From C to Rust

MODERNIZING PRODUCTION SYSTEMS WITHOUT STARTING OVER

```c
// legacy C
int process(struct data* d) {
    if (d == NULL) return -1;
    // ...
    return 0;
}
```

```rust
// safe, modern Rust
fn process(data: &Data) -> Result<()> {
    // ...
    Ok(())
}
```

Evaluate Your System

Design Safe Boundaries

Migrate Incrementally

Keep Production Running

Practical strategies, real-world tradeoffs, and complete, runnable examples at every step.

JAMES REDFORD

From C to Rust

Modernizing Production Systems Without Starting Over

Steve Publications

This book is available at https://leanpub.com/fromctorust

This version was published on 2026-08-25

Contents

Modernizing Production Systems Without Starting Over

This book is a practical, production-focused guide for engineers and organizations incrementally introducing Rust into mature C codebases. It shows how to evaluate existing systems, design safe interoperability boundaries, migrate components without destabilizing production behavior, and maintain working software throughout the transition. The emphasis is on engineering realism over hype: when Rust delivers real value, when it does not, what costs are involved, and exactly how to proceed step by step with complete, runnable examples at every stage.

Introduction: Why Incremental Modernization Matters

You have a C codebase that works. It has been in production for years or perhaps decades. It handles millions of requests per day, controls critical hardware, or forms the backbone of infrastructure your organization depends on. Someone has suggested rewriting it in Rust to improve safety and maintainability. Your immediate reaction is probably skepticism, because you know how rewrites go: they take longer than anyone predicts, they introduce new bugs into code that already works, and they often end up abandoned when business priorities shift.

You are right to be skeptical. A big-bang rewrite of a mature C system into Rust is almost always the wrong answer. But that does not mean you should ignore the growing pressure to improve memory safety in your systems. Around seventy percent of critical security vulnerabilities at major technology companies stem from memory safety defects in C and C++ code alone [1,2]. Government agencies including CISA and the NSA are now formally urging organizations to adopt memory-safe languages for new development and to create roadmaps for reducing unsafe code in existing systems [3]. The pressure is real, the statistics are clear, and ignoring them is no longer a defensible position.

The good news is that you do not have to choose between a reckless rewrite and doing nothing. Rust was designed from the beginning with interoperability as a first-class concern. You can introduce Rust into a C system gradually, component by component, while keeping everything working throughout the transition. This book shows you exactly how to do that.

Over the course of the following chapters we will cover the complete modernization lifecycle: assessing your existing codebase to identify migration candidates, designing FFI boundaries that prevent undefined behavior between languages, creating safe abstractions around unsafe C APIs, building mixed-language projects with real build systems, testing behavioral equivalence rigorously before releasing changes, measuring performance to ensure you are

not regressing, and managing the organizational challenges of a team learning a new language while maintaining production software.

Every major concept in this book is accompanied by complete, runnable code examples that you can build and run yourself. We start with small, self-contained demonstrations and progressively construct realistic migration projects that begin as working C applications and are modernized piece by piece while remaining executable at every stage. You will see the before state, the migration boundary, the intermediate interoperable state, and the final architecture for each example.

This book assumes you are already comfortable with C: pointers, manual memory management, system calls, build systems, and production deployment. It is not a general introduction to Rust for beginners. Instead it maps your existing mental model of systems programming onto Rust precisely, showing what changes, what stays the same, and where the pitfalls lie when moving between languages.

The central thesis is simple: Rust can materially improve the safety, maintainability, and reliability of mature C systems, but only when introduced incrementally through disciplined FFI boundaries, careful module selection, and rigorous testing. The best modernization strategy is rarely a rewrite. It is a phased migration that keeps the system working throughout, leverages C where it already works well, and uses Rust to eliminate specific classes of defects in high-risk components.

Let us begin by understanding exactly what problem we are trying to solve.

Chapter 1: The Modernization Case

What Production C Systems Actually Look Like Today

C is everywhere in production software. Operating system kernels, database engines, networking stacks, embedded firmware, cryptographic libraries, compilers, file systems, protocol parsers, graphics drivers, storage subsystems, command-line utilities, and countless internal tools all run on C code written over the last fifty years. This ubiquity is not accidental. C gives programmers direct control over memory layout, hardware interaction, and execution behavior with minimal abstraction overhead. When you need predictable performance, small binaries, or access to platform-specific features without a runtime in the way, C remains one of the few practical choices.

But that same direct control comes at a cost. Every pointer dereference, every manual allocation, every buffer size calculation, every race condition between threads is your responsibility. The compiler will not save you from use-after-free bugs, buffer overflows, integer overflows that wrap silently, data races, null pointer dereferences, or double frees. In small programs these mistakes are relatively easy to find and fix. In large systems with millions of lines of code maintained by dozens of engineers over many years, they accumulate into a persistent source of crashes, security vulnerabilities, and maintenance burden.

Most mature C codebases have several characteristics that make them hard to change:

- They have grown organically over many years or decades, with architectural decisions from the early days still constraining current design choices.
- They depend on global mutable state, complex callback chains, manual reference counting, arena allocators, and other patterns that worked for the original team but are now poorly documented.
- Their test coverage is uneven, with critical paths sometimes well-covered and many edge cases only discovered through production incidents.

- The ABI has hardened over time, with external consumers depending on symbol names, struct layouts, and error conventions that cannot be changed without breaking compatibility.
- Performance characteristics have been tuned empirically, often with hand-written optimizations whose rationale is no longer clear but whose removal might cause regression.

Any modernization effort must account for these realities. You are not starting from a clean slate with perfect requirements. You are working inside a living system that people depend on right now.

Defect Patterns in Mature C Codebases

Memory safety vulnerabilities in C fall into several well-known categories, each of which can lead to crashes, data corruption, or arbitrary code execution:

Buffer overflows occur when a program writes beyond the bounds of an allocated buffer. This is one of the most common and exploitable vulnerability classes in C. It happens when size calculations are wrong, when input validation is incomplete, or when off-by-one errors slip past review.

Use-after-free bugs happen when a pointer is dereferenced after the memory it points to has been freed. These are particularly insidious because they may appear to work most of the time and only manifest under specific timing conditions or memory allocation patterns that change between builds.

Double free errors occur when the same memory region is freed twice, corrupting the allocator's internal data structures and potentially allowing an attacker to redirect execution.

Null pointer dereferences crash programs when a function returns NULL to indicate an error and the caller does not check before using the result.

Integer overflows can cause incorrect size calculations that lead to buffer overflows, or can wrap around to create negative values where unsigned is expected, triggering unexpected behavior in comparisons and loops.

Data races arise when multiple threads access shared mutable state without proper synchronization, leading to unpredictable results that are notoriously difficult to reproduce and debug.

These patterns are not theoretical concerns limited to academic examples. They are the dominant source of serious security vulnerabilities at major technology companies. Microsoft reported that approximately seventy percent of the CVEs they patched over a twelve-year period were memory safety bugs [1]. Google's Chromium team found similar proportions in their analysis of high and critical severity bugs since 2015, with use-after-free alone accounting for roughly half of memory safety issues [2].

The cost is not just security. These defects cause production incidents that require emergency patches, lead to downtime, erode user trust, and consume engineering time that could be spent on new features instead of defensive fixes.

The Safety Promise: Memory, Concurrency and Types

Rust addresses the defect patterns described above through a combination of compile-time checks enforced by its ownership system, borrow checker, and type system. These mechanisms prevent entire classes of bugs at compile time without requiring runtime overhead like garbage collection or bounds checking on every array access in optimized code.

Ownership ensures that every piece of allocated memory has exactly one owner responsible for freeing it. When the owner goes out of scope, the memory is automatically deallocated. This eliminates double free and use-after-free errors because the compiler guarantees that no other reference to that memory can be used after it is freed.

Borrowing allows you to pass references to data without transferring ownership, but with strict rules: you can have either one mutable reference or any number of immutable references at a time, never both. This prevents data races at compile time because two threads cannot simultaneously hold mutable access to the same data through safe Rust code alone.

Lifetimes track how long references are valid, ensuring that no reference outlives the data it points to. The borrow checker enforces these constraints statically, catching dangling pointer bugs before the program ever runs.

The type system provides additional safety guarantees: there is no implicit null in safe Rust (you must explicitly use Option for nullable values), integer overflow behavior is well-defined and can be checked or wrapped as appropriate, enums with discriminants prevent invalid state combinations, and traits provide a way to express generic constraints that are checked at compile time.

For concurrency specifically, Rust's Send and Sync marker traits allow the compiler to verify that types can be safely transferred between threads or shared across thread boundaries. Types containing raw pointers or interior mutability without synchronization are automatically marked as not thread-safe, preventing accidental data races that would be undefined behavior in C.

These guarantees are not aspirational or dependent on programmer discipline. They are enforced by the compiler. If your Rust code compiles in safe mode (without unsafe blocks), it is guaranteed to be free of data races and memory safety violations according to Rust's formal specification [4].

Where Rust Gains Are Real and Measurable

Rust does not deliver equal benefit across every part of a C system. The gains are concentrated in specific areas where C's lack of safety guarantees creates the most risk:

Security-sensitive components such as protocol parsers, cryptographic code, authentication handlers, and input validation logic are prime candidates. These modules process untrusted data and are attractive targets for attackers who exploit memory corruption bugs to achieve remote code execution. Rewriting a parser in Rust eliminates buffer overflow and use-after-free vulnerabilities that might have existed in the C version, without requiring runtime checks or address sanitizers.

Concurrency-heavy components benefit from Rust's compile-time race detection. A multithreaded event loop, connection pool, or worker queue that was previously protected by manual mutex management can be redesigned using Rust's synchronization primitives with stronger guarantees about lock ordering and data access patterns. The compiler will reject designs that could cause races, catching bugs that might have taken months to surface in production.

Complex ownership domains where resources like file descriptors, network sockets, database connections, or memory pools are shared between many components can be simplified using Rust's RAII pattern and smart pointers. Instead of tracking reference counts manually across dozens of call sites, ownership is expressed explicitly in the type system and enforced by Drop implementations that run deterministically when values go out of scope.

New functionality added to an existing C system can be implemented in Rust from the start, wrapped behind a stable C-compatible API. This prevents new memory safety bugs from accumulating in the codebase while keeping existing C consumers unchanged. Google's Android team has used this approach extensively, achieving a reduction in memory safety vulnerability density of over one thousand times for new Rust code compared to legacy C/C++ [5].

Measurable outcomes from production Rust adoption support these claims. Dropbox reported that their Rust-based sync engine rewrite eliminated entire classes of consistency bugs that had plagued the original Python/Go implementation [6]. Google found that Rust changes in Android have a four times lower rollback rate and spend twenty-five percent less time in code review than comparable C++ changes, suggesting that the compile-time safety checks reduce defects early enough to improve developer productivity as well as security [5].

The Hidden Costs of Migration

Rust adoption is not free. Organizations that underestimate the costs often find themselves stalled mid-migration with frustrated engineers and no clear path forward. Understanding these costs upfront helps you plan realistically.

Learning curve is significant for experienced C programmers. Ownership, borrowing, lifetimes, traits, and the borrow checker represent a fundamentally different mental model from manual memory management. Engineers who have written C professionally for twenty years must unlearn habits that served them well and adopt new patterns. Expect productivity to dip initially as the team learns the language, then recover and potentially exceed previous levels once Rust idioms become familiar.

Build system integration adds complexity. Mixing Cargo with Make, CMake, or other build systems requires careful coordination of compilation flags, linking order, dependency management, and cross-compilation support. Generated bindings must be kept in sync with C headers, and changes to either side can break the other in subtle ways that are not immediately obvious from error messages.

FFI boundaries create their own risks. While Rust's FFI is well-designed, it is inherently unsafe by necessity: crossing into C code means leaving the safety guarantees behind. Panics that escape across FFI boundaries cause

process aborts unless carefully handled, type mismatches in struct layouts lead to undefined behavior, and ownership assumptions about who frees what memory must be documented explicitly since neither compiler can verify them.

Ecosystem maturity varies by domain. Rust's standard library and crate ecosystem are excellent for general systems programming but may lack specialized libraries available in C for certain domains like embedded platforms, real-time operating systems, or niche hardware interfaces. You may need to write your own bindings or wrappers, adding maintenance burden.

Binary size and startup time can increase. A minimal C program might produce a few kilobytes of executable code, while even a simple Rust binary typically includes hundreds of kilobytes of standard library code unless carefully optimized with features like stripping, LTO, and no-std configurations. For embedded systems or latency-sensitive applications this matters.

Organizational friction is real. Introducing a new language into an organization creates questions about ownership: which team maintains the Rust code, how do we review mixed-language pull requests, what training is needed, how do we onboard new engineers who know C but not Rust? Without clear answers these questions create confusion and slow down development.

When Staying in C Is the Right Decision

Rust is not universally better than C for every use case. There are situations where migrating to Rust would be a poor engineering decision:

Simple, stable components with no history of bugs may not justify migration cost. If a module has been correct and unchanged for years, processes trusted input only, and has comprehensive tests, the risk from keeping it in C is low compared to the risk introduced by rewriting it.

Highly optimized assembly-adjacent code where performance was achieved through platform-specific tricks, hand-tuned cache layouts, or inline assembly may be difficult to replicate in Rust without losing the original optimizations or introducing unsafe code that defeats the safety purpose.

Components with hard real-time constraints on exotic hardware where the Rust toolchain does not yet provide adequate support may be better left in C until ecosystem maturity catches up. The embedded Rust ecosystem has grown significantly but still lags behind C for many microcontroller families and RTOS platforms [7].

Code that depends heavily on preprocessor macros, complex conditional compilation, or platform-specific ABI details that bindgen cannot handle cleanly may require so much manual rewriting that the migration becomes effectively a new implementation rather than a translation. In such cases you might be better off wrapping the existing C code with safe interfaces instead of rewriting it.

Legacy systems where external consumers depend on exact binary compatibility and any change to struct layouts, symbol names, or calling conventions would break thousands of downstream projects may need to remain in C indefinitely. You can still add Rust components alongside them but replacing core libraries might not be feasible.

The key insight is that modernization should be risk-based and value-driven. Components with high security exposure, complex ownership patterns, concurrency requirements, or a history of defects are good migration candidates. Components that are simple, stable, well-tested, and low-risk are often better left alone. A thoughtful modernization strategy uses Rust where it delivers the most benefit while accepting that some C code will remain in the system for the foreseeable future.

The next chapter establishes the technical foundation you need to understand how C and Rust differ at a semantic level, so you can make informed decisions about what to migrate and how to design safe interoperability boundaries between the two languages.

Chapter 2: C and Rust Side by Side

Compilation, Linking and the Object Model

Both C and Rust follow the same fundamental compilation model that has been standard for systems programming since the early days of Unix: source files are compiled into object files containing machine code and symbol tables, then a linker combines those object files with libraries to produce an executable or shared library. This common foundation is what makes interoperability possible in the first place.

In C you typically invoke gcc or clang to compile each .c file into a .o object file, then link everything together:

```
cc -c module.c -o module.o
cc main.o module.o -lm -o myprogram
```

Rust uses rustc for compilation and Cargo as its build system and package manager. A single Rust crate (the unit of compilation) can contain many source files but is compiled as a whole:

```
rustc src/main.rs --crate-type bin -o myprogram
# or via Cargo:
cargo build --release
```

The key insight for C programmers is that Rust produces standard ELF, Mach-O, or PE binaries depending on the target platform, with standard symbol tables and section layouts. A Rust dynamic library (.so, .dylib, or .dll) can be loaded and called from C code exactly like any other C library, provided the exported symbols follow the correct ABI conventions.

Both compilers use LLVM as their backend (rustc exclusively, clang as an alternative to gcc), which means they share similar optimization passes, instruction selection strategies, and code generation quality. This is why Rust programs can achieve performance comparable to well-written C: at the machine code level there is often very little difference.

The linking model differs in one important respect: Cargo manages dependencies transitively and builds them automatically from source or cached artifacts, while traditional C build systems require explicit dependency declarations in Makefiles or CMakeLists.txt files. When integrating Rust into a C project you will need to bridge these two worlds, either by invoking Cargo from your existing build system or by using tools like the cc crate to compile C code from within Cargo build scripts.

Primitive Types and Integer Semantics

Rust's primitive types are broadly similar to C's but with important differences in size guarantees and behavior:

- **bool** – 1 byte; true or false; not implicitly convertible to integers
- **u8, u16, u32, u64, u128** – fixed-width unsigned integers with exact sizes
- **i8, i16, i32, i64, i128** – fixed-width signed integers with exact sizes
- **usize, isize** – pointer-sized integers (platform-dependent; 8 bytes on 64-bit)
- **f32, f64** – IEEE 754 floating point; identical layout to C float/double

The critical differences from C are:

Rust integer types have guaranteed sizes across all platforms. There is no int with variable width depending on the architecture. If you need a pointer-sized integer use usize or isize explicitly. This eliminates an entire class of portability bugs that plague C code when moving between 32-bit and 64-bit systems.

Rust bool is a distinct type, not defined as an integer like in C where _Bool or int serves the purpose. You cannot use an integer directly where a bool is expected without explicit conversion, which prevents accidental truthiness bugs from comparing pointers or sizes incorrectly.

Integer overflow behavior differs significantly. In C signed integer overflow is undefined behavior, which allows compilers to optimize aggressively but can cause subtle bugs when overflow actually occurs. Rust provides three modes: checked arithmetic that returns Option and never overflows (wrapping_add), wrapping arithmetic that silently wraps like unsigned C integers (saturating_add), and saturating arithmetic that clamps at min/max values. In debug builds most arithmetic operators panic on overflow; in release builds they wrap by default for performance but you can opt into checked behavior explicitly [8].

This means when porting C code to Rust you must decide how each potentially overflowing operation should behave rather than relying on implicit wrapping or undefined behavior. This is not just pedantry: it forces you to confront assumptions about numerical ranges that may have been wrong in the original C implementation.

Pointers Versus References: What Actually Changes

This is where C programmers encounter their first major conceptual shift. Rust distinguishes between three different concepts that are all expressed as pointers in C:

Raw pointers (*const T and *mut T) behave exactly like C pointers: they can be null, they can dangle, dereferencing them is unsafe, and the compiler does not track their validity. You use raw pointers exclusively for FFI with C code or when implementing low-level data structures inside unsafe blocks.

References (&T for immutable, &mut T for mutable) are guaranteed to be non-null and always point to valid memory of the correct type. The borrow checker enforces these guarantees at compile time by tracking where references live and ensuring they never outlive the data they point to. You cannot create a dangling reference in safe Rust code because the compiler will reject any program where a reference could become invalid.

Smart pointers (Box, Rc, Arc) are heap-allocated wrappers that manage ownership and deallocation automatically. Box is analogous to malloc with automatic free when it goes out of scope. Rc provides reference counting for single-threaded shared ownership, while Arc provides atomic reference counting for multi-threaded contexts.

For C programmers the mental translation is:

- Where you used struct foo * in C for “pointer that might be null”, use Option<&foo> or Option<Box> in Rust
- Where you used const struct foo * for “read-only access to someone else’s data”, use &foo
- Where you used struct foo * for “I own this and will free it later”, use Box

The borrow checker rules that govern references can feel restrictive at first but they are the mechanism that prevents entire classes of bugs. The two fundamental rules are:

1. At any given time, you may have either one mutable reference (&mut T) to a value or any number of immutable references (&T), but not both.
2. References must always be valid; they cannot outlive the data they point to.

These rules prevent data races (rule 1 ensures no concurrent mutation without synchronization) and use-after-free bugs (rule 2 ensures memory is not accessed after deallocation). In C you enforce these properties manually through discipline and code review. In Rust the compiler enforces them automatically.

Structs, Unions, Enums and Data Layout

Rust structs are similar to C structs but with different default layout rules:

```
struct Point {
    x: f64,
    y: f64,
}
```

By default Rust uses repr(Rust) which allows the compiler to reorder fields for optimization. For FFI compatibility you must use repr(C) to guarantee C-compatible field ordering and padding:

```
#[repr(C)]
struct Point {
    x: f64,
    y: f64,
}
```

With repr(C) the struct layout matches what a C compiler would produce: fields in declaration order with padding inserted as needed for alignment [9]. This is essential when passing structs across FFI boundaries because mismatched layouts cause silent memory corruption.

Rust unions are identical to C unions in behavior but require unsafe to access:

```rust
#[repr(C)]
union Value {
    integer: i32,
    float: f32,
}
```

You must use unsafe blocks to read or write union fields because the compiler cannot track which variant is currently active. For most use cases Rust's enums provide a safer alternative with discriminants that prevent reading the wrong variant.

Rust enums are more powerful than C enums. A C enum is essentially a named integer constant:

```c
enum Color { RED, GREEN, BLUE };
```

A Rust enum can carry data in each variant, making it a discriminated union or tagged union:

```rust
enum Color {
    Red,
    Green,
    Blue,
    Rgb(u8, u8, u8),
}
```

This pattern is invaluable for representing state machines, error types with context, and protocol messages where different variants carry different payloads. When used across FFI boundaries enums must be marked with repr(C) or a specific integer representation to ensure compatible layout.

Ownership, Moves and Copy Semantics

Ownership is the cornerstone of Rust's safety model and the concept that requires the most adjustment for C programmers. In C you reason about ownership informally: "this function allocates memory that the caller must free" or "this pointer is borrowed and will remain valid as long as the parent object exists." These conventions live in documentation and comments, not in the type system.

In Rust ownership is explicit and enforced by the compiler. Every value has exactly one owner variable. When that variable goes out of scope, the value is dropped (deallocated or cleaned up). There are two ways to transfer or share values:

Move semantics transfer ownership from one variable to another. After a move the original variable is no longer valid:

```
let s1 = String::from("hello");
let s2 = s1;  // s1 is moved to s2, s1 is no longer usable
// println!("{}", s1);  // compile error: value borrowed here after move
```

This prevents double-free bugs because the compiler ensures that only one variable owns the data at a time. When you pass a non-trivial value to a function in Rust, ownership moves into the function by default unless you explicitly pass a reference instead.

Copy semantics apply to small, trivially copyable types like integers, floats, booleans, and tuples of copy types. These are copied on assignment rather than moved:

```
let x = 42;
let y = x;  // x is copied, both x and y are valid
println!("{} {}", x, y);
```

You can mark your own types as Copy by implementing the Copy trait (which requires also implementing Clone):

```
#[derive(Copy, Clone)]
struct Point { x: f32, y: f32 }
```

For C programmers the practical implication is: when you pass data to a Rust function, decide whether the function should take ownership (take the value by value), borrow it temporarily (take a reference &T or &mut T), or clone it (call .clone() explicitly). This decision is explicit in the function signature rather than implied by convention.

Lifetimes and Borrowing for C Programmers

Lifetimes are Rust's mechanism for tracking how long references remain valid. In C you manage this manually: you ensure that a pointer is not used after its backing memory is freed, typically through scope discipline or reference counting. In Rust the compiler tracks lifetimes automatically using lifetime annotations on function signatures.

Consider a function that returns a pointer into a string argument:

```c
// C version - potentially dangerous
const char* find_substring(const char* haystack, const char* needle) {
    return strstr(haystack, needle);
}
```

In C this works as long as the caller ensures haystack outlives the returned pointer. If haystack is a local buffer that goes out of scope, the returned pointer dangles. The compiler cannot check this.

The Rust equivalent uses lifetime annotations to express the constraint explicitly:

```rust
fn find_substring<'a>(haystack: &'a str, needle: &str) -> Option<&'a str> {
    haystack.find(needle)
}
```

The lifetime parameter 'a says: the returned reference lives as long as haystack. If you try to use the result after haystack goes out of scope, the compiler rejects it. This is not runtime overhead; it is a compile-time check that guarantees no dangling references can exist in safe Rust code.

For most functions the compiler can infer lifetimes automatically using standard elision rules. You only write explicit lifetime annotations when the relationship between input and output lifetimes is ambiguous, typically when multiple borrowed inputs could potentially be the source of an output reference.

When designing FFI boundaries you must understand that C has no concept of Rust lifetimes. A C caller cannot express or enforce lifetime constraints. This means any function exposed to C through FFI must be designed so that its safety does not depend on lifetime relationships that C code cannot track.

Typically this means returning owned data (allocated with malloc or a Rust allocator) rather than borrowed references into Rust-internal memory.

Stack, Heap and Allocation Patterns

Rust uses the same stack and heap model as C: small values are allocated on the stack automatically when variables are declared, and larger or dynamically-sized values are allocated on the heap explicitly. The key difference is how heap allocation is managed.

In C you call malloc, calloc, realloc, and free manually. Forgetting to free causes memory leaks. Freeing too early or twice causes use-after-free or double-free bugs. Using freed memory causes undefined behavior.

In Rust heap allocation happens through Box, Vec, String, and other smart pointer types that automatically deallocate when they go out of scope:

```
let v = vec![1, 2, 3];  // allocates on heap
// v is automatically dropped here, memory freed
```

The Drop trait defines cleanup logic that runs deterministically when a value goes out of scope. This is similar to RAII in C++ but built into the language core. For FFI with C code you can implement Drop to release C-allocated resources:

```
struct FileHandle {
    fd: libc::FILE*,
}

impl Drop for FileHandle {
    fn drop(&mut self) {
        if !self.fd.is_null() {
            unsafe { libc::fclose(self.fd) };
        }
    }
}
```

For allocation across FFI boundaries you have two main options: let Rust allocate and C free using a provided deallocator function, or let C allocate and Rust wrap the result in a safe type with Drop handling cleanup. The choice depends on which side should own the resource and which allocator is more convenient for the use case.

RAII, Drop and Resource Management

RAII (Resource Acquisition Is Initialization) ties resource lifetime to object lifetime: acquire a resource when constructing an object, release it when the object is destroyed. Rust implements this through the Drop trait, which defines cleanup code that runs automatically when a value goes out of scope.

This pattern eliminates entire classes of resource leaks in C where exceptions, early returns, or complex control flow can cause cleanup code to be skipped:

```c
// C version - error-prone
void process_file(const char* path) {
    FILE* f = fopen(path, "r");
    if (!f) return;

    char* buffer = malloc(4096);
    if (!buffer) { fclose(f); return; }

    // ... many lines of processing with possible early returns ...

    free(buffer);
    fclose(f);
}
```

```rust
// Rust equivalent - cleanup is automatic
use std::fs::File;
use std::io::{BufReader, BufRead};

fn process_file(path: &str) -> Result<()> {
    let file = File::open(path)?;
    let mut reader = BufReader::new(file);
    let mut line = String::new();

    while reader.read_line(&mut line)? > 0 {
        // process line
        line.clear();
        // if we return early here, file is still closed automatically
    }

    Ok(())
}
```

For C resources wrapped in Rust you implement Drop to ensure cleanup happens regardless of how the enclosing scope exits. This is one of the most

practical benefits of introducing Rust into a C system: wrapping C resource handles in RAII types prevents leaks that were previously managed through fragile manual cleanup chains.

Mutability, Aliasing and Interior Mutability

Rust's default stance on mutability differs from C. In C any pointer can be cast to mutable or immutable with minimal friction, and aliasing rules are mostly advisory (the restrict keyword helps but is not universally used). Rust separates mutable and immutable access at the type level: &T is read-only, &mut T allows modification, and you cannot have both kinds of references to the same data simultaneously.

This prevents accidental mutation bugs where code that was supposed to be read-only modifies shared state. It also enables compiler optimizations because the compiler knows that immutable references cannot alias mutable state.

Interior mutability is a pattern where data can be mutated through an apparently immutable reference using special wrapper types like Cell, RefCell, Mutex, and RwLock. These types use runtime checks to enforce borrowing rules dynamically instead of at compile time:

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

struct Counter {
    value: RefCell<i32>,
}

impl Counter {
    fn increment(&self) {
        *self.value.borrow_mut() += 1;
    }
}
```

Interior mutability is necessary when you need to mutate data that is shared behind an immutable reference, such as caching computed results or implementing observer patterns. However it comes with runtime overhead and potential panics if borrowing rules are violated at runtime (RefCell::borrow_mut panics if the data is already mutably borrowed). Use interior mutability sparingly and only when compile-time borrowing rules cannot express your access pattern.

Error Handling: Return Codes Versus Result

C error handling typically uses one of three patterns: return codes with errno, sentinel values like NULL or -1, or output parameters that indicate success or failure. All of these approaches share a weakness: they are easy to ignore. A caller can forget to check the return value and the compiler will not complain.

Rust uses the Result<T, E> enum for recoverable errors:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

A function that might fail returns Result instead of the success value directly:

```
fn parse_int(s: &str) -> Result<i32, ParseIntError> {
    s.parse()
}
```

The caller must handle both cases explicitly using match or the ? operator for propagation:

```
let n = parse_int("42")?;  // returns Err from this function if parsing fails
```

This makes error handling mandatory rather than optional. A function that ignores a Result will not compile, which prevents entire classes of bugs where errors are silently dropped and cause crashes or corrupted state downstream.

For FFI with C code Rust's Result is not directly usable because C has no equivalent type. You must translate between Result and C-compatible error conventions at the boundary: return an integer error code and set errno, return a boolean success flag with output parameters, or use opaque error objects that C can query for details. Chapter 4 covers these patterns in detail.

Generics, Traits and Pattern Matching

Rust generics work similarly to C templates but are monomorphized at compile time rather than generating shared code. Each use of a generic function with a different type produces a separate compiled version:

```
fn max<T: PartialOrd>(a: T, b: T) -> T {
    if a >= b { a } else { b }
}
```

The trait bound T: PartialOrd constrains the generic to types that support comparison. This is similar to C++ template constraints but checked more aggressively by the compiler.

Traits are Rust's mechanism for defining shared behavior across types, analogous to interfaces in other languages but with more flexibility. A type can implement multiple traits, and traits can have default implementations:

```
trait Display {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result;
}

impl Display for Point {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "({}, {})", self.x, self.y)
    }
}
```

Traits are essential for writing generic code that works with different concrete types while maintaining type safety. For FFI purposes traits have no direct equivalent in C; trait-based abstractions must be resolved to concrete types before crossing the language boundary.

Pattern matching is Rust's powerful alternative to switch statements and if-else chains. It allows you to destructure enums, tuples, and structs concisely while ensuring all cases are handled:

```
match result {
    Ok(value) => println!("Got: {}", value),
    Err(e) if e.kind() == std::io::ErrorKind::NotFound => {
        println!("File not found");
    }
    Err(e) => println!("Error: {}", e),
}
```

The compiler checks that all enum variants are covered, preventing bugs where a new variant is added but existing match statements are not updated. This is particularly valuable for error handling and protocol parsing where missing a case can cause subtle failures.

Concurrency Primitives and Memory Ordering

Rust's concurrency model builds on the same threading abstractions as C (pthreads on Unix, Windows threads on Windows) but adds compile-time guarantees about data race freedom through the Send and Sync marker traits.

Send means a type can be safely transferred to another thread. Most types are Send by default except those containing non-thread-safe interior mutability like RefCell or raw pointers without synchronization.

Sync means a type can be safely shared between threads through references (&T is Send). A type T is Sync if and only if &T is Send. Types with unsynchronized mutable state are not Sync.

These traits are automatically derived based on the type's fields. If all fields of a struct are Send, the struct is Send. If you need to override this behavior (for example when wrapping an audited C library handle that you know is thread-safe), you can implement these traits manually using unsafe impl [10].

Rust provides standard synchronization primitives: Mutex for mutual exclusion, RwLock for read-write locking, Condvar for condition variables, and atomic types (AtomicBool, AtomicUsize, etc.) with well-defined memory ordering semantics matching C11 atomics. Channels (mpsc::channel) provide message passing between threads without shared mutable state.

The key advantage over C is that the type system prevents data races in safe code: you cannot share a &mut T across threads because mutable references are not Sync, and you cannot send a non-Send type across thread boundaries because the compiler rejects it. This eliminates entire classes of concurrency bugs that require expensive runtime testing to catch in C.

The Role and Meaning of unsafe

The unsafe keyword in Rust marks code where the compiler cannot verify safety guarantees. Inside an unsafe block you can:

- Dereference raw pointers
- Call unsafe functions (including most FFI calls)
- Access or modify mutable static variables
- Implement unsafe traits
- Access union fields

Crucially, unsafe does not mean "this code is definitely unsafe." It means "the programmer asserts that this code maintains safety invariants that the compiler cannot check." Well-written unsafe code is safe because it upholds its contracts. Poorly written unsafe code causes undefined behavior just like buggy C code.

For C/Rust interoperability unsafe is unavoidable: all FFI calls cross into territory where Rust's guarantees do not apply, so they must be wrapped in unsafe blocks. The goal is to minimize the amount of unsafe code and encapsulate it behind safe interfaces so that most of your Rust codebase remains fully verified by the borrow checker.

A common pattern is to write a thin unsafe layer that calls C functions directly, then wrap those calls in safe Rust functions that validate inputs, check error conditions, and manage resource lifetimes:

```
// Unsafe FFI boundary
unsafe extern "C" {
    fn c_library_function(input: *const c_char) -> c_int;
}

// Safe wrapper
pub fn library_function(input: &str) -> Result<i32, LibraryError> {
    let c_str = CString::new(input)?;
    let result = unsafe { c_library_function(c_str.as_ptr()) };
    if result < 0 {
        Err(LibraryError::from_errno(result))
    } else {
        Ok(result as i32)
    }
}
```

The safe wrapper ensures that callers cannot misuse the underlying C function: they must pass valid UTF-8 strings, and errors are returned as typed Result values instead of opaque integers. This encapsulation pattern is central to successful C/Rust integration and will be explored in depth throughout the book.

Understanding these semantic differences between C and Rust is essential before attempting any migration. The next chapter shows you how to assess an existing C codebase systematically to identify which components are good candidates for modernization and which should remain in C.

Chapter 3: Assessing Your C Codebase

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Inventorying Components and Dependencies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Mapping Ownership Assumptions and Lifetime Contracts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Locating Undefined Behavior and Memory Risks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Measuring Test Coverage and Reliability Baselines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Classifying Modules by Risk, Value and Complexity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Identifying Migration Candidates: The Decision Matrix

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Building the Modernization Roadmap

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Chapter 4: FFI Foundations: C and Rust Talking to Each Other

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

The ABI Contract: extern “C” and Symbol Visibility

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Data Layout Compatibility with repr(C)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Passing Primitives, Pointers and Arrays

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Strings Across the Boundary: CString and CStr

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Structs, Unions, Enums and Opaque Handles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Function Pointers and Callbacks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Ownership Transfer at FFI Boundaries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Error Propagation Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Panic Handling and Unwinding Across Languages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Thread Safety and Global State

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Chapter 5: Build Systems and Tooling Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Cargo in a Make and CMake World

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

The cc Crate and Building C from Cargo

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Bindgen: Generating Bindings Automatically

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Build Scripts and pkg-config Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Cross-Compilation and Target Triples

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Static Versus Dynamic Linking Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Reproducible Builds and Dependency Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Chapter 6: Migration Pattern 1: Wrapping C Libraries Safely

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

When Wrapping Is Better Than Rewriting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Designing Opaque Handle Interfaces

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

RAII Wrappers for Resource Handles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Lifetime-Aware APIs Around Raw Pointers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Validated Constructors and Error Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Encapsulating unsafe: The Safety Boundary Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Complete Example: Wrapping a C Configuration Parser

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Chapter 7: Migration Pattern 2: Replacing Leaf Modules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Identifying Replaceable Leaf Components

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Designing Drop-In Replacements via Stable ABIs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Migrating String Utilities and Formatters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Migrating Data Serialization Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Migrating Protocol Parsers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Complete Example: Replacing a C Parser with Rust

Chapter 8: Migration Pattern 3: Strangler and Sidecar Approaches

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

The Strangler Pattern for Long-Lived Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Process Boundary Separation via IPC

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Sidecar Services in Rust

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Plugin Architectures with C-Compatible APIs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Gradual Traffic Migration and Feature Flags

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Complete Example: Extracting a Subsystem Behind an API

Chapter 9: Creating Safe Abstractions Over Unsafe C APIs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Newtypes and Domain-Specific Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Typestate Patterns for Stateful Resources

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Smart Pointer Wrappers for Ownership Tracking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Synchronization Wrappers for Thread Safety

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Invariants and Encapsulated unsafe Blocks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Writing Auditable Safety Comments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Complete Example: Safe Wrapper Around a C Memory Pool

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Chapter 10: Testing Behavioral Equivalence During Migration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Characterization Tests for Legacy Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Differential Testing of C and Rust Implementations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Protocol Corpus and Golden Testing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Fuzzing Mixed-Language Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Concurrency and Stress Testing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

ABI Compatibility and Regression Testing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Complete Example: Test Harness for Migrated Parser

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Chapter 11: Performance Engineering and Verification

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Establishing Reproducible C Baselines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Benchmarking Methodology and Pitfalls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Comparing Latency, Throughput and Memory Use

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Allocation Behavior and Cache Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

When Rust Is Slower (and Why)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Optimization Without Sacrificing Safety

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Complete Example: Before/After Performance Comparison

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Chapter 12: Security Modernization and unsafe Governance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Vulnerability Classes Rust Actually Prevents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

What Rust Does Not Fix Automatically

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Threat Modeling for Mixed-Language Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Supply Chain Security and Dependency Governance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Auditing and Reducing unsafe Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Organizational Policies for unsafe Review

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

SBOM Generation and Reproducible Builds

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Chapter 13: Concurrency Modernization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Thread Safety Patterns in Mature C Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Send and Sync: What They Mean for C Interop

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Migrating Lock-Based Components

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Channels and Message Passing Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Atomics and Memory Ordering Across Languages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Async Runtimes: When They Make Sense

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Complete Example: Migrating a Threading Component

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Chapter 14: Deployment, Operations and Organizational Adoption

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Packaging Mixed-Language Applications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Observability Across Language Boundaries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Debugging Production Issues in C and Rust

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Crash Reporting and Core Dump Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Training C Engineers for Rust

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Code Review Practices for Mixed Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Decision Frameworks: Migrate, Wrap, Rewrite, or Keep

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Common Failed Migration Approaches and Anti-Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Production Checklists

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Chapter 15: Case Studies in Realistic Migration Scenarios

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Case Study 1: Migrating a Binary Protocol Parser

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Case Study 2: Wrapping a C Cryptographic Library

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Case Study 3: Migrating a Multithreaded Event Daemon

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Case Study 4: Embedded System Firmware Migration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Case Study 5: Command-Line Utility Rewrite

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

Conclusion: A Roadmap for Evolving C Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/fromctorust.