
    p.translate(5, -3);
    println!("({}, {})", p.x(), p.y());
}
```

1.2 Method Definitions and `Self`

Rust methods explicitly state how they access the receiver:

- `&self`: shared borrow (read-only; allows aliasing).
- `&mut self`: exclusive mutable borrow (no aliasing; enforces safe mutation).
- `self`: takes ownership (move); useful for builders, consuming APIs, state transitions.

This receiver design is not cosmetic; it is an API contract enforced by the compiler.

Read-only vs Mutable Methods

```
#[derive(Clone)]
pub struct Counter {
    value: u64,
}

impl Counter {
    pub fn new() -> Self { Self { value: 0 } }

    pub fn value(&self) -> u64 { self.value }      // &self
    pub fn inc(&mut self) { self.value += 1; }     // &mut self

    pub fn into_value(self) -> u64 { self.value }  // self (moves/consumes)
}

fn main() {
    let mut c = Counter::new();
    c.inc();
    println!("{}", c.value());
    let v = c.into_value();
    println!("{}", v);
}
```

Self and Return-Style APIs

Returning `Self` is common for fluent/immutable APIs ("builder-like" without mutation):

```
#[derive(Clone)]
pub struct Flags {
    bits: u32,
}

impl Flags {
    pub fn empty() -> Self { Self { bits: 0 } }

    pub fn with_debug(self) -> Self { Self { bits: self.bits | 0x1 } }
    pub fn with_trace(self) -> Self { Self { bits: self.bits | 0x2 } }

    pub fn bits(&self) -> u32 { self.bits }
}

fn main() {
    let f = Flags::empty().with_debug().with_trace();
    println!("bits={:#x}", f.bits());
}
```

1.3 Associated Functions

Associated functions are functions scoped to a type, called as `Type::func(...)`. They are the standard way to express:

- constructors (`new`, `with_capacity`),
- factories (`from_*`),
- conversion helpers,
- "static" utilities tied to a type.

Idiomatic Construction: `new` + Factories

```
use std::path::PathBuf;

pub struct Config {
    app_name: String,
    data_dir: PathBuf,
}

impl Config {
    pub fn new(app_name: impl Into<String>, data_dir: impl Into<PathBuf>) -> Self {
        Self { app_name: app_name.into(), data_dir: data_dir.into() }
    }

    pub fn for_windows_default(app_name: impl Into<String>) -> Self {
        // Example only; choose your policy per product requirements.
        let base = PathBuf::from(r"C:\ProgramData");
        let name = app_name.into();
        Self { data_dir: base.join(&name), app_name: name }
    }
}

fn main() {
    let c1 = Config::new("ForgeVM", r"C:\Temp\ForgeVM");
    let c2 = Config::for_windows_default("ForgeVM");
    println!("{} -> {:?}", c1.app_name, c1.data_dir);
    println!("{} -> {:?}", c2.app_name, c2.data_dir);
}
```

1.4 Visibility Control and API Boundaries

Rust encapsulation is module-based and strict by default:

- Items are private unless declared pub.
- Fields are private unless declared pub.
- You design an API by exposing a minimal public surface and keeping invariants behind private state.

Public Type, Private Fields: Stable Encapsulation

```
pub mod net {
```

```rust
    pub struct Endpoint {
        host: String,    // private field
        port: u16,       // private field
    }

    impl Endpoint {
        pub fn new(host: impl Into<String>, port: u16) -> Self {
            Self { host: host.into(), port }
        }

        pub fn host(&self) -> &str { &self.host }
        pub fn port(&self) -> u16 { self.port }
    }
}

fn main() {
    let ep = net::Endpoint::new("127.0.0.1", 8080);
    // ep.host = "x".into(); // ERROR: private field
    println!("{}:{}", ep.host(), ep.port());
}
```

Crate-Level Boundaries: `pub(crate)`

```rust
pub struct Token(String);

impl Token {
    pub fn as_str(&self) -> &str { &self.0 }
}

// Visible inside the crate, hidden from external users.
pub(crate) fn sign_internal(input: &str) -> Token {
    Token(format!("signed:{input}"))
}
```

Facade API: Re-export What You Want Users to See

```rust
mod internal {
    pub struct Engine {
        pub(crate) state: u32,
    }

    impl Engine {
        pub fn new() -> Self { Self { state: 0 } }
        pub fn tick(&mut self) { self.state += 1; }
        pub fn state(&self) -> u32 { self.state }
    }
}

// Public facade exposes only the intended type.
pub use internal::Engine;

fn main() {
    let mut e = Engine::new();
    e.tick();
    println!("{}", e.state());
}
```

1.5 Enforcing Invariants

The most important “class design” skill in Rust is making invalid states unrepresentable (or at least unconstructible). Common techniques:

- private fields + smart constructors,
- validated setters (or no setters at all),
- newtype wrappers to distinguish domains,

- type-state modeling for stateful protocols,
- RAII guards for resource lifetimes.

Smart Constructor + Validated API

```rust
#[derive(Debug, Clone)]
pub struct Username(String);

#[derive(Debug)]
pub enum UsernameError {
    Empty,
    TooLong,
    BadChar,
}

impl Username {
    pub fn parse(s: &str) -> Result<Self, UsernameError> {
        if s.is_empty() { return Err(UsernameError::Empty); }
        if s.len() > 24 { return Err(UsernameError::TooLong); }
        if !s.chars().all(|c| c.is_ascii_alphanumeric() || c == '_') {
            return Err(UsernameError::BadChar);
        }
        Ok(Self(s.to_string()))
    }

    pub fn as_str(&self) -> &str { &self.0 }
}

fn main() {
    let u = Username::parse("Ayman_1989").unwrap();
    println!("{:?}", u);
    // let bad = Username::parse("bad name").unwrap(); // would error
}
```

Newtype to Prevent Domain Confusion

In C++, two `int`s can silently mix. In Rust, newtypes make intent explicit.

```
#[derive(Copy, Clone, Debug)]
pub struct UserId(u64);

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

fn load_user(_id: UserId) {}
fn load_order(_id: OrderId) {}

fn main() {
    let u = UserId(7);
    let o = OrderId(7);
    load_user(u);
    load_order(o);
    // load_user(o); // ERROR: mismatched types
}
```

Type-State Pattern: Illegal Operations Do Not Compile

A practical way to enforce protocol order without runtime flags.

```
use std::marker::PhantomData;

pub struct Disconnected;
pub struct Connected;

pub struct Client<State> {
    addr: String,
    _st: PhantomData<State>,
}

impl Client<Disconnected> {
    pub fn new(addr: impl Into<String>) -> Self {
        Self { addr: addr.into(), _st: PhantomData }
    }

    pub fn connect(self) -> Client<Connected> {
        Client { addr: self.addr, _st: PhantomData }
    }
}

impl Client<Connected> {
    pub fn send(&self, msg: &str) {
        println!("send to {}: {}", self.addr, msg);
    }

    pub fn disconnect(self) -> Client<Disconnected> {
        Client { addr: self.addr, _st: PhantomData }
    }
}

fn main() {
    let c = Client::new("127.0.0.1:9000");
    // c.send("hi"); // ERROR: no method `send` for Disconnected
    let c = c.connect();
    c.send("hello");
    let _c = c.disconnect();
}
```

1.6 Comparison with C++ (Modern)

This chapter maps familiar C++ concepts to Rust design choices.

C++ Classes vs Rust `struct` + `impl`

- **Encapsulation:** Rust defaults to private; public API is explicit via `pub`.
- **Mutation control:** Rust enforces exclusive mutation with `&mut self` (no data races by default).
- **Inheritance:** Rust omits implementation inheritance; encourages composition and traits.

Constructors

- **C++:** overloaded constructors, initializer lists, implicit conversions risks.
- **Rust:** associated functions (`new`, `from_*`) and builders; no implicit constructors.
- **Invariant enforcement:** Rust typically uses private fields + validated constructors returning `Result`.

Access Specifiers and Boundaries

- **C++:** `private/protected/public`, friends.
- **Rust:** module privacy, `pub`, `pub(crate)`, controlled re-exports; "friend" is replaced by module-level design.

RAII and Deterministic Cleanup

- **Common ground:** both languages provide deterministic destruction.
- **Rust:** `Drop` runs automatically when values go out of scope; ownership rules make double-free/use-after-free structurally hard.

RAII Example: Scoped Resource Guard

```rust
use std::fs::File;
use std::io::{self, Write};

pub struct LogFile {
    f: File,
}

impl LogFile {
    pub fn create(path: &str) -> io::Result<Self> {
        Ok(Self { f: File::create(path)? })
    }

    pub fn write_line(&mut self, s: &str) -> io::Result<()> {
        writeln!(self.f, "{}", s)
    }
}

// Drop is optional here (File already closes on drop),
// but shown to demonstrate explicit finalization patterns.
impl Drop for LogFile {
    fn drop(&mut self) {
        let _ = self.f.flush();
    }
}
```

```
fn main() -> io::Result<()> {
    let mut log = LogFile::create(r"C:\Temp\rust_oop_log.txt")?;
    log.write_line("hello from Rust RAII")?;
    Ok(())
}
```

1.7 Outcome

After this chapter, you can:

- design “class-like” Rust types using `struct` + `impl`,
- express API intent precisely via `&self`/`&mut self`/`self`,
- build safe constructors and factories using associated functions,
- enforce invariants using privacy, validation, newtypes, and type-state modeling,
- define clear API boundaries with modules and `pub`/`pub(crate)` re-exports,
- apply RAII confidently with deterministic cleanup on Windows.

Traits as Interfaces

2.1 Traits as Contracts

Traits in Rust are *behavioral contracts*: they define a required set of methods (and optionally associated items) that a type must provide. This supports interface-based design without implementation inheritance. Unlike a base class, a trait does not carry data. A type may implement many traits, enabling orthogonal composition of behaviors.

A Minimal Contract

```rust
trait Serializer {
    fn serialize(&self) -> Vec<u8>;
}

struct User {
    id: u64,
    name: String,
}

impl Serializer for User {
    fn serialize(&self) -> Vec<u8> {
        // Very small example: a stable application-level encoding policy is your
        ↪  responsibility.
        // Here we demonstrate the idea, not an interchange format.
        let mut out = Vec::new();
        out.extend_from_slice(&self.id.to_le_bytes());
        out.extend_from_slice(self.name.as_bytes());
        out
    }
}

fn main() {
    let u = User { id: 7, name: "Ayman".to_string() };
    let bytes = u.serialize();
    println!("{} bytes", bytes.len());
}
```

Interface-First Design: Work Against the Trait, Not the Type

```
trait Clock {
    fn now_ms(&self) -> u64;
}

struct SystemClock;

impl Clock for SystemClock {
    fn now_ms(&self) -> u64 {
        use std::time::{SystemTime, UNIX_EPOCH};
        SystemTime::now()
            .duration_since(UNIX_EPOCH)
            .unwrap()
            .as_millis() as u64
    }
}

fn measure<C: Clock>(c: &C) -> u64 {
    let t0 = c.now_ms();
    let t1 = c.now_ms();
    t1.saturating_sub(t0)
}

fn main() {
    let c = SystemClock;
    println!("delta={}ms", measure(&c));
}
```

2.2 Default Methods

Traits can provide default method implementations, enabling shared behavior while keeping the contract explicit. This is similar in spirit to providing a non-virtual helper in a C++ interface, but with a crucial difference: the default lives in the trait and is inherited by implementors unless overridden.

Default Method with Optional Override

```
trait Logger {
    fn write(&self, msg: &str);

    fn info(&self, msg: &str) {  // default method
        self.write(&format!("[INFO] {msg}"));
    }

    fn error(&self, msg: &str) { // default method
        self.write(&format!("[ERROR] {msg}"));
    }
}
```

```rust
struct StdoutLogger;

impl Logger for StdoutLogger {
    fn write(&self, msg: &str) {
        println!("{msg}");
    }
}

struct PrefixLogger {
    prefix: String,
}

impl Logger for PrefixLogger {
    fn write(&self, msg: &str) {
        println!("{}{}", self.prefix, msg);
    }

    fn error(&self, msg: &str) { // override only what you need
        self.write(&format!("[FATAL] {msg}"));
    }
}

fn main() {
    let a = StdoutLogger;
    a.info("hello");
    a.error("oops");

    let b = PrefixLogger { prefix: "ForgeVM: ".to_string() };
    b.info("start");
    b.error("panic-like event");
}
```

2.3 Associated Types

Associated types let a trait declare placeholder types that each implementation chooses. This is essential for scalable API design because it avoids pushing every type parameter to the trait user. In practice, associated types provide a clean alternative to heavy template parameterization.

Associated Type in an Iterator-Like Contract

```
trait Source {
    type Item;

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

struct Counter {
    cur: u32,
    end: u32,
}

impl Counter {
    fn new(end: u32) -> Self { Self { cur: 0, end } }
}

impl Source for Counter {
    type Item = u32;

    fn next_item(&mut self) -> Option<Self::Item> {
        if self.cur >= self.end { return None; }
        let v = self.cur;
        self.cur += 1;
        Some(v)
    }
}

fn main() {
    let mut c = Counter::new(3);
    while let Some(x) = c.next_item() {
        println!("{x}");
    }
}
```

Why Associated Types Improve Ergonomics

With associated types, users write `S: Source` instead of `Source<T>` in many designs. This matches real systems: the implementor owns the decision of what `Item` means.

2.4 Trait Bounds

Trait bounds restrict generic parameters to those that satisfy required capabilities. This is the main mechanism for static polymorphism in Rust: compile-time dispatch with precise constraints.

Bounds in Signatures

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

fn join_with_comma<T: Display>(items: &[T]) -> String {
    let mut s = String::new();
    for (i, it) in items.iter().enumerate() {
        if i != 0 { s.push_str(", "); }
        s.push_str(&it.to_string());
    }
    s
}

fn main() {
    let v = [1, 2, 3];
    println!("{}", join_with_comma(&v));
}
```

where Clauses for Readable, Scalable Constraints

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

fn render_table<T>(rows: &[T]) -> String
where
    T: Display + Clone,
{
    let mut out = String::new();
    for r in rows {
        out.push_str(&format!("| {} |\n", r.clone()));
    }
    out
}

fn main() {
    let rows = vec!["A", "B", "C"];
    println!("{}", render_table(&rows));
}
```

Bounds as Design: Require What You Use

A strong Rust API avoids over-constraining. If you only need `Display`, do not require `Debug` or `Clone`. This keeps implementations open and prevents accidental coupling.

2.5 Blanket Implementations

A blanket implementation provides an `impl` of a trait for *all* types that meet certain bounds. This is a high-leverage abstraction tool: it creates reusable behavior layers without inheritance and without boilerplate.

Add a Default Capability to Many Types

```
trait HexDump {
    fn hex_dump(&self) -> String;
}

impl<T> HexDump for T
where
    T: AsRef<[u8]>,
{
    fn hex_dump(&self) -> String {
        let bytes = self.as_ref();
        let mut s = String::new();
        for (i, b) in bytes.iter().enumerate() {
            if i != 0 { s.push(' '); }
            s.push_str(&format!("{:02X}", b));
        }
        s
    }
}

fn main() {
    let a: Vec<u8> = vec![0, 1, 254, 255];
    let b: &[u8] = b"Rust";
    println!("{}", a.hex_dump());
    println!("{}", b.hex_dump());
}
```

Extension Traits: “Add Methods” Without Modifying the Type

This pattern is the Rust-native alternative to “deriving from a base class to gain helpers”.

```rust
trait StrExt {
    fn is_ascii_word(&self) -> bool;
}

impl StrExt for str {
    fn is_ascii_word(&self) -> bool {
        !self.is_empty() && self.chars().all(|c| c.is_ascii_alphanumeric() || c == '_')
    }
}

fn main() {
    let s = "ForgeVM_2026";
    println!("{}", s.is_ascii_word());
}
```

2.6 Comparison with Modern C++

This section maps trait-driven abstraction to familiar C++ mechanisms.

Pure Virtual Interfaces

- **C++:** `struct I { virtual f()=0; };` with vtable-based runtime dispatch.
- **Rust:** a trait can be used for static dispatch (generics) or dynamic dispatch (`dyn Trait`) when object-safe.

- **Key difference:** trait bounds are the default abstraction tool; dynamic dispatch is a deliberate choice.

Abstract Base Classes

- **C++:** abstract base classes can include shared state, partial implementations, protected members.
- **Rust:** traits do not contain fields; shared state is modeled via composition (a field that holds a helper type).
- **Result:** less implicit coupling; invariants remain local to concrete types.

CRTP

- **C++:** CRTP encodes "static polymorphism" using templates and inheritance.
- **Rust:** generics + trait bounds express static polymorphism directly; you write what you mean ("T implements Trait").

C++20 Concepts

- **C++20:** concepts constrain templates and improve error messages; they describe requirements on types.
- **Rust:** trait bounds have always served this role; traits also provide default methods and associated types, enabling both constraints and shared behavior.
- **Design posture:** Rust pushes interface-driven architecture early, with strong coherence rules to keep impls well-scoped.

2.7 Outcome

After this chapter, you can:

- model interfaces as explicit, composable trait contracts,
- use default methods to share behavior without inheritance,
- design ergonomic APIs using associated types instead of sprawling generics,
- express precise capabilities with trait bounds and `where` clauses,
- apply blanket implementations and extension traits to scale behaviors across types,
- map Rust trait-based abstraction to (and beyond) C++ interfaces, ABCs, CRTP, and concepts.

Static vs Dynamic Polymorphism

3.1 Generics and Monomorphization

Rust generics provide **static polymorphism**. For each concrete type used with a generic function or type, the compiler generates a specialized version of the code. This process is called **monomorphization**.
This is conceptually similar to C++ templates, but with stricter coherence rules and explicit trait bounds.

Basic Generic Example

```
trait Area {
    fn area(&self) -> f64;
}

struct Circle { r: f64 }
struct Square { s: f64 }

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

impl Area for Square {
    fn area(&self) -> f64 { self.s * self.s }
}

fn print_area<T: Area>(shape: &T) {
    println!("area={}", shape.area());
}

fn main() {
    let c = Circle { r: 2.0 };
    let s = Square { s: 3.0 };
    print_area(&c);
    print_area(&s);
}
```

The compiler generates specialized code for `Circle` and `Square`. There is no runtime dispatch overhead.

What Monomorphization Implies

- Each concrete type produces a specialized version of the generic function.
- Enables inlining and aggressive optimization.
- Increases binary size if many concrete types are used.
- Zero runtime dispatch cost.

3.2 When Static Dispatch Is Superior

Static dispatch (generics) is preferable when:

- Performance is critical.

- Call sites are performance-sensitive (tight loops).
- The set of types is known at compile time.
- You want maximum inlining and optimizer visibility.
- You do not need heterogeneous collections.

Performance-Critical Loop Example

```rust
trait Transform {
    fn apply(&self, x: f64) -> f64;
}

struct Scale { factor: f64 }

impl Transform for Scale {
    fn apply(&self, x: f64) -> f64 { x * self.factor }
}

fn process<T: Transform>(t: &T, data: &mut [f64]) {
    for v in data {
        *v = t.apply(*v);
    }
}

fn main() {
    let mut data = vec![1.0, 2.0, 3.0];
    let scale = Scale { factor: 2.0 };
    process(&scale, &mut data);
    println!("{:?}", data);
}
```

The compiler can inline `apply`, remove abstraction cost, and optimize aggressively.

3.3 Trait Objects and dyn

Dynamic polymorphism in Rust uses `dyn Trait`. This enables runtime dispatch through a vtable. Unlike generics, no code duplication occurs.

Basic Trait Object Example

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

struct Html;
struct Markdown;

impl Render for Html {
    fn render(&self) -> String { "<p>Hello</p>".into() }
}

impl Render for Markdown {
    fn render(&self) -> String { "**Hello**".into() }
}

fn render_all(items: &[Box<dyn Render>]) {
    for item in items {
        println!("{}", item.render());
    }
}

fn main() {
    let items: Vec<Box<dyn Render>> =
        vec![Box::new(Html), Box::new(Markdown)];
    render_all(&items);
}
```

Here:

- All elements share a common trait.
- Dispatch occurs at runtime via vtable.
- Enables heterogeneous collections.

3.4 Object Safety Rules

Not every trait can be turned into a trait object. A trait must be **object-safe**.
A trait is object-safe if:

- It does not require `Self: Sized`.
- Methods do not return `Self`.
- Methods do not have generic type parameters.

Non Object-Safe Example

```
trait CloneLike {
    fn clone_like(&self) -> Self; // not object-safe
}
```

This cannot be used as `dyn CloneLike` because the return type depends on concrete `Self`.

Object-Safe Alternative

```
trait CloneBox {
    fn clone_box(&self) -> Box<dyn CloneBox>;
}
```

Designing traits with object safety in mind is essential when dynamic dispatch is required.

3.5 Vtables in Rust vs C++

Both Rust trait objects and C++ virtual classes use vtables internally.

Common Properties

- A pointer to data.
- A pointer to a vtable containing function pointers.
- One indirect call per method dispatch.

Key Differences

- Rust does not allow implicit inheritance-based dispatch.
- Dynamic dispatch is explicit via dyn.
- Rust separates static and dynamic polymorphism cleanly.
- No accidental virtual dispatch; the choice is architectural.

3.6 Performance Tradeoffs

Static Dispatch (Generics)

- Zero runtime overhead.
- Maximum inlining and optimization.
- Larger binary size (code duplication).
- Compile-time cost increases with many instantiations.

Dynamic Dispatch (dyn)

- Single code version (smaller binary).
- One pointer indirection + one indirect function call.
- Reduced inlining.
- Enables heterogeneous collections and runtime flexibility.

Enum-Based Alternative

Sometimes neither generics nor trait objects are optimal. An `enum` can replace virtual dispatch entirely:

```
enum Shape {
    Circle(f64),
    Square(f64),
}

impl Shape {
    fn area(&self) -> f64 {
        match self {
            Shape::Circle(r) => std::f64::consts::PI * r * r,
            Shape::Square(s) => s * s,
        }
    }
}

fn main() {
    let shapes = vec![Shape::Circle(2.0), Shape::Square(3.0)];
    for s in shapes {
        println!("{}", s.area());
    }
}
```

- No vtable.
- Exhaustive matching.
- Excellent optimization potential.
- Less extensible than trait-based polymorphism.

Outcome

After this chapter, you can:

- Use generics for zero-cost static polymorphism.
- Recognize when monomorphization benefits outweigh binary size growth.
- Use `dyn Trait` for runtime polymorphism intentionally.
- Design object-safe traits correctly.

- Understand vtable behavior in Rust and how it differs architecturally from C++.
- Evaluate performance tradeoffs and choose the appropriate abstraction mechanism.

Part II

Replacing Inheritance Properly

Composition, Delegation, and the Newtype Pattern

4.1 Why Inheritance Is Removed

Rust intentionally omits *implementation inheritance* (a derived type inheriting data and method implementations from a base class). The architectural reasons are practical:

- **Tighter coupling:** inheritance binds representation, behavior, and evolution of types across layers.
- **Fragile base class issues:** changes in a base type can silently break derived types.
- **Complex initialization/teardown:** constructor/destructor ordering and virtual dispatch during construction create subtle hazards.
- **Ambiguity:** multiple inheritance introduces diamond conflicts and method resolution complexity.
- **Encapsulation erosion:** `protected` state invites hidden dependencies and invalid states.

Rust replaces these with explicit tools:

- **Traits** for shared behavior contracts (interfaces),

- **Composition** for reuse of state and implementation,
- **Delegation** (manual, via macros, or via helper types) to forward behavior,
- **Newtypes** to isolate and control semantics,
- **Enums** for closed-world polymorphism when appropriate.

4.2 Composition as First Principle

Composition means building a type out of other types and exposing only the intended surface. In Rust, composition is the normal way to reuse implementation and state without exposing internals.

Compose a Reusable Component

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

struct Metrics {
    counts: HashMap<String, u64>,
}

impl Metrics {
    fn new() -> Self { Self { counts: HashMap::new() } }

    fn inc(&mut self, key: &str) {
        *self.counts.entry(key.to_string()).or_insert(0) += 1;
    }

    fn get(&self, key: &str) -> u64 {
        self.counts.get(key).copied().unwrap_or(0)
    }
}

struct Service {
    name: String,
    metrics: Metrics, // composition: Service "has a" Metrics
}

impl Service {
    fn new(name: impl Into<String>) -> Self {
        Self { name: name.into(), metrics: Metrics::new() }
    }

    fn handle(&mut self, route: &str) {
        self.metrics.inc(route);
        println!("{} handled {}", self.name, route);
    }

    fn route_hits(&self, route: &str) -> u64 {
        self.metrics.get(route)
    }
}

fn main() {
    let mut s = Service::new("api");
    s.handle("/health");
    s.handle("/health");
    println!("hits={}", s.route_hits("/health"));
}
```

This is the default Rust posture: reuse implementation by holding it as a field, not by inheriting it.

4.3 Delegation Patterns

Delegation means forwarding calls to a composed field. Rust does not implicitly "inherit methods" from fields; you decide what to expose and how.

Manual Delegation (Explicit API)

```rust
use std::fs::File;
use std::io::{self, Write};

struct LogSink {
    file: File,
}

impl LogSink {
    fn create(path: &str) -> io::Result<Self> {
        Ok(Self { file: File::create(path)? })
    }

    fn write_line(&mut self, s: &str) -> io::Result<()> {
        writeln!(self.file, "{}", s)
    }
}

struct AppLogger {
    app: String,
    sink: LogSink,
}

impl AppLogger {
    fn create(app: impl Into<String>, path: &str) -> io::Result<Self> {
        Ok(Self { app: app.into(), sink: LogSink::create(path)? })
    }

    fn info(&mut self, msg: &str) -> io::Result<()> {
        self.sink.write_line(&format!("[{}][INFO] {}", self.app, msg))
    }

    fn error(&mut self, msg: &str) -> io::Result<()> {
        self.sink.write_line(&format!("[{}][ERROR] {}", self.app, msg))
    }
}

fn main() -> io::Result<()> {
    let mut lg = AppLogger::create("ForgeVM", r"C:\Temp\forgevm.log")?;
    lg.info("start")?;
    lg.error("something happened")?;
    Ok(())
}
```

Delegation via Trait + Forwarding

You can formalize forwarding by implementing a trait for a wrapper type and routing to the inner field. This keeps substitution at the trait boundary (interfaces), not through inheritance.

```
trait Store {
    fn put(&mut self, k: &str, v: &str);
    fn get(&self, k: &str) -> Option<&str>;
}

struct MemStore {
    items: std::collections::HashMap<String, String>,
}

impl MemStore {
    fn new() -> Self { Self { items: std::collections::HashMap::new() } }
}

impl Store for MemStore {
    fn put(&mut self, k: &str, v: &str) {
        self.items.insert(k.to_string(), v.to_string());
    }
    fn get(&self, k: &str) -> Option<&str> {
        self.items.get(k).map(|s| s.as_str())
    }
}

struct TracedStore<S> {
    inner: S,
}

impl<S: Store> TracedStore<S> {
    fn new(inner: S) -> Self { Self { inner } }
}

impl<S: Store> Store for TracedStore<S> {
    fn put(&mut self, k: &str, v: &str) {
        println!("[put] {}", k);
        self.inner.put(k, v);
    }
    fn get(&self, k: &str) -> Option<&str> {
        println!("[get] {}", k);
        self.inner.get(k)
    }
}

fn main() {
    let mut s = TracedStore::new(MemStore::new());
    s.put("lang", "Rust");
    println!("{:?}", s.get("lang"));
}
```

This is the Rust replacement for “derive from base store and override some virtuals”.

4.4 Newtype for Behavioral Isolation

A **newtype** is a tuple struct with one field, used to:

- prevent accidental mixing of similar representations (domain safety),
- attach new trait implementations without changing the original type,
- isolate behavior and control the API surface.

Domain-Specific Safety

```
#[derive(Copy, Clone, Debug, Eq, PartialEq, Hash)]
struct UserId(u64);

#[derive(Copy, Clone, Debug, Eq, PartialEq, Hash)]
struct SessionId(u64);

fn load_user(_id: UserId) {}
fn load_session(_id: SessionId) {}

fn main() {
    let u = UserId(10);
    let s = SessionId(10);

    load_user(u);
    load_session(s);
    // load_user(s); // ERROR: different types, no accidental mix
}
```

Behavioral Isolation: Add Capabilities Without Touching the Inner Type

```
use std::fmt;

struct Secret(String);

impl Secret {
    fn new(s: impl Into<String>) -> Self { Self(s.into()) }
    fn expose(&self) -> &str { &self.0 }
}

// Control formatting: do not leak the secret via Debug/Display.
impl fmt::Debug for Secret {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        f.write_str("Secret(**redacted**)")
    }
}

impl fmt::Display for Secret {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        f.write_str("**redacted**")
    }
}

fn main() {
    let x = Secret::new("top-token");
    println!("{:?}", x);
    println!("{}", x);
    println!("len={}", x.expose().len()); // explicit access only
}
```

In C++, this often becomes a base class with protected members or ad-hoc conventions. In Rust, the wrapper is the policy.

4.5 Avoiding Diamond Problems by Design

Rust avoids the diamond problem by removing implementation inheritance. The language steers you to designs where ambiguity is eliminated:

- Use **composition** for shared state and shared implementation.
- Use **traits** for shared behavior contracts, implemented independently.
- If you need a single combined behavior, define a **new trait** that composes requirements.

Trait Composition Without Ambiguity

```rust
trait Readable {
    fn read(&self) -> String;
}

trait Writable {
    fn write(&mut self, s: &str);
}

// A combined contract: no diamond ambiguity, just explicit requirements.
trait ReadWrite: Readable + Writable {}
impl<T: Readable + Writable> ReadWrite for T {}

struct Buffer {
    s: String,
}
```

```
impl Buffer {
    fn new() -> Self { Self { s: String::new() } }
}

impl Readable for Buffer {
    fn read(&self) -> String { self.s.clone() }
}

impl Writable for Buffer {
    fn write(&mut self, s: &str) { self.s.push_str(s); }
}

fn use_rw<T: ReadWrite>(x: &mut T) {
    x.write("hi");
    println!("{}", x.read());
}

fn main() {
    let mut b = Buffer::new();
    use_rw(&mut b);
}
```

If two traits expose same method names, Rust requires explicit disambiguation (*fully qualified syntax*) rather than silently choosing a base.

Explicit Disambiguation When Needed

```
trait A { fn f(&self) -> i32; }
trait B { fn f(&self) -> i32; }

struct X;

impl A for X { fn f(&self) -> i32 { 1 } }
impl B for X { fn f(&self) -> i32 { 2 } }

fn main() {
    let x = X;
    println!("{}", A::f(&x));
    println!("{}", B::f(&x));
}
```

No hidden method resolution order, no diamond surprises: ambiguity is handled explicitly.

4.6 Comparison with Modern C++

4.6.1 Multiple Inheritance

- **C++:** multiple inheritance combines interfaces and possibly shared implementation/state, risking ambiguity (diamond) and subtle base initialization behavior.
- **Rust:** no implementation inheritance; you combine behavior via traits and reuse state via composition, eliminating diamond conflicts by construction.

4.6.2 Mixins

- **C++:** mixins (often templates/CRTP) inject behavior into a type.
- **Rust:** blanket implementations and extension traits provide “mixin-like” behavior without inheriting state; wrapper types (newtypes) provide policy injection with explicit composition.

4.6.3 Protected Members

- **C++:** `protected` is frequently used for reuse but weakens encapsulation (derived types become dependent on internal representation).

- **Rust:** module privacy + private fields encourage exposing *operations* not *state*. Shared internals are expressed by composing a helper type or by placing cooperating types in the same module with controlled visibility.

4.7 Outcome

After this chapter, you can:

- replace inheritance trees with composition graphs,
- reuse implementation by embedding components and delegating explicitly,
- isolate behavior and enforce policy using newtypes,
- avoid diamond ambiguity by using trait composition and explicit disambiguation,
- design Rust APIs that remain stable and maintainable as systems evolve.

Enums and Algebraic Modeling

5.1 Sum Types as Superior Hierarchies

Rust `enum` is not merely an "integer enum" as in C/C++; it is an **sum type** (algebraic data type) where each variant can carry typed data. This makes `enum` a strong replacement for many class hierarchies that exist only to represent *a closed set of alternatives*.
Use `enum` when:

- the set of cases is known and intentionally closed,
- behavior depends on which case is present,
- you want the compiler to enforce handling of every case,
- you want to avoid downcasts, RTTI, and fragile virtual hierarchies.

A Closed Hierarchy in One Type

```
#[derive(Debug, Clone)]
enum Expr {
    Num(i64),
    Add(Box<Expr>, Box<Expr>),
    Mul(Box<Expr>, Box<Expr>),
}
```

```
impl Expr {
    fn eval(&self) -> i64 {
        match self {
            Expr::Num(n) => *n,
            Expr::Add(a, b) => a.eval() + b.eval(),
            Expr::Mul(a, b) => a.eval() * b.eval(),
        }
    }
}

fn main() {
    let e = Expr::Add(
        Box::new(Expr::Num(2)),
        Box::new(Expr::Mul(Box::new(Expr::Num(3)), Box::new(Expr::Num(4)))),
    );
    println!("{:?} = {}", e, e.eval()); // 2 + (3*4) = 14
}
```

This replaces a typical `Expr` base class with `Add`/`Mul`/`Num` derived classes.

5.2 Exhaustive Pattern Matching

Rust `match` is exhaustive: if you add a new variant, the compiler forces you to update all matching sites (unless you used a wildcard arm). This is a correctness tool, not just syntax.

Exhaustiveness as a Maintenance Guarantee

```
#[derive(Debug)]
enum Status {
    Ok,
    NotFound,
    Forbidden,
}

fn to_code(s: Status) -> u16 {
    match s {
        Status::Ok => 200,
        Status::NotFound => 404,
        Status::Forbidden => 403,
    }
}

fn main() {
    println!("{}", to_code(Status::Ok));
}
```

If you later add `Status::Timeout`, every match without a wildcard becomes a compile error until updated. This is a structural advantage over virtual hierarchies where missing overrides can remain latent until runtime.

Destructuring: Pattern Matching Extracts Data Safely

```
#[derive(Debug)]
enum Message {
    Ping,
    Text { from: String, body: String },
    Move { x: i32, y: i32 },
}

fn handle(m: Message) {
    match m {
        Message::Ping => println!("ping"),
        Message::Text { from, body } => println!("from={} body={}", from, body),
        Message::Move { x, y } => println!("move to ({}, {})", x, y),
```

```rust
    }
}

fn main() {
    handle(Message::Text { from: "Ayman".to_string(), body: "Hello".to_string() });
}
```

No downcasting, no null checks, no unsafe reinterpretation: the data shape is enforced by the type system.

5.3 State Machines in Rust

A major domain where `enum` outperforms classical OOP is state machines. Instead of a base class with derived “state” classes and virtual transitions, Rust models the states explicitly and makes illegal states hard to represent.

Explicit State Machine with Owned State

```rust
#[derive(Debug)]
enum ConnState {
    Disconnected,
    Connecting { attempts: u32 },
    Connected { peer: String },
}

struct Connection {
    st: ConnState,
}

impl Connection {
    fn new() -> Self {
        Self { st: ConnState::Disconnected }
    }

    fn connect(&mut self, peer: &str) {
        self.st = ConnState::Connecting { attempts: 1 };
        // simplified: assume success
        self.st = ConnState::Connected { peer: peer.to_string() };
    }

    fn send(&self, payload: &str) {
        match &self.st {
            ConnState::Connected { peer } => {
                println!("send to {}: {}", peer, payload);
            }
            ConnState::Disconnected => {
                println!("cannot send: disconnected");
            }
            ConnState::Connecting { attempts } => {
                println!("cannot send: connecting (attempts={})", attempts);
            }
        }
    }

    fn disconnect(&mut self) {
        self.st = ConnState::Disconnected;
    }
}

fn main() {
    let mut c = Connection::new();
    c.send("hi");
    c.connect("127.0.0.1:9000");
    c.send("hello");
    c.disconnect();
    c.send("bye");
}
```

This style is straightforward and forces every operation to consider all states.

Stronger State Machines: Type-State (Compile-Time Transitions)

When invalid operations must not even compile, combine enums with the type-state pattern (introduced earlier). Enums handle *runtime state*, type-state handles *protocol guarantees* at compile time.

5.4 Replacing Virtual Hierarchies with Enums

Many virtual hierarchies exist for one of two reasons:

- **closed alternatives** (a finite set of node kinds, messages, commands),
- **open extension** (third parties add new types later).

Enums excel at closed alternatives. Traits (especially `dyn Trait`) excel at open extension. A useful rule:

- **Closed set** → `enum` + `match`.
- **Open set** → traits (`T: Trait` or `dyn Trait`).

Command Hierarchy as Enum

```
#[derive(Debug)]
enum Command {
    CreateUser { name: String },
    DeleteUser { id: u64 },
    ResetPassword { id: u64 },
}

fn execute(cmd: Command) {
    match cmd {
        Command::CreateUser { name } => println!("create user {}", name),
        Command::DeleteUser { id } => println!("delete user {}", id),
        Command::ResetPassword { id } => println!("reset password {}", id),
    }
}

fn main() {
    execute(Command::CreateUser { name: "Ayman".to_string() });
}
```

In C++ this is often a base `Command` with derived types and a virtual `execute`. The Rust enum keeps the family together and eliminates allocation and vtable dispatch in many cases.

Visitor-Style Logic Becomes Plain `match`

In C++ you may use Visitor to avoid RTTI. In Rust, `match` is the built-in visitor.

```
#[derive(Debug)]
enum Node {
    Int(i64),
    Str(String),
    Pair(Box<Node>, Box<Node>),
}

fn pretty(n: &Node) -> String {
    match n {
        Node::Int(x) => x.to_string(),
        Node::Str(s) => format!("{:?}", s),
        Node::Pair(a, b) => format!("({}, {})", pretty(a), pretty(b)),
    }
}

fn main() {
    let n = Node::Pair(Box::new(Node::Int(10)), Box::new(Node::Str("hi".to_string())));
    println!("{}", pretty(&n));
}
```

5.5 Comparison with `std::variant` and Exception Polymorphism

5.5.1 Rust enum vs `std::variant`

- **Model:** Rust enums are algebraic sum types with ergonomic pattern matching; `std::variant` is a tagged union requiring `std::visit`.
- **Exhaustiveness:** Rust `match` can be fully exhaustive by construction; `std::visit` exhaustiveness depends on visitor completeness and tooling.
- **Ergonomics:** Rust destructuring patterns are concise and deeply integrated; `std::variant` is powerful but typically more verbose.
- **Design pressure:** Rust encourages enum-first modeling for closed sets; C++ often defaults to inheritance and virtuals, then introduces `variant` later as a refactor.

5.5.2 Exception Polymorphism vs Result/Enum Modeling

Some C++ designs rely on exceptions for “polymorphic failure” (different derived exception types). Rust encourages explicit failure modeling:

- **C++:** throw derived exceptions; handle via catch hierarchy.
- **Rust:** return `Result<T, E>` where `E` is often an `enum` of well-defined failure cases.

Error Hierarchy as Enum

```rust
#[derive(Debug)]
enum AuthError {
    MissingToken,
    InvalidToken,
    Expired,
    Forbidden,
}

fn check(token: Option<&str>) -> Result<(), AuthError> {
    let t = token.ok_or(AuthError::MissingToken)?;
    if t.len() < 8 { return Err(AuthError::InvalidToken); }
    if t == "expired000" { return Err(AuthError::Expired); }
    if t == "forbiddenX" { return Err(AuthError::Forbidden); }
    Ok(())
}

fn main() {
    for tok in [None, Some("short"), Some("expired000"), Some("good_token_123")] {
        let r = check(tok);
        match r {
            Ok(()) => println!("ok"),
            Err(e) => println!("err={:?}", e),
        }
    }
}
```

This is a closed, explicit failure model with exhaustive handling, rather than a runtime exception hierarchy.

5.6 Outcome

After this chapter, you can:

- identify class hierarchies that are actually closed sets and replace them with enums,
- use exhaustive `match` to make future changes safe and compiler-enforced,
- model robust runtime state machines with enums and transitions,
- replace many virtual-dispatch “visitor” designs with direct pattern matching,
- choose enums for closed families and traits for open extension,
- replace exception hierarchies with explicit `Result<T, E>` enum error models.

Part III

Ownership-Driven Object Design

Ownership and Lifetimes in Object APIs

6.1 Ownership as Architectural Constraint

In Rust, ownership is not a memory-management trick; it is an **architectural constraint** that shapes every public API. A well-designed object API makes it obvious:

- who owns a value,
- who may mutate it,
- how long references remain valid,
- whether sharing is allowed and under what synchronization policy.

This turns many "discipline-based" C++ rules into compile-time guarantees: no use-after-free, no double-free, and no data races from unsynchronized shared mutable state.

Ownership Transfer is Explicit

When a method takes `self` (by value), the object is consumed and cannot be used again. This is a powerful pattern for enforcing protocol steps.

```rust
#[derive(Debug)]
struct FileName(String);
```

```
impl FileName {
    fn new(s: impl Into<String>) -> Self { Self(s.into()) }

    // Consuming method: ensures the name cannot be reused accidentally after
    ↪  "finalization".
    fn into_windows_path(self) -> String {
        format!(r"C:\Temp\{}", self.0)
    }
}

fn main() {
    let n = FileName::new("log.txt");
    let p = n.into_windows_path();
    println!("{}", p);
    // println!("{:?}", n); // ERROR: use of moved value
}
```

6.2 Borrowing in Method Signatures

Borrowing communicates intent and constraints precisely. The main categories are:

- `&T`: shared borrow (read-only, multiple aliases allowed),
- `&mut T`: exclusive mutable borrow (single writer, no aliases),
- returning `&T`: the caller receives a view tied to the callee's lifetime.

Borrowing Return Values Safely

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

```rust
struct Cache {
    items: HashMap<String, String>,
}

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

    fn insert(&mut self, k: &str, v: &str) {
        self.items.insert(k.to_string(), v.to_string());
    }

    fn get(&self, k: &str) -> Option<&str> {
        self.items.get(k).map(|s| s.as_str())
    }
}

fn main() {
    let mut c = Cache::new();
    c.insert("lang", "Rust");
    if let Some(v) = c.get("lang") {
        println!("{}", v);
    }
}
```

The returned reference cannot outlive `c`. This removes an entire class of dangling-pointer bugs common in manual designs.

Accept Borrowed Inputs to Avoid Unnecessary Allocation

```rust
struct Greeter;

impl Greeter {
    fn greet(&self, name: &str) -> String {
        format!("Hello, {}", name)
    }
}

fn main() {
    let g = Greeter;
    let s = String::from("Ayman");
    println!("{}", g.greet(&s));      // borrow String as &str
    println!("{}", g.greet("World")); // borrow string literal as &str
}
```

This is "zero-cost" at the call boundary: `&str` is a lightweight view.

6.3 `&self` vs `&mut self`

Receiver choice is the core OOP design lever in Rust.

- `&self`: a read-only method; allows multiple callers to hold references simultaneously.
- `&mut self`: a mutating method; requires exclusive access, preventing aliasing-related bugs.

Exclusive Mutation Prevents Aliasing Bugs

```
struct Buffer {
    data: Vec<u8>,
}

impl Buffer {
    fn new() -> Self { Self { data: Vec::new() } }

    fn len(&self) -> usize { self.data.len() }         // &self
    fn push(&mut self, b: u8) { self.data.push(b); }   // &mut self

    fn as_slice(&self) -> &[u8] { &self.data }         // returns shared borrow
}

fn main() {
    let mut b = Buffer::new();
    b.push(1);
    b.push(2);

    let view = b.as_slice();
    println!("len={} first={}", b.len(), view[0]);

    // b.push(3); // ERROR: cannot borrow `b` as mutable because it is also borrowed as
    ↪  immutable
}
```

This compile-time rejection is the point: it prevents holding a reference into a buffer while mutating the buffer in a way that could reallocate and invalidate the reference.

API Design: Prefer Non-Mutating Methods Where Possible

Mutability is not "forbidden" in Rust; it is *controlled*. An API that uses `&self` where appropriate becomes easier to compose and reason about.

6.4 Designing Zero-Cost Safe APIs

"Zero-cost" in this context means:

- no hidden allocations to satisfy the API,
- no reference counting unless explicitly requested,

- no runtime borrow checks unless you opt into interior mutability,
- no runtime polymorphism unless you choose dyn.

A typical strategy:

- accept borrowed views (`&str`, `&[T]`) at boundaries,
- return borrowed views when lifetime allows,
- use owned types only when you must store data beyond the call,
- avoid exposing raw pointers; expose safe references/slices instead.

Boundary Pattern: Borrow In, Own Internally (When Necessary)

```rust
use std::collections::HashSet;

struct Registry {
    names: HashSet<String>, // owns storage
}

impl Registry {
    fn new() -> Self { Self { names: HashSet::new() } }

    // Borrow input to allow callers to pass &str or String with no extra burden.
    fn register(&mut self, name: &str) -> bool {
        self.names.insert(name.to_string())
    }

    // Return a borrowed view: caller cannot outlive self.
    fn contains(&self, name: &str) -> bool {
        self.names.contains(name)
    }
}

fn main() {
    let mut r = Registry::new();
    let n = String::from("ForgeVM");
    println!("{}", r.register(&n));
    println!("{}", r.contains("ForgeVM"));
}
```

Lifetimes as an API Guarantee (No Runtime Cost)

When returning references, lifetimes express validity. You rarely need to write explicit lifetime parameters; Rust infers them. When you do need them, they define contracts precisely.

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

fn main() {
    let x = String::from("abcd");
    let y = String::from("xyz");
    let r = longest(&x, &y);
    println!("{}", r);
}
```

The returned reference is guaranteed to be valid as long as both inputs are valid.

6.5 Comparison with C++ Pointer Models

6.5.1 `unique_ptr` vs Rust Ownership

- **C++:** `std::unique_ptr<T>` encodes unique ownership at runtime with move-only semantics.

- **Rust:** unique ownership is the default for *all values*. Moves are implicit and checked everywhere, not only for heap pointers.
- **Implication:** APIs naturally express ownership transfer via taking `T` or `self`.

6.5.2 `shared_ptr` vs Rust Shared Ownership

- **C++:** `std::shared_ptr<T>` enables shared ownership via reference counting; aliasing is easy, mutation hazards remain.
- **Rust:** shared ownership is explicit via `Rc<T>` (single-thread) or `Arc<T>` (multi-thread). Mutable sharing typically requires `RefCell`/`Mutex`/`RwLock`.
- **Implication:** if you want shared mutation, you must choose the synchronization/borrow policy explicitly.

6.5.3 Raw Pointers

- **C++:** raw pointers are ubiquitous for performance and interoperability, but correctness depends on discipline.
- **Rust:** raw pointers (`*const T`, `*mut T`) exist for FFI and low-level work, but most safe APIs expose `&T`, `&mut T`, `&[T]`, and iterators instead.
- **Implication:** safe references carry borrowing rules; raw pointers push responsibility into `unsafe` blocks.

C++-Style “Borrowed Pointer” vs Rust Borrowed Reference

A typical C++ pattern returns `T*` or `T&` with informal lifetime rules. Rust makes those lifetime rules part of the type system.

```rust
struct Table {
    rows: Vec<String>,
}

impl Table {
    fn new() -> Self { Self { rows: vec![] } }

    fn push(&mut self, s: &str) { self.rows.push(s.to_string()); }

    fn row(&self, i: usize) -> Option<&str> {
        self.rows.get(i).map(|s| s.as_str())
    }
}

fn main() {
    let mut t = Table::new();
    t.push("r0");
    if let Some(r) = t.row(0) {
        println!("{}", r);
    }
}
```

6.6 Outcome

After this chapter, you can:

- treat ownership as a design constraint and express it in public APIs,
- choose borrowing signatures that communicate and enforce correct usage,
- use `&self` and `&mut self` deliberately to control aliasing and mutation,
- build zero-cost safe boundaries using borrowed views and owned storage appropriately,

- map Rust ownership to C++ `unique_ptr`/`shared_ptr`/raw pointers and avoid their typical misuse patterns,
- design APIs that prevent misuse by construction instead of relying on conventions.

Interior Mutability and Shared Ownership

7.1 Motivation: When “&mut” Is Too Strong

Rust’s default rule—**many readers or one writer**—prevents data races and aliasing bugs. Interior mutability and shared ownership are the explicit escape hatches for designs that need mutation behind a shared reference, caching, reference-counted graphs, or cross-thread sharing. These tools are safe when used with clear invariants and narrow scopes.

7.2 `Cell` and `RefCell`

7.2.1 `Cell<T>`: Copy-Style Interior Mutability

`Cell<T>` enables mutation through `&self` for small `Copy` data. It does not hand out references to the inner value; it moves values in/out, preventing aliasing of interior data.

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

struct Stats {
    hits: Cell<u64>,
}

impl Stats {
    fn new() -> Self { Self { hits: Cell::new(0) } }

    fn hit(&self) {
        let v = self.hits.get();
        self.hits.set(v + 1);
    }

    fn hits(&self) -> u64 { self.hits.get() }
}

fn main() {
    let s = Stats::new();
    s.hit();
    s.hit();
    println!("hits={}", s.hits());
}
```

Use `Cell` for counters, flags, and small numeric state where borrowing references is unnecessary.

7.2.2 `RefCell<T>`: Borrow-Checked at Runtime (Single-Thread)

`RefCell<T>` allows borrowing `&T` and `&mut T` at runtime, enforcing the same borrowing rules dynamically. Violations cause a panic. This is ideal for single-threaded graphs and caches where compile-time borrowing is too restrictive.

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

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

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

    fn put(&self, k: &str, v: &str) {
        self.map.borrow_mut().insert(k.to_string(), v.to_string());
    }

    fn get(&self, k: &str) -> Option<String> {
        self.map.borrow().get(k).cloned()
    }
}

fn main() {
    let c = Cache::new();
    c.put("lang", "Rust");
    println!("{:?}", c.get("lang"));
}
```

Key rule: `RefCell` is for **single-threaded** interior mutability; it is not `Sync` by default.

7.3 Rc and Arc

7.3.1 Rc<T>: Shared Ownership (Single-Thread)

`Rc<T>` is reference-counted shared ownership for single-threaded code. It is common in tree/graph structures.

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

#[derive(Debug)]
struct Node {
    name: String,
    child: Option<Rc<Node>>,
}

fn main() {
    let leaf = Rc::new(Node { name: "leaf".to_string(), child: None });
    let root = Rc::new(Node { name: "root".to_string(), child: Some(Rc::clone(&leaf)) });

    println!("root={:?}", root.name);
    println!("leaf strong={}", Rc::strong_count(&leaf));
    let _another = Rc::clone(&leaf);
    println!("leaf strong={}", Rc::strong_count(&leaf));
}
```

Important: `Rc` does not provide thread safety. For cross-thread sharing use `Arc`.

7.3.2 `Arc<T>`: Shared Ownership (Multi-Thread)

`Arc<T>` is atomic reference counting and can be shared across threads when `T: Send + Sync`.

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

fn main() {
    let msg = Arc::new(String::from("hello"));
    let t1 = {
        let m = Arc::clone(&msg);
        thread::spawn(move || println!("t1 {}", m))
    };
    let t2 = {
        let m = Arc::clone(&msg);
        thread::spawn(move || println!("t2 {}", m))
    };
    t1.join().unwrap();
    t2.join().unwrap();
}
```

7.4 Mutex and RwLock

Shared ownership does not imply shared mutability. In Rust, shared mutation across threads must be protected by synchronization primitives, typically with `Arc<Mutex<T»` or `Arc<RwLock<T»`.

7.4.1 Mutex<T>: Exclusive Lock

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

fn main() {
    let counter = Arc::new(Mutex::new(0u64));

    let mut threads = Vec::new();
    for _ in 0..4 {
        let c = Arc::clone(&counter);
        threads.push(thread::spawn(move || {
            for _ in 0..10_000 {
                let mut g = c.lock().unwrap();
                *g += 1;
            }
        }));
    }

    for t in threads { t.join().unwrap(); }
    println!("counter={}", *counter.lock().unwrap());
}
```

The lock guard enforces RAII: the mutex is unlocked when the guard is dropped.

7.4.2 RwLock<T>: Many Readers or One Writer

`RwLock` is effective when reads dominate writes.

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

fn main() {
    let data = Arc::new(RwLock::new(vec![1, 2, 3]));

    let r1 = {
        let d = Arc::clone(&data);
        thread::spawn(move || {
            let g = d.read().unwrap();
            println!("r1 len={}", g.len());
        })
    };

    let w = {
        let d = Arc::clone(&data);
        thread::spawn(move || {
            let mut g = d.write().unwrap();
            g.push(4);
        })
    };

    r1.join().unwrap();
    w.join().unwrap();

    let g = data.read().unwrap();
    println!("final={:?}", *g);
}
```

7.5 Tradeoffs and Costs

7.5.1 What You Pay For

- **`Cell`:** minimal overhead; limited to patterns that don't require references to interior data.
- **`RefCell`:** runtime borrow checks; panics on violation; single-thread oriented.
- **`Rc`:** reference counting overhead; not thread-safe; can leak via cycles.
- **`Arc`:** atomic refcount overhead; thread-safe; still can leak via cycles.
- **`Mutex`/`RwLock`:** contention and potential deadlock if misused; lock acquisition costs; poisoning behavior on panic.

7.5.2 Avoiding `Rc`/`Arc` Cycles

Reference cycles can cause memory leaks because refcounts never reach zero. Use `Weak` for back-references.

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

#[derive(Debug)]
struct Parent {
    child: RefCell<Option<Rc<Child>>>,
}

#[derive(Debug)]
struct Child {
    parent: RefCell<Weak<Parent>>,
}
```

```rust
fn main() {
    let p = Rc::new(Parent { child: RefCell::new(None) });
    let c = Rc::new(Child { parent: RefCell::new(Weak::new()) });

    *p.child.borrow_mut() = Some(Rc::clone(&c));
    *c.parent.borrow_mut() = Rc::downgrade(&p);

    println!("parent strong={}", Rc::strong_count(&p));
    println!("child strong={}", Rc::strong_count(&c));
}
```

7.6 Designing Thread-Safe Objects

A practical, idiomatic pattern is "**immutable interface + synchronized interior**":

- expose behavior via `&self` methods,
- store mutable state behind `Mutex`/`RwLock`,
- share the object via `Arc`.

7.6.1 Thread-Safe Service Object with `Arc<RwLock<T»`

```rust
use std::collections::HashMap;
use std::sync::{Arc, RwLock};
use std::thread;

struct ConfigStore {
    inner: RwLock<HashMap<String, String>>,
}

impl ConfigStore {
    fn new() -> Self { Self { inner: RwLock::new(HashMap::new()) } }
    fn set(&self, k: &str, v: &str) {
        let mut g = self.inner.write().unwrap();
        g.insert(k.to_string(), v.to_string());
    }

    fn get(&self, k: &str) -> Option<String> {
        let g = self.inner.read().unwrap();
        g.get(k).cloned()
    }
}

fn main() {
    let store = Arc::new(ConfigStore::new());
    store.set("mode", "dev");

    let t1 = {
        let s = Arc::clone(&store);
        thread::spawn(move || {
            for _ in 0..3 {
                println!("t1 mode={:?}", s.get("mode"));
            }
        })
    };

    let t2 = {
        let s = Arc::clone(&store);
        thread::spawn(move || {
            s.set("mode", "prod");
            println!("t2 updated");
        })
    };
    t1.join().unwrap();
    t2.join().unwrap();
    println!("main mode={:?}", store.get("mode"));
}
```

This is Windows-friendly: standard library threads and locks behave consistently across platforms.

7.7 Comparison with C++ Data Races

In C++, it is easy to accidentally create data races by sharing mutable objects across threads via raw pointers or `shared_ptr` without synchronization. The type system typically cannot enforce safe sharing; correctness depends on discipline and review.
Rust changes the default:

- A value is **not** shareable across threads unless it satisfies `Send` (move to another thread) and `Sync` (shared references across threads).
- Shared mutable state across threads generally requires **explicit synchronization** (`Mutex`/`RwLock`) and typically `Arc`.
- Interior mutability in single-threaded code is explicit (`Cell`/`RefCell`), and its costs and failure modes are visible.

The Core Design Takeaway

- Use `Cell` for tiny `Copy` state.
- Use `RefCell` for single-threaded interior mutability with strict invariants.
- Use `Rc` for single-threaded shared graphs; use `Weak` to break cycles.
- Use `Arc` for cross-thread sharing; combine with `Mutex`/`RwLock` for shared mutation.

7.8 Outcome

After this chapter, you can:

- identify when interior mutability is justified and choose `Cell` vs `RefCell`,
- choose `Rc` vs `Arc` based on threading requirements,
- design shared mutable state safely using `Mutex` or `RwLock`,
- reason about costs: runtime borrow checks, atomic refcounts, and lock contention,
- prevent common pitfalls such as reference cycles using `Weak`,
- build thread-safe object APIs that are explicit, disciplined, and portable on Windows.

Part IV

Real-World Architectural Patterns

Design Patterns Rewritten in Rust

8.1 Strategy (Generic vs dyn)

In Rust, Strategy is usually a choice between:

- **Generic strategy** (`T: Trait`) for static dispatch (zero runtime dispatch cost),
- **Trait object** (`dyn Trait`) for runtime selection and heterogeneous storage.

8.1.1 Generic Strategy (Static Dispatch)

```rust
trait Compress {
    fn compress(&self, input: &[u8]) -> Vec<u8>;
}

struct Identity;
impl Compress for Identity {
    fn compress(&self, input: &[u8]) -> Vec<u8> { input.to_vec() }
}

struct Rle;
impl Compress for Rle {
    fn compress(&self, input: &[u8]) -> Vec<u8> {
        if input.is_empty() { return vec![]; }
        let mut out = Vec::new();
        let mut cur = input[0];
        let mut cnt: u8 = 1;
        for &b in &input[1..] {
            if b == cur && cnt < u8::MAX {
                cnt += 1;
            } else {
                out.push(cur);
                out.push(cnt);
                cur = b;
                cnt = 1;
            }
        }
        out.push(cur);
        out.push(cnt);
        out
    }
}

struct Pipeline<C: Compress> {
    c: C,
}

impl<C: Compress> Pipeline<C> {
    fn new(c: C) -> Self { Self { c } }
    fn run(&self, data: &[u8]) -> Vec<u8> { self.c.compress(data) }
}

fn main() {
    let p1 = Pipeline::new(Identity);
    let p2 = Pipeline::new(Rle);

    let data = b"aaabbbcccc";
    println!("{:?}", p1.run(data));
    println!("{:?}", p2.run(data));
}
```

Use this when the strategy type is known at compile time and performance/inlining matters.

8.1.2 Trait Object Strategy (Dynamic Dispatch)

```
trait HashAlgo {
    fn hash(&self, s: &str) -> u64;
}

struct Fnv1a;
impl HashAlgo for Fnv1a {
    fn hash(&self, s: &str) -> u64 {
        let mut h: u64 = 14695981039346656037;
        for b in s.as_bytes() {
            h ^= *b as u64;
            h = h.wrapping_mul(1099511628211);
        }
        h
    }
}

struct Sum;
impl HashAlgo for Sum {
    fn hash(&self, s: &str) -> u64 {
        s.as_bytes().iter().fold(0u64, |acc, &b| acc + b as u64)
    }
}

struct Hasher {
    algo: Box<dyn HashAlgo>,
```

```rust
}

impl Hasher {
    fn new(algo: Box<dyn HashAlgo>) -> Self { Self { algo } }
    fn run(&self, s: &str) -> u64 { self.algo.hash(s) }
}

fn main() {
    let h = Hasher::new(Box::new(Fnv1a));
    println!("{}", h.run("ForgeVM"));

    let h = Hasher::new(Box::new(Sum));
    println!("{}", h.run("ForgeVM"));
}
```

Use this when you need runtime selection, plugin-like behavior, or heterogeneous collections.

8.2 Factory

Rust factories are typically associated functions returning concrete types, enums, or trait objects. There is no need for inheritance-based factories; construction policies are explicit.

8.2.1 Factory Returning an `enum` (Closed Set)

```rust
enum Transport {
    Tcp { addr: String },
    Udp { addr: String },
}

impl Transport {
    fn connect(&self) {
        match self {
            Transport::Tcp { addr } => println!("tcp connect {}", addr),
            Transport::Udp { addr } => println!("udp connect {}", addr),
        }
    }
}

fn make_transport(kind: &str, addr: &str) -> Transport {
    match kind {
        "tcp" => Transport::Tcp { addr: addr.to_string() },
        _ => Transport::Udp { addr: addr.to_string() },
    }
}

fn main() {
    let t = make_transport("tcp", "127.0.0.1:9000");
    t.connect();
}
```

8.2.2 Factory Returning a Trait Object (Open Set)

```
trait Sink: Send + Sync {
    fn write(&self, s: &str);
}

struct StdoutSink;
impl Sink for StdoutSink {
    fn write(&self, s: &str) { println!("{}", s); }
}

struct FileSink {
    path: String,
    // simplified: in real systems you may keep a handle behind a Mutex and do buffering
}
```

```
impl Sink for FileSink {
    fn write(&self, s: &str) { println!("file({}): {}", self.path, s); }
}

fn make_sink(kind: &str) -> Box<dyn Sink> {
    match kind {
        "stdout" => Box::new(StdoutSink),
        _ => Box::new(FileSink { path: r"C:\Temp\out.log".to_string() }),
    }
}

fn main() {
    let s = make_sink("stdout");
    s.write("hello");
}
```

8.3 Observer

Observer in Rust is usually implemented with:

- a list of subscribers as closures (`Fn`/`FnMut`),
- or a trait-based subscriber interface,
- plus explicit ownership/sharing (`Rc`/`RefCell` single-thread, `Arc`/`Mutex` multi-thread).

8.3.1 Single-Thread Observer with Callbacks

```
struct EventBus {
    subs: Vec<Box<dyn Fn(&str)>>,
}

impl EventBus {
    fn new() -> Self { Self { subs: Vec::new() } }

    fn subscribe<F>(&mut self, f: F)
    where
        F: Fn(&str) + 'static,
    {
        self.subs.push(Box::new(f));
    }

    fn publish(&self, msg: &str) {
        for s in &self.subs {
            s(msg);
        }
    }
}

fn main() {
    let mut bus = EventBus::new();
    bus.subscribe(|m| println!("A got {}", m));
    bus.subscribe(|m| println!("B got {}", m));
    bus.publish("tick");
}
```

8.3.2 Thread-Safe Observer with `Arc<Mutex<...»`

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

type Callback = Box<dyn Fn(&str) + Send + Sync>;

struct SharedBus {
    subs: Mutex<Vec<Callback>>,
}
```

```rust
impl SharedBus {
    fn new() -> Self { Self { subs: Mutex::new(Vec::new()) } }

    fn subscribe(&self, cb: Callback) {
        self.subs.lock().unwrap().push(cb);
    }

    fn publish(&self, msg: &str) {
        let subs = self.subs.lock().unwrap();
        for s in subs.iter() {
            s(msg);
        }
    }
}

fn main() {
    let bus = Arc::new(SharedBus::new());
    bus.subscribe(Box::new(|m| println!("sub1 {}", m)));

    let b2 = Arc::clone(&bus);
    let t = thread::spawn(move || {
        b2.subscribe(Box::new(|m| println!("sub2 {}", m)));
        b2.publish("from thread");
    });

    t.join().unwrap();
    bus.publish("from main");
}
```

8.4 Command

Command becomes especially clean in Rust: represent commands as `enum` variants (closed set) and execute with `match`. This avoids deep class hierarchies and visitor boilerplate.

8.4.1 Command as `enum`

```
#[derive(Debug)]
enum Command {
    CreateUser { name: String },
    DeleteUser { id: u64 },
    ResetPassword { id: u64 },
}

struct App {
    next_id: u64,
}

impl App {
    fn new() -> Self { Self { next_id: 1 } }

    fn execute(&mut self, cmd: Command) {
        match cmd {
            Command::CreateUser { name } => {
                let id = self.next_id;
                self.next_id += 1;
                println!("create user id={} name={}", id, name);
            }
            Command::DeleteUser { id } => {
                println!("delete user id={}", id);
            }
            Command::ResetPassword { id } => {
                println!("reset password id={}", id);
            }
        }
    }
}

fn main() {
    let mut app = App::new();
    app.execute(Command::CreateUser { name: "Ayman".to_string() });
    app.execute(Command::ResetPassword { id: 1 });
}
```

8.5 State Pattern via Enums

The classic GoF State pattern uses a state base class and derived state objects. In Rust, state is often best modeled as an enum with explicit transitions.

8.5.1 Enum State Machine

```
#[derive(Debug)]
enum Door {
    Open,
    Closed,
    Locked { failed: u32 },
}

impl Door {
    fn open(self) -> Self {
        match self {
            Door::Closed => Door::Open,
            Door::Locked { failed } => {
                println!("cannot open: locked (failed={})", failed);
                Door::Locked { failed }
            }
            Door::Open => Door::Open,
        }
    }

    fn close(self) -> Self {
        match self {
            Door::Open => Door::Closed,
            x => x,
        }
    }

    fn lock(self) -> Self {
        match self {
            Door::Closed => Door::Locked { failed: 0 },
            x => x,
        }
    }

    fn unlock(self, key_ok: bool) -> Self {
        match self {
            Door::Locked { failed } => {
                if key_ok { Door::Closed } else { Door::Locked { failed: failed + 1 } }
            }
            x => x,
        }
    }
}

fn main() {
    let d = Door::Closed;
    let d = d.lock();
    let d = d.open();
    let d = d.unlock(false);
    let d = d.unlock(true);
    let d = d.open();
    println!("{:?}", d);
}
```

This eliminates dynamic dispatch and forces all transitions to be explicit and reviewable.

8.6 Dependency Injection via Traits

DI in Rust is typically *trait-based*:

- accept dependencies as generic parameters (`T: Trait`) for static dispatch,
- or accept `Arc<dyn Trait>` for runtime polymorphism and shared services.

Because ownership is explicit, DI naturally expresses lifetimes and sharing policies.

8.6.1 DI with Generics (Compile-Time)

```
trait Repo {
    fn save(&mut self, name: &str) -> u64;
}

struct MemRepo {
    next: u64,
}

impl MemRepo {
    fn new() -> Self { Self { next: 1 } }
}
```

```
impl Repo for MemRepo {
    fn save(&mut self, name: &str) -> u64 {
        let id = self.next;
        self.next += 1;
        println!("saved {} => {}", name, id);
        id
    }
}

struct Service<R: Repo> {
    repo: R,
}

impl<R: Repo> Service<R> {
    fn new(repo: R) -> Self { Self { repo } }
    fn create_user(&mut self, name: &str) -> u64 { self.repo.save(name) }
}

fn main() {
    let repo = MemRepo::new();
    let mut svc = Service::new(repo);
    svc.create_user("Ayman");
}
```

8.6.2 DI with `Arc<dyn Trait>` (Runtime + Sharing)

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

trait Clock: Send + Sync {
    fn now_ms(&self) -> u64;
}

struct FixedClock { t: u64 }
```

```rust
impl Clock for FixedClock {
    fn now_ms(&self) -> u64 { self.t }
}

trait Audit: Send + Sync {
    fn record(&self, s: &str);
}

struct MemAudit {
    buf: Mutex<Vec<String>>,
}

impl MemAudit {
    fn new() -> Self { Self { buf: Mutex::new(Vec::new()) } }
}

impl Audit for MemAudit {
    fn record(&self, s: &str) {
        self.buf.lock().unwrap().push(s.to_string());
    }
}

struct Controller {
    clock: Arc<dyn Clock>,
    audit: Arc<dyn Audit>,
}

impl Controller {
    fn new(clock: Arc<dyn Clock>, audit: Arc<dyn Audit>) -> Self {
        Self { clock, audit }
    }

    fn action(&self, what: &str) {
        let msg = format!("t={} action={}", self.clock.now_ms(), what);
        self.audit.record(&msg);
        println!("{}", msg);
    }
}

fn main() {
    let clock: Arc<dyn Clock> = Arc::new(FixedClock { t: 12345 });
    let audit: Arc<dyn Audit> = Arc::new(MemAudit::new());

    let c = Controller::new(clock, audit);
    c.action("login");
}
```

This is Windows-friendly and scales well: `Arc` expresses sharing, dyn expresses runtime polymorphism, and `Mutex` expresses synchronization.

8.7 Outcome

After this chapter, you can:

- implement Strategy with either generics (static) or dyn (dynamic) based on architectural needs,
- express factories as explicit constructors returning enums or trait objects,
- build Observer systems using closures or trait-based subscribers with correct ownership/synchronization,
- model Command as enums to eliminate class hierarchies and visitor boilerplate,
- replace GoF State objects with explicit enum state machines and transitions,

- implement DI using trait boundaries with either compile-time generics or runtime `Arc<dyn Trait>` services.

Case Study: Plugin Architecture

9.1 Defining a Plugin Trait

A Rust plugin architecture starts with a **stable behavioral contract** (a trait) and a **host-owned context** that controls what plugins may access. The trait boundary is where you enforce safety, versioning policy, and capabilities.
Key design rules:

- Keep the plugin trait **small** and **object-safe** if you need dynamic dispatch.
- Prefer passing **capability objects** (narrow interfaces) over passing the whole host state.
- Make plugin errors explicit (`Result`) and keep panic across boundaries contained.

Minimal Object-Safe Plugin Contract

```
#[derive(Debug)]
pub enum PluginError {
    Failed(&'static str),
}

pub struct HostApi<'a> {
    // Narrow capability surface. Add only what plugins must do.
    log: &'a dyn Fn(&str),
```

```rust
}

impl<'a> HostApi<'a> {
    pub fn new(log: &'a dyn Fn(&str)) -> Self { Self { log } }
    pub fn log(&self, msg: &str) { (self.log)(msg); }
}

// Object-safe: no generics, no returning Self, no Self: Sized requirement.
pub trait Plugin: Send + Sync {
    fn name(&self) -> &'static str;
    fn init(&mut self, api: HostApi<'_>) -> Result<(), PluginError>;
    fn on_event(&mut self, api: HostApi<'_>, event: &str) -> Result<(), PluginError>;
}
```

9.2 Static vs Dynamic Plugins

Rust supports two broad plugin models:

- **Static plugins:** compiled into the binary; selected at compile time; typically generic dispatch or enum dispatch; fastest and simplest.
- **Dynamic plugins:** selected at runtime; typically trait objects; can be loaded from configuration; can be extended without recompiling the host (with extra ABI constraints if loaded from DLL).

9.2.1 Static Plugins (Compile-Time Selection)

Static plugins are ideal when the set of plugins is known and you want maximum performance and simplicity.

```rust
pub struct Uppercase;
```

```
impl Plugin for Uppercase {
    fn name(&self) -> &'static str { "uppercase" }

    fn init(&mut self, api: HostApi<'_>) -> Result<(), PluginError> {
        api.log("Uppercase init");
        Ok(())
    }

    fn on_event(&mut self, api: HostApi<'_>, event: &str) -> Result<(), PluginError> {
        api.log(&event.to_ascii_uppercase());
        Ok(())
    }
}

pub struct Prefix {
    p: String,
}

impl Prefix {
    pub fn new(p: &str) -> Self { Self { p: p.to_string() } }
}

impl Plugin for Prefix {
    fn name(&self) -> &'static str { "prefix" }

    fn init(&mut self, api: HostApi<'_>) -> Result<(), PluginError> {
        api.log("Prefix init");
        Ok(())
    }

    fn on_event(&mut self, api: HostApi<'_>, event: &str) -> Result<(), PluginError> {
        api.log(&format!("{}{}", self.p, event));
        Ok(())
    }
}

fn main() {
    let log = |s: &str| println!("[host] {}", s);
    let api = HostApi::new(&log);

    // Static selection: explicit concrete types.
    let mut a = Uppercase;
    let mut b = Prefix::new("ForgeVM: ");

    a.init(api).unwrap();
    b.init(api).unwrap();

    let api = HostApi::new(&log);
    a.on_event(api, "hello").unwrap();

    let api = HostApi::new(&log);
    b.on_event(api, "hello").unwrap();
}
```

This is the simplest "plugin" shape: just components in the same build.

9.2.2 Dynamic Plugins (Runtime Selection via dyn)

Dynamic selection uses `Box<dyn Plugin>` so you can keep a heterogeneous list of plugins and decide at runtime which plugins are enabled.

```
fn registry(kind: &str) -> Box<dyn Plugin> {
    match kind {
        "uppercase" => Box::new(Uppercase),
        "prefix" => Box::new(Prefix::new("ForgeVM: ")),
        _ => Box::new(Uppercase),
    }
}

fn run_pipeline(mut plugins: Vec<Box<dyn Plugin>>) -> Result<(), PluginError> {
    let log = |s: &str| println!("[host] {}", s);

    for p in plugins.iter_mut() {
        p.init(HostApi::new(&log))?;
    }

    let events = ["boot", "tick", "shutdown"];
    for e in events {
        for p in plugins.iter_mut() {
            p.on_event(HostApi::new(&log), e)?;
        }
    }
    Ok(())
}

fn main() {
    let enabled = ["prefix", "uppercase"]; // pretend this came from config
    let plugins = enabled.into_iter().map(registry).collect::<Vec<_>>();
    run_pipeline(plugins).unwrap();
}
```

9.3 Extensibility vs Performance

Use the following decision rules:

- **Static** when: plugin set is fixed; performance is critical; you want maximal inlining and minimal allocations.

- **Dynamic** when: plugin set is user-configurable; you need heterogeneous storage; you want runtime toggles; you accept vtable dispatch.
- If the plugin family is **closed**, consider **enum dispatch** as a third option (no vtables, still runtime selection).

Enum Dispatch (Closed Set, Runtime Selection, No Vtable)

```rust
enum BuiltinPlugin {
    Uppercase(Uppercase),
    Prefix(Prefix),
}

impl BuiltinPlugin {
    fn name(&self) -> &'static str {
        match self {
            BuiltinPlugin::Uppercase(p) => p.name(),
            BuiltinPlugin::Prefix(p) => p.name(),
        }
    }

    fn init(&mut self, api: HostApi<'_>) -> Result<(), PluginError> {
        match self {
            BuiltinPlugin::Uppercase(p) => p.init(api),
            BuiltinPlugin::Prefix(p) => p.init(api),
        }
    }

    fn on_event(&mut self, api: HostApi<'_>, event: &str) -> Result<(), PluginError> {
        match self {
            BuiltinPlugin::Uppercase(p) => p.on_event(api, event),
            BuiltinPlugin::Prefix(p) => p.on_event(api, event),
        }
    }
}

fn main() {
    let log = |s: &str| println!("[host] {}", s);
    let mut plugins = vec![
        BuiltinPlugin::Prefix(Prefix::new("ForgeVM: ")),
        BuiltinPlugin::Uppercase(Uppercase),
    ];

    for p in plugins.iter_mut() {
        p.init(HostApi::new(&log)).unwrap();
    }
    for p in plugins.iter_mut() {
        p.on_event(HostApi::new(&log), "hello").unwrap();
    }
}
```

This often beats dynamic dispatch when the plugin set is known and small but you still want runtime enable/disable.

9.4 Safety Boundaries

A plugin boundary is a **trust boundary**. In Rust, you design safety with:

- **Narrow capabilities:** pass a `HostApi` exposing only allowed operations.
- **No raw pointers:** keep the API safe; use slices, strings, and owned types.
- **Explicit errors:** return `Result`; never rely on exceptions.
- **Panic containment:** catch unwinding so one plugin cannot crash the host.

9.4.1 Contain Plugin Panics

```rust
use std::panic::{catch_unwind, AssertUnwindSafe};

fn safe_call(p: &mut dyn Plugin, event: &str) -> Result<(), PluginError> {
    let log = |s: &str| println!("[host] {}", s);
    let api = HostApi::new(&log);

    let r = catch_unwind(AssertUnwindSafe(|| p.on_event(api, event)));
    match r {
        Ok(v) => v,
        Err(_) => Err(PluginError::Failed("plugin panicked")),
    }
}

struct PanicPlugin;
impl Plugin for PanicPlugin {
    fn name(&self) -> &'static str { "panic" }
    fn init(&mut self, _api: HostApi<'_>) -> Result<(), PluginError> { Ok(()) }
    fn on_event(&mut self, _api: HostApi<'_>, _event: &str) -> Result<(), PluginError> {
        panic!("boom");
    }
}

fn main() {
    let mut p: Box<dyn Plugin> = Box::new(PanicPlugin);
    p.init(HostApi::new(&|s| println!("[host] {}", s))).unwrap();

    let r = safe_call(p.as_mut(), "tick");
    println!("result={:?}", r);
}
```

This is a practical safety measure for “in-process” plugins.

9.4.2 Versioning the Contract

The plugin trait is your ABI-at-the-language-level. Keep it stable:

- avoid changing method signatures,
- add new functionality via new traits or capability structs,
- prefer additive evolution: add new methods with defaults only if object-safety remains valid for your model.

9.5 Comparison with C++ Plugin Systems

C++ plugin systems commonly use:

- abstract base classes with virtual methods,
- factories returning base pointers,
- dynamic library loading (DLL) with exported C symbols to create instances,
- manual lifetime and exception boundary management.

Rust differs structurally:

- **Trait objects** provide the "virtual" behavior explicitly (`dyn`).
- **Ownership** clarifies lifetimes: host owns `Box<dyn Plugin>` and drops it deterministically.
- **No exceptions:** error propagation is explicit (`Result`); panic is not an error channel and should be contained.
- **Thread-safety is explicit:** `Send + Sync` bounds on `Plugin` enforce safe cross-thread usage.

Important boundary note (especially on Windows):

- In-process, runtime-loaded DLL plugins require a **stable ABI**. Rust's native ABI is not a stable cross-crate ABI for plugin binaries.
- For DLL-loaded plugins, the usual approach is a C ABI boundary (`extern "C"`) plus `#[repr(C)]` types and function tables, keeping Rust traits on one side of the boundary.

This booklet focuses on architectural patterns and safe in-process polymorphism. If you need DLL ABI stability, design an explicit C ABI layer and treat it as a separate engineering topic.

9.6 Outcome

After this chapter, you can:

- define a minimal, object-safe plugin trait with clear contracts,
- choose static, dynamic, or enum-based plugin dispatch intentionally,
- balance extensibility versus performance and binary size,
- enforce safety boundaries via narrow capabilities, explicit `Result` errors, and panic containment,
- map Rust plugin architecture choices to (and beyond) common C++ virtual/DLL plugin patterns on Windows.

Part V

Performance and Engineering Discipline

Performance-Aware OOP Design

10.1 Heap vs Stack Decisions

In Rust, allocation strategy is largely a design choice. Most values live on the stack by default; heap allocation appears explicitly via owning pointers like `Box`, `Vec`, `String`, `Rc`, and `Arc`. Performance-aware OOP in Rust means knowing when an object should be:

- **stack-owned** (fast allocation, predictable locality),
- **heap-owned** (needed for dynamic size, indirection, trait objects, shared ownership),
- **borrowed** (avoid ownership transfer, avoid allocation).

10.1.1 Stack-Owned Object (No Allocation)

```rust
struct Accum {
    sum: u64,
}

impl Accum {
    fn new() -> Self { Self { sum: 0 } }
    fn add(&mut self, x: u64) { self.sum += x; }
    fn sum(&self) -> u64 { self.sum }
}
```

```
fn main() {
    let mut a = Accum::new(); // stack-owned
    for i in 0..1_000 {
        a.add(i);
    }
    println!("{}", a.sum());
}
```

10.1.2 Heap Allocation for Trait Objects and Indirection

Trait objects typically require indirection because the size of the concrete type behind dyn `Trait` is not known at compile time.

```
trait Op { fn run(&self, x: u64) -> u64; }

struct Add1;
impl Op for Add1 { fn run(&self, x: u64) -> u64 { x + 1 } }

struct Mul2;
impl Op for Mul2 { fn run(&self, x: u64) -> u64 { x * 2 } }

fn main() {
    let ops: Vec<Box<dyn Op>> = vec![Box::new(Add1), Box::new(Mul2)];
    let mut v = 1u64;
    for op in ops {
        v = op.run(v);
    }
    println!("{}", v);
}
```

10.1.3 Avoiding Heap Where Possible: Generic Alternatives

If the set of operations is fixed at compile time, generics avoid indirection and usually allow inlining.

```rust
trait Op { fn run(&self, x: u64) -> u64; }

struct Add1; impl Op for Add1 { fn run(&self, x: u64) -> u64 { x + 1 } }
struct Mul2; impl Op for Mul2 { fn run(&self, x: u64) -> u64 { x * 2 } }

fn apply2<A: Op, B: Op>(a: &A, b: &B, mut x: u64) -> u64 {
    x = a.run(x);
    x = b.run(x);
    x
}

fn main() {
    let a = Add1;
    let b = Mul2;
    println!("{}", apply2(&a, &b, 1));
}
```

10.2 Dispatch Costs

Rust offers multiple dispatch mechanisms with different costs:

- **Static dispatch (generics):** resolved at compile time, no vtable, best inlining potential.
- **Dynamic dispatch (dyn):** vtable indirection + indirect call; inlining typically limited.
- **Enum dispatch:** a tagged branch (`match`); can often be optimized well; no vtable.

10.2.1 Dynamic Dispatch Cost Shape

A `dyn Trait` call typically means:

- load vtable pointer,
- load function pointer,
- indirect call (harder for branch prediction/inlining),
- extra indirection may reduce cache locality.

10.2.2 Static Dispatch Cost Shape

Static dispatch typically means:

- direct call (often inlined),
- more specialization opportunities,
- possibly larger binary due to monomorphization.

10.3 Inlining and Monomorphization

Monomorphization creates specialized copies per concrete type, enabling inlining and constant propagation across abstraction layers. This is how Rust achieves “zero-cost abstractions”.

10.3.1 Inlining-Friendly Generic Hot Path

```
trait Step { fn step(&self, x: u64) -> u64; }

struct XorShift;
impl Step for XorShift {
    fn step(&self, mut x: u64) -> u64 {
        x ^= x << 13;
        x ^= x >> 7;
        x ^= x << 17;
        x
    }
}

fn run<S: Step>(s: &S, mut x: u64, n: u32) -> u64 {
    for _ in 0..n {
        x = s.step(x);
    }
    x
}

fn main() {
    let s = XorShift;
    println!("{}", run(&s, 1, 1_000_000));
}
```

In optimized builds, the compiler can inline `step` into the loop and optimize aggressively.

10.3.2 Binary Size Consideration

If `run` is instantiated with many different `S` types, code size can grow. Cost-aware design weighs:

- speed and inlining benefits,
- compile time,
- final binary size and i-cache pressure.

10.4 Enum Dispatch vs dyn Dispatch

When the set of implementors is closed and known, enum dispatch is often a strong option:

- no heap allocation required if stored directly,
- no vtable indirection,
- branch-based dispatch that the optimizer can sometimes simplify.

10.4.1 Enum Dispatch Example

```rust
trait Op { fn run(&self, x: u64) -> u64; }

struct Add1; impl Op for Add1 { fn run(&self, x: u64) -> u64 { x + 1 } }
struct Mul2; impl Op for Mul2 { fn run(&self, x: u64) -> u64 { x * 2 } }

enum OpEnum {
    Add1(Add1),
    Mul2(Mul2),
}

impl OpEnum {
    fn run(&self, x: u64) -> u64 {
        match self {
            OpEnum::Add1(o) => o.run(x),
            OpEnum::Mul2(o) => o.run(x),
        }
    }
}

fn main() {
    let ops = vec![OpEnum::Add1(Add1), OpEnum::Mul2(Mul2)];
    let mut v = 1u64;
    for op in &ops {
        v = op.run(v);
    }
    println!("{}", v);
}
```

10.4.2 dyn Dispatch Example

```
trait Op { fn run(&self, x: u64) -> u64; }

struct Add1; impl Op for Add1 { fn run(&self, x: u64) -> u64 { x + 1 } }
struct Mul2; impl Op for Mul2 { fn run(&self, x: u64) -> u64 { x * 2 } }

fn main() {
    let ops: Vec<Box<dyn Op>> = vec![Box::new(Add1), Box::new(Mul2)];
    let mut v = 1u64;
    for op in ops {
        v = op.run(v);
    }
    println!("{}", v);
}
```

Decision rule:

- **Enum** when the set is closed and you want predictable performance without vtables.
- **dyn** when the set is open/extensible (plugins, user-provided implementors).

10.5 Benchmarking Guidance

Performance-aware Rust OOP requires measurement in realistic conditions. Practical guidance:

- Benchmark with **release builds** (`cargo run -release`) and keep inputs representative.
- Separate algorithm cost from measurement overhead; avoid I/O in the hot loop.
- Warm up: run the function multiple times before timing to reduce first-run noise.
- Prefer **stable timers** and measure multiple iterations; report min/median.
- Compare alternatives that differ only in dispatch/allocation, not in unrelated logic.

10.5.1 Simple Timing Harness (Std-Only, Windows-Friendly)

```rust
use std::time::Instant;

trait Op { fn run(&self, x: u64) -> u64; }

struct Add1; impl Op for Add1 { fn run(&self, x: u64) -> u64 { x + 1 } }
struct Mul2; impl Op for Mul2 { fn run(&self, x: u64) -> u64 { x ^ (x << 1) } }

fn bench_static<A: Op, B: Op>(a: &A, b: &B, iters: u32) -> u64 {
    let mut x = 1u64;
    for _ in 0..iters {
        x = a.run(x);
        x = b.run(x);
    }
    x
}

fn bench_dyn(ops: &[Box<dyn Op>], iters: u32) -> u64 {
    let mut x = 1u64;
    for _ in 0..iters {
        for op in ops {
            x = op.run(x);
        }
    }
    x
}

fn main() {
    let iters = 2_000_00u32;

    // Warm-up
    let _ = bench_static(&Add1, &Mul2, 10_000);

    let t0 = Instant::now();
    let out1 = bench_static(&Add1, &Mul2, iters);
    let d1 = t0.elapsed();

    let ops: Vec<Box<dyn Op>> = vec![Box::new(Add1), Box::new(Mul2)];
    let _ = bench_dyn(&ops, 10_000); // warm-up

    let t1 = Instant::now();
    let out2 = bench_dyn(&ops, iters);
    let d2 = t1.elapsed();

    println!("static out={} time={:?}", out1, d1);
    println!("dyn    out={} time={:?}", out2, d2);
}
```

This harness is intentionally simple. For serious work, you typically use a statistical benchmarking framework, but the architectural point remains: measure *your* workloads and compare dispatch models under `-release`.

10.6 Outcome

After this chapter, you can:

- choose stack vs heap based on object size, lifetime, heterogeneity, and sharing needs,
- estimate dispatch costs and select static, dynamic, or enum dispatch intentionally,
- exploit monomorphization and inlining for hot paths while managing binary size,
- decide between enum dispatch and `dyn` dispatch based on closed vs open extensibility,
- benchmark alternatives correctly on Windows using release builds and noise-resistant timing.

Migrating C++ OOP Code to Rust

11.1 Converting Inheritance Hierarchies

Most C++ inheritance trees fall into two categories:

- **Closed hierarchies:** the set of derived types is known and fixed (AST nodes, protocol messages, commands).
- **Open hierarchies:** third parties can add new derived types later (plugins, user-defined behaviors).

Rust has two primary replacements:

- **Closed** $\to$ `enum` + `match`.
- **Open** $\to$ traits + generics or `dyn Trait`.

11.1.1 Closed Hierarchy: Base Class $\to$ `enum`

C++ style (conceptual):

- `Expr` base with derived `Num`/`Add`/`Mul`.

Rust replacement:

```rust
#[derive(Clone, Debug)]
enum Expr {
    Num(i64),
    Add(Box<Expr>, Box<Expr>),
    Mul(Box<Expr>, Box<Expr>),
}

impl Expr {
    fn eval(&self) -> i64 {
        match self {
            Expr::Num(n) => *n,
            Expr::Add(a, b) => a.eval() + b.eval(),
            Expr::Mul(a, b) => a.eval() * b.eval(),
        }
    }
}

fn main() {
    let e = Expr::Add(Box::new(Expr::Num(2)), Box::new(Expr::Mul(Box::new(Expr::Num(3)),
    ↪  Box::new(Expr::Num(4)))));
    println!("{}", e.eval());
}
```

11.1.2 Open Hierarchy: Base Class Interface → Trait

Instead of “derive and override”, define a trait and allow implementors.

```rust
trait Payment {
    fn pay(&self, amount_cents: u64) -> bool;
}

struct Card;
impl Payment for Card {
    fn pay(&self, amount_cents: u64) -> bool {
        println!("card charge {}", amount_cents);
        true
    }
}

struct Cash;
impl Payment for Cash {
    fn pay(&self, amount_cents: u64) -> bool {
        println!("cash {}", amount_cents);
        true
    }
}

fn main() {
    let p: Vec<Box<dyn Payment>> = vec![Box::new(Card), Box::new(Cash)];
    for x in p {
        let _ = x.pay(500);
    }
}
```

11.2 Mapping Virtual Interfaces to Traits

C++ virtual interfaces map directly to Rust traits, but the migration requires choosing the dispatch style intentionally.

- **Static dispatch (generic):** `fn f<T: Trait>(x: T)` — fastest, monomorphized.
- **Dynamic dispatch:** `fn f(x: &dyn Trait)` or store `Box<dyn Trait>` — runtime flexibility.

11.2.1 Static Dispatch Replacement

```rust
trait Writer {
    fn write(&mut self, s: &str);
}

struct Buffer { out: String }
impl Buffer { fn new() -> Self { Self { out: String::new() } } }

impl Writer for Buffer {
    fn write(&mut self, s: &str) { self.out.push_str(s); }
}

fn emit<W: Writer>(w: &mut W) {
    w.write("hello");
    w.write(" ");
    w.write("world");
}

fn main() {
    let mut b = Buffer::new();
    emit(&mut b);
}
```

11.2.2 Dynamic Dispatch Replacement

```rust
trait Writer {
    fn write(&mut self, s: &str);
}

struct Stdout;
impl Writer for Stdout {
    fn write(&mut self, s: &str) { print!("{}", s); }
}

fn emit(w: &mut dyn Writer) {
    w.write("hello");
    w.write(" ");
    w.write("world");
}

fn main() {
    let mut s = Stdout;
    emit(&mut s);
}
```

11.2.3 Object Safety Checklist (Migration Trap)

A trait can be used as `dyn Trait` only if it is object-safe. Avoid:

- methods returning `Self`,
- generic methods on the trait,
- implicit `Self: Sized` requirements for object use.

11.3 Replacing Smart Pointers

Many C++ OOP designs are built around pointer ownership semantics. In Rust, ownership is the default; heap/shared ownership is explicit.

11.3.1 `unique_ptr<T>` → Owned Value or `Box<T>`

- If you do not need heap allocation: store `T` directly.

- If you need indirection / stable address / dynamic size: use `Box<T>`.

```rust
struct Engine { v: u64 }

fn main() {
    let e = Engine { v: 1 };      // owned on stack (often best)
    let b = Box::new(Engine { v: 2 }); // heap indirection
    println!("{} {}", e.v, b.v);
}
```

11.3.2 `shared_ptr<T>` → `Rc<T>` or `Arc<T>`

- Single-threaded sharing: `Rc<T>`.
- Cross-thread sharing: `Arc<T>`.
- Shared mutation: combine with `RefCell` (single-thread) or `Mutex/RwLock` (multi-thread).

Single-Thread Shared Graph: `Rc<RefCell<T»`

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

#[derive(Debug)]
struct Node {
    name: String,
    next: Option<Rc<RefCell<Node>>>,
}

fn main() {
    let a = Rc::new(RefCell::new(Node { name: "A".to_string(), next: None }));
    let b = Rc::new(RefCell::new(Node { name: "B".to_string(), next: None }));
    a.borrow_mut().next = Some(Rc::clone(&b));
    println!("{:?}", a.borrow().next.as_ref().unwrap().borrow().name);
}
```

Multi-Thread Shared State: `Arc<Mutex<T»`

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

fn main() {
    let c = Arc::new(Mutex::new(0u64));

    let mut ts = Vec::new();
    for _ in 0..4 {
        let cc = Arc::clone(&c);
        ts.push(thread::spawn(move || {
            for _ in 0..10000 {
                *cc.lock().unwrap() += 1;
            }
        }));
    }
    for t in ts { t.join().unwrap(); }
    println!("{}", *c.lock().unwrap());
}
```

11.3.3 `T*` Raw Pointers → Borrowed References

In safe Rust, prefer `&T`, `&mut T`, slices, and iterators. Use raw pointers mainly for FFI and low-level `unsafe` code.

11.4 Common Mistakes C++ Developers Make

11.4.1 Mistake 1: Rebuilding Inheritance Instead of Modeling the Domain

C++ developers often attempt to recreate base/derived hierarchies. Use:

- `enum` for closed families,
- traits for open extension,
- composition for reuse of state/implementation.

11.4.2 Mistake 2: Overusing `Arc<Mutex<T»`

This is the Rust equivalent of "everything is a `shared_ptr`". Prefer:

- ownership + borrowing for most flows,
- message passing,
- `RwLock` when reads dominate,
- narrower locking scopes.

11.4.3 Mistake 3: Fighting Borrowing Instead of Designing Around It

If a type is hard to use under borrowing rules, it often indicates a design smell:

- unclear ownership,
- hidden aliasing requirements,
- missing separation between immutable view and mutable core,
- state machine that should be explicit.

11.4.4 Mistake 4: Returning References to Temporaries

C++ code sometimes returns references/pointers to ephemeral values by mistake. Rust prevents this, but you may still struggle when you try to fabricate lifetimes. Fix by:

- returning owned values when the callee cannot guarantee lifetime,
- storing the data in the owning struct and returning references to stored data,
- redesigning the API boundary.

11.4.5 Mistake 5: Trait Object Misuse (Object Safety and Allocation)

If you do not need runtime polymorphism, do not use `Box<dyn Trait>`. Prefer generics or enums:

- generics for open but compile-time selection,
- enums for closed families.

11.4.6 Mistake 6: Assuming C++-Style "Const" Equals Rust `&self`

Rust `&self` means shared borrow; mutation is not possible unless you use interior mutability. This is stronger than C++ `const` in practice, and it changes how you design caches and lazy fields.

11.5 Outcome

After this chapter, you can:

- convert inheritance hierarchies into `enum`-based closed models or trait-based open extension,

- map virtual interfaces to traits with a deliberate static vs dynamic dispatch decision,
- replace C++ smart pointers with Rust ownership, `Box`, `Rc`, `Arc`, and explicit synchronization,
- avoid the most common migration traps (inheritance emulation, over-locking, fighting borrow rules, misusing `dyn`),
- produce Rust designs that are simpler, safer, and performance-aware on Windows.

Mastery Checklist

12.1 Complete Concept Checklist

Use this as a final verification that you can design Rust "OOP" systems intentionally and idiomatically.

12.1.1 Object Modeling and Encapsulation

- I can design a type with `struct + impl` where fields are private and invariants are enforced by constructors.
- I can choose `&self` vs `&mut self` vs `self` to encode usage constraints.
- I can expose stable APIs with `pub`/`pub(crate)` and module boundaries.
- I can use newtypes to enforce domain meaning (IDs, tokens, units) and prevent mixing.

12.1.2 Traits and Polymorphism

- I can define object-safe traits for runtime polymorphism and keep them small.
- I can use generic bounds (`T: Trait`) for static dispatch in hot paths.

- I can choose between generics, `dyn Trait`, and `enum` dispatch based on open vs closed sets.
- I can apply default methods, associated types, and blanket implementations safely.

12.1.3 Ownership-Driven API Design

- I can design APIs that borrow inputs (`&str`, `&[T]`) and avoid unnecessary allocation.
- I can return borrowed views safely (`&T`, `&str`, slices) when lifetime allows.
- I can explain why a reference cannot outlive the owner and redesign when needed.
- I can use `Result` and explicit error enums instead of exception hierarchies.

12.1.4 Interior Mutability and Shared Ownership

- I can justify using `Cell` for small `Copy` state and `RefCell` for single-threaded interior mutability.
- I can use `Rc` for single-thread graphs and `Arc` for cross-thread sharing.
- I can design shared mutable state using `Arc<Mutex<T»` or `Arc<RwLock<T»` with minimal lock scope.
- I can prevent reference cycles using `Weak`.

12.1.5 Architecture Patterns

- I can implement Strategy with generics (static) or `dyn` (dynamic) based on requirements.
- I can implement Command and State as enums and remove virtual hierarchies.

- I can implement DI via traits (`T: Trait` or `Arc<dyn Trait>`) without service locators.
- I can design plugin systems with explicit safety and versioning boundaries.

12.1.6 Performance Discipline

- I know when heap allocation is required (`Box`, trait objects, dynamic size) and when stack is best.
- I can reason about dispatch costs, inlining, monomorphization, and binary size growth.
- I can benchmark correctly in `-release` and isolate measurement overhead.

12.2 Practical Mini-Projects

Complete these small projects to prove mastery. Each is designed to touch key constraints: ownership, polymorphism, invariants, and concurrency. Keep them std-only and Windows-friendly.

12.2.1 Mini-Project 1: Policy-Driven Logger (Strategy + DI)

Goal: implement a `Logger` contract and inject it into a service. Provide both static and dynamic variants.

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

trait Logger: Send + Sync {
    fn log(&self, msg: &str);
}

struct StdoutLogger;
impl Logger for StdoutLogger {
    fn log(&self, msg: &str) { println!("[stdout] {}", msg); }
}

struct MemLogger {
    buf: Mutex<Vec<String>>,
}
impl MemLogger {
    fn new() -> Self { Self { buf: Mutex::new(Vec::new()) } }
    fn snapshot(&self) -> Vec<String> { self.buf.lock().unwrap().clone() }
}
impl Logger for MemLogger {
    fn log(&self, msg: &str) { self.buf.lock().unwrap().push(msg.to_string()); }
}

struct Service {
    log: Arc<dyn Logger>,
}
impl Service {
    fn new(log: Arc<dyn Logger>) -> Self { Self { log } }
    fn run(&self) {
        self.log.log("start");
        self.log.log("work");
        self.log.log("done");
    }
}

fn main() {
    let mem = Arc::new(MemLogger::new());
    let svc = Service::new(Arc::clone(&mem));
    svc.run();
    println!("{:?}", mem.snapshot());
}
```

Tasks:

- Add a generic version `Service<L: Logger>` and compare with dyn.
- Add filtering policy (only log warnings) using a wrapper type (composition + delegation).

12.2.2 Mini-Project 2: Command Processor (Enum + Exhaustive Match)

Goal: replace a virtual command hierarchy with a closed command enum.

```
#[derive(Debug)]
enum Command {
    AddUser { name: String },
    RemoveUser { id: u64 },
    RenameUser { id: u64, new_name: String },
}

struct App {
    next: u64,
}

impl App {
    fn new() -> Self { Self { next: 1 } }

    fn exec(&mut self, cmd: Command) {
        match cmd {
            Command::AddUser { name } => {
                let id = self.next;
                self.next += 1;
                println!("add id={} name={}", id, name);
            }
            Command::RemoveUser { id } => println!("remove id={}", id),
            Command::RenameUser { id, new_name } => println!("rename id={} -> {}", id,
            ↪   new_name),
        }
    }
}

fn main() {
    let mut app = App::new();
    app.exec(Command::AddUser { name: "Ayman".to_string() });
    app.exec(Command::RenameUser { id: 1, new_name: "Ayman2".to_string() });
}
```

Tasks:

- Add serialization format (simple) and prove exhaustive update when adding a variant.
- Add validation using newtypes (`UserId`, `UserName`).

12.2.3 Mini-Project 3: State Machine (Enum State Pattern)

Goal: encode a protocol as explicit states and transitions.

```
#[derive(Debug)]
enum Session {
    New,
    Authenticated { user: String },
    Closed,
}

impl Session {
    fn login(self, user: &str) -> Self {
        match self {
            Session::New => Session::Authenticated { user: user.to_string() },
            x => x,
        }
    }
    fn logout(self) -> Self {
        match self {
            Session::Authenticated { .. } => Session::Closed,
            x => x,
        }
    }
}

fn main() {
    let s = Session::New;
    let s = s.login("Ayman");
    let s = s.logout();
    println!("{:?}", s);
}
```

Tasks:

- Add an error enum and return `Result<Self, Error>` for invalid transitions.
- Convert to type-state (compile-time) if the protocol must not allow invalid calls.

12.2.4 Mini-Project 4: Plugin Pipeline (dyn + Safety Boundary)

Goal: implement a plugin list and isolate errors/panics.

```
use std::panic::{catch_unwind, AssertUnwindSafe};

#[derive(Debug)]
enum PluginError { Failed(&'static str) }

struct HostApi<'a> { log: &'a dyn Fn(&str) }
impl<'a> HostApi<'a> { fn new(log: &'a dyn Fn(&str)) -> Self { Self { log } } fn log(&self,
↪ s: &str) { (self.log)(s) } }
```

```
trait Plugin {
    fn name(&self) -> &'static str;
    fn on_event(&mut self, api: HostApi<'_>, e: &str) -> Result<(), PluginError>;
}

struct Upper;
impl Plugin for Upper {
    fn name(&self) -> &'static str { "upper" }
    fn on_event(&mut self, api: HostApi<'_>, e: &str) -> Result<(), PluginError> {
        api.log(&e.to_ascii_uppercase());
        Ok(())
    }
}

fn safe_event(p: &mut dyn Plugin, e: &str) -> Result<(), PluginError> {
    let log = |s: &str| println!("[host] {}", s);
    let api = HostApi::new(&log);

    let r = catch_unwind(AssertUnwindSafe(|| p.on_event(api, e)));
    match r {
        Ok(v) => v,
        Err(_) => Err(PluginError::Failed("panic in plugin")),
    }
}

fn main() {
    let mut ps: Vec<Box<dyn Plugin>> = vec![Box::new(Upper)];
    for p in ps.iter_mut() {
        let _ = safe_event(p.as_mut(), "tick");
    }
}
```

Tasks:

- Add a `PluginId` newtype and a registration mechanism.
- Add version fields and refuse incompatible plugins at runtime.

12.3 Architectural Self-Review Template

Use this checklist before you commit to a design. It catches typical C++→Rust migration mistakes early.

12.3.1 Model Choice

- Is the set of variants **closed**? If yes, prefer `enum` dispatch.
- Is the set **open/extensible**? If yes, use traits and decide `generic` vs `dyn`.
- Am I using `dyn` only because I am thinking in virtual base classes? If yes, reconsider.

12.3.2 Ownership and Lifetimes

- Who owns each object? Is ownership transfer explicit (`T/self`)?
- Are inputs borrowed where possible (`&str`, `&[T]`)?
- Are outputs borrowed only when the owner can guarantee lifetime?
- Did I introduce shared ownership (`Rc/Arc`) only when required?

12.3.3 Mutability and Concurrency

- Can the API remain mostly `&self` with internal synchronization?
- If I used `Mutex/RwLock`, is the lock scope minimal?

- Do I understand contention hotspots and potential deadlocks?
- Did I accidentally create a design that needs `Rc<RefCell<...»` everywhere? If yes, revisit ownership model.

12.3.4 Safety Boundary

- Are failures represented as `Result` with explicit error types?
- If this is a plugin boundary, do I contain panics and narrow capabilities?
- Did I keep unsafe code isolated, documented, and minimal (ideally zero)?

12.3.5 Performance Posture

- Is heap allocation truly needed here?
- Is static dispatch required for hot paths?
- Would enum dispatch be faster and simpler than `dyn`?
- Did I benchmark in `-release` with realistic inputs?

12.4 When to Prefer Rust OOP Over C++ OOP

Prefer Rust OOP when one or more of the following are primary requirements:

12.4.1 Safety-Critical Constraints

- You cannot tolerate use-after-free, double-free, or iterator invalidation bugs.
- The codebase handles untrusted input and must be resilient by construction.
- Multi-threaded correctness is essential; data races are unacceptable.

12.4.2 Architecture That Benefits from Strong Invariants

- Domain invariants must be enforced at compile time (state machines, protocols, lifetimes).
- API misuse is common and must be prevented structurally (not via documentation).
- Refactoring should produce compile-time feedback when cases are missing (exhaustive `match` on enums).

12.4.3 Extensibility With Explicit Boundaries

- You want plugin-like extensibility but still require tight safety boundaries.
- You need explicit ownership and concurrency policies encoded in types (`Send`/`Sync`, `Arc`, locks).

12.4.4 When C++ Still Wins

C++ OOP may be preferable when:

- ABI stability across independently built components is a hard requirement and you rely on a long-lived C++ ABI strategy inside one toolchain ecosystem.
- You have deep existing frameworks that assume inheritance and virtual dispatch and the cost of redesign is not justified.
- You must integrate tightly with legacy C++ object models where rewriting boundaries would dominate the project.

12.5 Outcome

At this point you can:

- verify mastery of Rust OOP mechanisms with a complete checklist,
- build small projects that exercise real architectural constraints (ownership, polymorphism, concurrency),
- review your designs using a Rust-native architectural template,
- decide when Rust OOP is the right engineering choice compared to modern C++ OOP.

Appendices

Appendix A: C++ → Rust Concept Mapping Table

Core Object Model Mapping

C++ Concept	Rust Equivalent	Notes
Class with private members	`struct + impl`	Fields private by default; visibility controlled with pub.
Constructor	`impl Type { fn new(...) -> Self }`	Any associated function can act as constructor; no special syntax.
Destructor	`Drop` trait	Deterministic destruction; RAII preserved.
Virtual method	Trait method	Use generics (static) or dyn `Trait` (dynamic).
Abstract base class	Trait	No state in trait; state via composition.
Multiple inheritance	Trait composition + composition of structs	No implementation inheritance; avoids diamond problem.

C++ Concept	**Rust Equivalent**	**Notes**
Protected members	Module privacy + composition	Prefer exposing behavior over shared state.
`std::variant`	`enum`	Exhaustive `match`; first-class algebraic sum type.
`unique_ptr<T>`	Owned value or `Box<T>`	Ownership is default; heap explicit.
`shared_ptr<T>`	`Rc<T>` / `Arc<T>`	Explicit single-thread vs multi-thread sharing.
Raw pointer	`&T`, `&mut T`, or raw pointer in `unsafe`	Borrowing rules enforced at compile time.
Const method	`&self`	No mutation unless interior mutability is used.
Exception hierarchy	`Result<T, E>` with enum error	Explicit, exhaustive failure modeling.

Polymorphism Strategy

Scenario	**Rust Pattern**	**Rationale**
Closed hierarchy	`enum` + `match`	Compile-time exhaustiveness; no vtable.
Open extension	Trait + generics	Zero-cost abstraction.
Runtime extensibility	`Box<dyn Trait>`	Vtable dispatch; heterogeneous collections.

Appendix B: Object Safety Rules Quick Reference

A trait can be used as dyn `Trait` only if it is object-safe.

Object-Safe Requirements

- The trait must not require `Self: Sized`.
- Methods must not return `Self`.
- Methods must not have generic type parameters.
- Methods must use `&self`, `&mut self`, or `Box<Self>` receiver (if consuming).

Non Object-Safe Example

```
trait Bad {
    fn make(&self) -> Self; // not object-safe
}
```

Object-Safe Version

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

Checklist Before Using dyn

- Does this trait need runtime polymorphism?
- Are there generic methods? If yes, redesign.
- Are you returning `Self`? If yes, consider associated constructors instead.

- Is the trait minimal and focused?

Appendix C: Trait Bound Cheat Sheet

Basic Bounds

```
fn f<T: Trait>(x: T) {}
fn g<T: Trait + Send + Sync>(x: T) {}
fn h<T>(x: T) where T: Trait {}
```

Borrowing with Bounds

```
fn print_all<T: std::fmt::Display>(items: &[T]) {
    for i in items {
        println!("{}", i);
    }
}
```

Associated Types

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

Blanket Implementation

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

```rust
impl<T: std::fmt::Display> Named for T {
    fn name(&self) -> &str { "displayable" }
}
```

Common Bounds Meaning

Bound	**Meaning**
`Send`	Type can be transferred across threads.
`Sync`	References can be shared across threads safely.
`'static`	No non-static references inside; can live for program duration.
`Sized`	Compile-time known size (default unless `?Sized`).

Appendix D: Ownership Patterns Quick Guide

Owned Value (Default)

```rust
struct User { name: String }

fn main() {
    let u = User { name: "Ayman".to_string() };
}
```

Borrowed View

```rust
fn greet(name: &str) {
    println!("Hello {}", name);
}
```

Move Semantics

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

fn main() {
    let s = String::from("x");
    consume(s);
    // s cannot be used here
}
```

Shared Ownership (Single Thread)

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

let a = Rc::new(5);
let b = Rc::clone(&a);
```

Shared Ownership (Multi Thread)

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

let a = Arc::new(5);
let b = Arc::clone(&a);
```

Interior Mutability (Single Thread)

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

let v = RefCell::new(vec![1,2,3]);
v.borrow_mut().push(4);
```

Thread-Safe Shared Mutation

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

let v = Arc::new(Mutex::new(vec![1,2,3]));
v.lock().unwrap().push(4);
```

Decision Table

Need	**Tool**	**Notes**
Exclusive ownership	Owned value	Fastest, simplest.
Heap indirection	`Box<T>`	For dynamic size or trait objects.
Shared (single-thread)	`Rc<T>`	No thread safety.
Shared (multi-thread)	`Arc<T>`	Atomic refcount.
Shared mutation (single-thread)	`RefCell<T>`	Runtime borrow checks.
Shared mutation (multi-thread)	`Mutex<T>` / `RwLock<T>`	Lock-based synchronization.

Final Architectural Reminder

- Prefer ownership and borrowing over shared pointers.
- Prefer `enum` over inheritance for closed hierarchies.
- Prefer generics over dyn when performance matters.
- Introduce shared ownership and interior mutability only when design requires it.

References

Official Rust Language and Standard Library Sources

- **The Rust Programming Language (The Book)** Authoritative introduction maintained by the Rust Project. Covers ownership, borrowing, traits, generics, enums, pattern matching, smart pointers, concurrency, and object safety fundamentals.
- **Rust Reference** Formal language specification describing syntax, semantics, type system rules, trait object behavior, object safety constraints, lifetimes, and memory model.
- **Rustonomicon** Advanced guide to unsafe Rust, ownership invariants, aliasing rules, drop semantics, Send/Sync auto traits, and FFI safety boundaries.
- **Standard Library Documentation** Official documentation for core types and traits: `Box`, `Rc`, `Arc`, `Cell`, `RefCell`, `Mutex`, `RwLock`, `Drop`, `Send`, `Sync`, `Result`, `Iterator`, and more.

Ownership, Lifetimes, and Type System Foundations

- Official compiler documentation and Rust Reference sections on:
  - Borrow checker rules

  - Lifetime elision
  - Variance and subtyping
  - Trait coherence and orphan rules
  - Object safety constraints

- Rust RFC archives describing:
  - Trait objects and `dyn` syntax
  - Non-lexical lifetimes
  - Specialization (design discussions)
  - Auto traits (`Send`, `Sync`)

Polymorphism and Abstraction Design

- Rust official materials on:
  - Generics and monomorphization
  - Trait bounds and associated types
  - Blanket implementations
  - Trait object layout and vtables
- Documentation and language discussions clarifying:
  - Differences between static dispatch and dynamic dispatch
  - Enum-based modeling for closed hierarchies
  - Performance implications of `dyn Trait`

Concurrency and Shared Ownership

- Official standard library documentation for:
  - `Rc` and `Arc`
  - `Cell` and `RefCell`
  - `Mutex` and `RwLock`
  - Thread spawning and synchronization primitives
- Rust Reference and Nomicon discussions on:
  - Data race definition
  - Send/Sync auto traits
  - Interior mutability guarantees
  - Memory safety model

Error Handling and Algebraic Modeling

- Standard library documentation for:
  - `Result<T, E>`
  - `Option<T>`
  - Pattern matching and exhaustive `match`
- Official guidance on:
  - Error enums as closed failure models
  - Idiomatic propagation using the `?` operator
  - Designing domain-specific error types

Performance and Engineering Discipline

- Official compiler documentation on:
  - Monomorphization
  - Inlining
  - Code generation in release mode
  - Zero-cost abstraction principle
- Standard practices in Rust engineering:
  - Measuring performance in `-release`
  - Avoiding unnecessary heap allocation
  - Understanding vtable dispatch cost
  - Enum dispatch vs dynamic dispatch trade-offs

Interoperability and ABI Considerations

- Rust Reference sections on:
  - FFI safety
  - `extern "C"` functions
  - `#[repr(C)]` layout guarantees
  - Unsafe blocks and boundary isolation
- Nomicon discussions on:
  - Raw pointers

  - Drop safety
  - Unwinding across boundaries

Comparative Context with Modern C++

- C++ standard library documentation for:
  - Smart pointers (`unique_ptr`, `shared_ptr`)
  - `std::variant`
  - Virtual dispatch and vtable behavior
  - RAII and deterministic destruction
- Comparative discussions in language documentation clarifying:
  - Differences between inheritance and trait-based abstraction
  - Differences between exception hierarchies and explicit error enums
  - Differences in data race prevention guarantees

Recommended Reading Path

For a systematic deepening of expertise:

1. Revisit ownership, borrowing, and lifetimes in the official language book.
2. Study trait object rules and object safety in the Rust Reference.
3. Review interior mutability and concurrency sections of the standard library documentation.

4. Explore unsafe invariants in the Rustonomicon to understand guarantees at the boundary.
5. Re-evaluate performance assumptions using release-mode benchmarks.

These materials collectively define the authoritative, up-to-date foundation for mastering object-oriented design in Rust as practiced in modern systems engineering.