https://simplifycpp.org

Rust
Ownership & Borrowing

Zero-Cost Memory Safety Without Garbage Collection

Prepared by Ayman alheraki

Rust Ownership & Borrowing

Zero-Cost Memory Safety Without Garbage Collection

`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

Preface

Purpose

This mini-booklet explains one thing only: **Rust ownership and borrowing**. It is written for experienced programmers coming from `C/C++`, `Java`, `Go`, `C#`, `Swift`, and similar languages who want a precise answer to:

- What does Rust enforce that other mainstream languages do not?
- How does that enforcement prevent real memory and concurrency bugs?
- Why is this safety achieved with **no garbage collector** and typically **no runtime overhead**?

The Problem Rust Targets

Most serious low-level failures repeatedly come from the same classes of mistakes:

- **Use-after-free** (dangling pointers / stale references),
- **Double free** (two owners freeing the same allocation),
- **Aliasing + mutation** (unexpected shared writes),

- **Data races** (unsynchronized concurrent access with a writer),
- **Lifetime mismatches** (references outliving the data they point to).

Historically, teams used discipline and tools (reviews, tests, sanitizers, fuzzing) to reduce these risks. Rust changes the default: it makes many of these bug classes **unrepresentable in safe code**.

What Rust Enforces

Rust is built around two compile-time mechanisms:

- **Ownership:** each value has exactly one owning binding at a time.
- **Borrowing:** references are temporary and must always be valid.

Borrowing follows a strict rule that blocks invalid aliasing:

Either many readers (`&T`) or one writer (`&mut T`), but not both at the same time.

This rule is checked by the compiler (the borrow checker). It is not a runtime feature.

Example 1: Moves Prevent Double Free

A heap-owning value is typically **moved** on assignment. The old name becomes unusable, preventing two frees.

```rust
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;              // move

    // println!("{s1}");       // error: moved value
    println!("{s2}");
}
```

If you want to keep using the original value, you must be explicit (clone):

```rust
fn main() {
    let s1 = String::from("hello");
    let s2 = s1.clone();      // explicit copy of heap data
    println!("{s1} {s2}");
}
```

Example 2: Borrowing for Shared Reads

Read-only sharing is safe and cheap: references do not allocate and do not copy the data.

```rust
fn main() {
    let s = String::from("hello");
    let a = &s;
    let b = &s;
    println!("{a} {b}");
}
```

Example 3: Exclusive Borrowing for Mutation

Mutation requires exclusive access. This blocks bugs caused by mutating through one alias while others still observe.

```rust
fn main() {
    let mut s = String::from("hello");
    let w = &mut s;
    w.push_str(" world");
    println!("{w}");
}
```

This is rejected because it mixes readers with a writer in the same time window:

```
fn main() {
    let mut s = String::from("hello");
    let r = &s;                 // shared borrow
    let w = &mut s;            // error: cannot borrow as mutable while borrowed as immutable
    println!("{r} {w}");
}
```

Example 4: Lifetimes as Compile-Time Contracts

Rust ties a reference to the lifetime of the data it refers to. Returning a reference to a local value is rejected at compile time.

```
fn bad_ref() -> &String {
    let s = String::from("temp");
    &s                          // error: returns reference to a value that will be dropped
}

fn main() {}
```

The safe solution is to return ownership instead of a reference:

```
fn make_string() -> String {
    let s = String::from("owned");
    s                           // move out (return ownership)
}

fn main() {
    let s = make_string();
    println!("{s}");
}
```

Why This Is Zero-Cost (No GC)

Rust's safety is primarily enforced at compile time:

- **No garbage collector:** destruction is deterministic when the owner goes out of scope.
- **No runtime borrow checking in normal safe code:** the compiler proves validity and exclusivity.
- **No hidden ref-counting by default:** shared ownership is explicit (`Rc`, `Arc`).

You pay only for what you choose to use.

Windows Workflow

All examples are cross-platform and run the same on Windows. Typical PowerShell commands:

```
cargo new borrow_ownership
cd borrow_ownership
cargo run
cargo build --release
```

Scope

This booklet focuses strictly on:

- Ownership and moves,
- Borrowing rules (`&T` and `&mut T`),
- Lifetimes as compile-time reasoning,

- Practical patterns that map to real bug prevention.

It deliberately avoids broader Rust topics (modules, traits, async, macros, crates) except where needed to explain ownership and borrowing precisely.

Ayman Alheraki

The Memory Problem in Modern Programming

1.1 Why Memory Bugs Still Exist

Modern software still fails for one core reason: **the program's view of object lifetime often diverges from reality**. This happens whenever code can: (1) keep an address/reference beyond the lifetime of the object, (2) alias the same object through multiple paths and mutate it unexpectedly, (3) race across threads without enforced exclusivity.
The recurring bug families are well-known:

- **Use-after-free:** a pointer/reference outlives its allocation.
- **Double free:** two owners free the same allocation.
- **Out-of-bounds access:** indexing or pointer arithmetic crosses object boundaries.
- **Uninitialized reads:** data is observed before being written.
- **Data races:** concurrent access with at least one writer and no synchronization.

These persist because most mainstream languages historically optimized for **expressiveness** and **performance** without making lifetime and aliasing constraints **provable by default** at compile time.

Example: Use-after-free (C)

```c
#include <stdio.h>
#include <stdlib.h>

static int* bad(void) {
    int *p = (int*)malloc(sizeof(int));
    *p = 123;
    free(p);
    return p;          // dangling pointer
}

int main(void) {
    int *x = bad();
    printf("%d\n", *x); // undefined behavior
    return 0;
}
```

Example: Data race (C/C++)

```cpp
#include <thread>
#include <iostream>

int main() {
    int x = 0;

    std::thread t1([&]{ for (int i = 0; i < 1'000'000; ++i) ++x; });
    std::thread t2([&]{ for (int i = 0; i < 1'000'000; ++i) ++x; });

    t1.join(); t2.join();
    std::cout << x << "\n";   // undefined behavior: data race
}
```

Example: Aliasing + mutation surprise (C++)

```cpp
#include <vector>
#include <iostream>

int main() {
    std::vector<int> v{1,2,3};
    int* p = &v[0];            // points into vector storage
    v.push_back(4);            // may reallocate -> p becomes dangling
    std::cout << *p << "\n";  // undefined behavior if reallocated
}
```

1.2 Manual Memory Management (C / C++)

Manual management gives maximum control and maximum responsibility. The programmer must guarantee:

- each allocation is freed exactly once,
- no pointer/reference outlives its target,
- aliasing does not violate invariants (especially with mutation),
- concurrent access is synchronized correctly.

C++ improves the situation with RAII and smart pointers, but **raw pointers, references, and aliasing remain expressible**. Therefore, many errors remain possible and are often discovered late (testing, fuzzing, sanitizers, production crashes).

C++ RAII reduces leaks, not all lifetime/aliasing bugs

```cpp
#include <memory>
```

```
#include <iostream>

int main() {
    auto p = std::make_unique<int>(7);  // RAII: will be freed once
    std::cout << *p << "\n";
}
```

RAII helps, but does not prevent accidental aliasing patterns:

```
#include <memory>
#include <iostream>

int main() {
    auto u = std::make_unique<int>(10);
    int* raw = u.get();           // raw alias
    u.reset();                    // frees memory
    std::cout << *raw << "\n";    // undefined behavior: use-after-free
}
```

Shared ownership is possible, but adds complexity

```
#include <memory>
#include <thread>
#include <iostream>

int main() {
    auto p = std::make_shared<int>(0);

    std::thread t1([&]{ for (int i=0;i<100000;i++) (*p)++; }); // data race
    std::thread t2([&]{ for (int i=0;i<100000;i++) (*p)++; }); // data race

    t1.join(); t2.join();
    std::cout << *p << "\n";
}
```

The memory is safe from premature free, but concurrency safety is still manual.

1.3 Garbage Collection Trade-offs (Java / Go / C#)

GC languages largely remove use-after-free and double free by construction: objects are reclaimed when unreachable. This is a huge practical win for many domains.
But GC typically implies trade-offs relevant to systems engineering:

- **Non-deterministic reclamation:** you do not control when memory is freed.
- **Latency variability:** collections can introduce pauses or scheduling noise.
- **Memory overhead:** extra metadata and higher peak memory usage are common.
- **Resource mismatch:** non-memory resources (files, sockets, locks) still require deterministic handling.

Java: GC handles memory, but resources need deterministic cleanup

```
import java.io.*;

public class Demo {
    public static void main(String[] args) throws Exception {
        try (FileInputStream f = new FileInputStream("input.bin")) {
            byte[] buf = f.readAllBytes();
            System.out.println(buf.length);
        } // deterministic close via try-with-resources
    }
}
```

Go: GC handles memory, but you still manage lifetimes of resources

```go
package main

import (
    "os"
    "fmt"
)

func main() {
    f, err := os.Open("input.bin")
    if err != nil { panic(err) }
    defer f.Close() // deterministic resource release

    info, _ := f.Stat()
    fmt.Println(info.Size())
}
```

C#: GC handles memory; deterministic disposal is explicit

```csharp
using System;
using System.IO;

class Demo {
    static void Main() {
        using var f = File.OpenRead("input.bin");
        Console.WriteLine(f.Length);
    } // deterministic disposal
}
```

GC reduces certain bug classes, but it does not give compile-time proofs about aliasing or data races, and it can be undesirable where tight control over latency, memory footprint, or deterministic destruction is required.

1.4 The Missing Model Before Rust

Before Rust, mainstream choices were often framed as:

- **Manual control:** fast and deterministic, but memory safety relies on discipline and tooling.
- **GC:** safer by default for memory, but with runtime cost and non-deterministic reclamation.

What was missing as a **default, language-enforced** model is:

Compile-time ownership + borrowing rules that prevent invalid lifetimes and unsafe aliasing,
while keeping deterministic destruction and avoiding a GC.

Rust fills this gap by making ownership explicit and making references safe by construction.

Rust: moves prevent double free

```rust
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;             // move: s1 becomes invalid

    // println!("{s1}");      // compile error: use of moved value
    println!("{s2}");
}
```

Rust: borrowing enables safe sharing (many readers)

```rust
fn main() {
    let s = String::from("hello");
    let a = &s;
    let b = &s;
    println!("{a} {b}");
}
```

Rust: borrowing enforces safe mutation (one writer)

```
fn main() {
    let mut s = String::from("hello");
    let w = &mut s;
    w.push_str(" world");
    println!("{w}");
}
```

Rust: prevents reader+writer aliasing in the same time window

```
fn main() {
    let mut s = String::from("hello");
    let r = &s;          // shared borrow (reader)
    // let w = &mut s;    // compile error: cannot borrow mutably while borrowed immutably
    println!("{r}");
}
```

Rust: the compiler blocks returning references to dead data

```
fn bad_ref() -> &String {
    let s = String::from("temp");
    &s // compile error: returns reference to data that will be dropped
}

fn main() {}
```

Windows workflow (PowerShell)

```
cargo new ch01_memory_problem
cd ch01_memory_problem
cargo run
cargo build --release
```

The key point is not syntax. The key point is the **model**: Rust makes lifetime and aliasing constraints enforceable at compile time, reducing whole families of memory and concurrency bugs without requiring a garbage collector.

Ownership: The Core Model

2.1 The Three Ownership Rules

Rust ownership is a compile-time model that answers three questions for every value: **Who owns it? How long does it live? Who may access it?** The ownership rules are simple and strict:

1. **Each value has exactly one owner at a time.**
2. **When the owner goes out of scope, the value is dropped.**
3. **Ownership can be transferred (moved) to another owner.**

These rules are enforced by the compiler. In safe Rust, this eliminates major bug classes: double free, many use-after-free patterns, and many lifetime mismatches.

Example: Single owner, scope-bound lifetime

```rust
fn main() {
    let s = String::from("owned"); // s owns the heap allocation
    println!("{s}");
} // s goes out of scope -> dropped (freed) deterministically
```

2.2 Stack vs Heap Under Ownership

Ownership applies to all values, but the safety payoff is most visible with heap allocations. A useful mental model:

- **Stack values** (fixed-size) are stored inline and copied by default when assigned if the type is Copy.
- **Heap-owning values** hold a small stack "handle" (pointer/len/cap, etc.) that manages heap storage.
- **Dropping** an owner runs destructors and releases owned resources (heap memory, OS handles, locks, etc.).

Example: Copy type (stack-only semantics)

```rust
fn main() {
    let a: u32 = 10;
    let b = a;              // copy (u32 implements Copy)
    println!("{a} {b}");    // both usable
}
```

Example: Heap owner (String)

```rust
fn main() {
    let s = String::from("hello"); // stack handle owns heap buffer
    println!("{s}");
} // drop(s) frees the heap buffer deterministically
```

Why this matters

If two names could own the same heap allocation, both would try to free it. Rust prevents that by making ownership transfer explicit (moves) and copies explicit (`clone()`).

2.3 Move Semantics

A **move** transfers ownership from one binding to another. After a move, the original binding becomes unusable (compile-time error if used). This is how Rust prevents double free and stale-owner bugs.

Example: Move of a heap-owning value

```rust
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;               // move ownership to s2

    // println!("{s1}");        // error: use of moved value
    println!("{s2}");          // OK
}
```

Example: Explicit deep copy via clone

```rust
fn main() {
    let s1 = String::from("hello");
    let s2 = s1.clone();       // allocates and copies heap data
    println!("{s1} {s2}");
}
```

Moves across function boundaries

Passing a value by value moves it (unless the type is `Copy`).

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

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

To keep the caller owning the value, pass a borrow:

```rust
fn view(s: &String) {
    println!("view: {s}");
}

fn main() {
    let s = String::from("hello");
    view(&s);                  // borrow, no move
    println!("{s}");           // still valid
}
```

2.4 Deterministic Destruction and `Drop`

Rust destruction is deterministic: when the owner ends, `drop` runs automatically. This is similar in spirit to C++ destructors, but ownership rules ensure values are dropped exactly once.

Example: Observing Drop order

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

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

fn main() {
    let _a = Tracer("A");
    let _b = Tracer("B");
    println!("in scope");
} // drops happen here (reverse declaration order): B then A
```

Example: Drop releases non-memory resources deterministically

```rust
use std::fs::File;
use std::io::{Read, Result};

fn main() -> Result<()> {
    let mut f = File::open("input.bin")?;  // OS handle owned by f
    let mut buf = Vec::new();
    f.read_to_end(&mut buf)?;
    println!("bytes: {}", buf.len());
    Ok(())
} // f dropped here -> file handle closed deterministically
```

Example: Early drop (explicit)

Sometimes you want to release a resource before end-of-scope:

```rust
use std::fs::File;

fn main() {
    let f = File::open("input.bin").unwrap();
    drop(f);                    // close now
    println!("file closed early");
}
```

2.5 Ownership Compared to C++ RAII

Rust ownership and C++ RAII share a core idea: **resources are released in destructors when objects leave scope**. However, Rust adds a crucial guarantee as a default language property:

- **C++:** RAII is a pattern. You can still create multiple raw aliases, misuse lifetimes, and violate invariants.
- **Rust:** ownership is a rule. In safe code, the compiler prevents many invalid lifetime and aliasing shapes.

C++: RAII works, but raw aliasing can reintroduce UAF

```cpp
#include <memory>
#include <iostream>

int main() {
    auto u = std::make_unique<int>(7);
    int* raw = u.get();      // raw alias
    u.reset();               // frees memory
    std::cout << *raw << "\n"; // undefined behavior: use-after-free
}
```

Rust: safe code blocks the same pattern structurally

In Rust, you cannot keep a safe reference alive past the owned value's lifetime. The following pattern is rejected at compile time:

```rust
fn main() {
    let r: &String;
    {
        let s = String::from("hello");
        r = &s;                 // borrow s
    }                            // s dropped here
    // println!("{r}");         // error: r would outlive s
}
```

C++ move vs Rust move (practical difference)

C++ move leaves the source object in a valid-but-unspecified state; it can still be used carefully. Rust move makes the old binding **unusable**, eliminating an entire category of "use-after-move" mistakes.

```rust
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;               // move
    // s1 is not usable at all here
    println!("{s2}");
}
```

Windows workflow (PowerShell)

```powershell
cargo new ch02_ownership
cd ch02_ownership
cargo run
cargo build --release
```

Ownership is the base layer. Borrowing (next chapter) builds on it to make safe references and safe concurrency practical, without garbage collection and without runtime cost in normal safe Rust code.

Borrowing: Safe References

3.1 Immutable Borrowing

An **immutable borrow** (`&T`) gives temporary read-only access to a value without transferring ownership. It is cheap (no allocation, no copy of the owned data) and is the primary way to pass large values efficiently.

Basic immutable borrow

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

fn main() {
    let s = String::from("hello");
    let n = len_of(&s);       // borrow, not move
    println!("len={}", n);
    println!("still owned: {}", s);
}
```

Prefer borrowing with slices when possible

Borrowing often uses `&str` instead of `&String` to accept more inputs.

```
fn starts_with_a(s: &str) -> bool {
    s.starts_with('A')
}

fn main() {
    let a = String::from("Ayman");
    let b = "Ali";
    println!("{}", starts_with_a(&a));
    println!("{}", starts_with_a(b));
}
```

Many immutable borrows are allowed

```
fn main() {
    let v = vec![10, 20, 30];
    let a = &v;
    let b = &v;
    println!("a[0]={} b[2]={}", a[0], b[2]);
}
```

3.2 Mutable Borrowing

A **mutable borrow** (`&mut T`) gives temporary exclusive access to mutate a value. While a mutable borrow exists, no other borrows (mutable or immutable) may exist.

Basic mutable borrow

```
fn add_suffix(s: &mut String) {
    s.push_str(" world");
}

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

Mutable borrow of a collection element

```
fn main() {
    let mut v = vec![1, 2, 3];
    let x = &mut v[1];
    *x += 10;
    println!("{:?}", v);
}
```

Why exclusivity matters

Exclusive access prevents accidental shared mutation and makes many optimizations valid. More importantly, it blocks a large class of concurrency and aliasing bugs by construction.

3.3 The "One Writer or Many Readers" Rule

Rust enforces this rule at compile time:

At any moment: either any number of `&T` OR exactly one `&mut T`.

This rule prevents **aliasing + mutation** in safe code. That combination is a common root cause of subtle bugs in systems programming.

This is allowed: many readers

```
fn main() {
```

```rust
    let s = String::from("hello");
    let r1 = &s;
    let r2 = &s;
    println!("{} {}", r1, r2);
}
```

This is allowed: one writer

```rust
fn main() {
    let mut s = String::from("hello");
    let w = &mut s;
    w.push('!');
    println!("{}", w);
}
```

This is rejected: readers + writer overlap

```rust
fn main() {
    let mut s = String::from("hello");
    let r = &s;          // reader
    // let w = &mut s;    // error: cannot borrow mutably while borrowed immutably
    println!("{}", r);
}
```

This is rejected: two writers overlap

```rust
fn main() {
    let mut x = 0;
    let a = &mut x;
    // let b = &mut x;    // error: cannot borrow x as mutable more than once at a time
    *a += 1;
    println!("{}", x);
}
```

3.4 Borrowing and Scope

Borrows last for as long as they are needed. In modern Rust, a borrow often ends at its **last use**, not necessarily at the end of the lexical block. This enables practical patterns while preserving safety.

Borrow ends after last use (common pattern)

```rust
fn main() {
    let mut s = String::from("hello");

    let r1 = &s;
    let r2 = &s;
    println!("{} {}", r1, r2); // last use of r1 and r2

    let w = &mut s;            // OK now
    w.push_str(" world");
    println!("{}", w);
}
```

Holding a borrow too long blocks mutation

```rust
fn main() {
    let mut s = String::from("hello");
    let r = &s;
    println!("{}", r);

    let w = &mut s;            // OK: r is not used after its last print
    w.push('!');
    println!("{}", s);
}
```

Borrowing and reallocation hazards (vector growth)

Rust prevents common “pointer into vector then grow vector” bugs by controlling borrows.

```rust
fn main() {
    let mut v = vec![1, 2, 3];
    let first = &v[0];
    // v.push(4);              // error: cannot borrow `v` as mutable because it is also
    ↪  borrowed as immutable
    println!("{}", first);
}
```

To do both safely, end the borrow before mutation:

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

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

    v.push(4);
    println!("{:?}", v);
}
```

3.5 Common Borrow Checker Errors

Borrow checker errors are usually about **overlapping access**. The fix is typically to shorten a borrow, change ownership (move/clone), or redesign to avoid overlapping aliases.

Error: use of moved value

Cause: ownership moved, old binding used again.

```rust
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;            // move
    // println!("{}", s1);  // error: moved value
    println!("{}", s2);
}
```

Fix: borrow or clone depending on intent.

```rust
fn main() {
    let s1 = String::from("hello");
    let s2 = s1.clone();     // explicit copy
    println!("{} {}", s1, s2);
}
```

Error: cannot borrow as mutable because it is also borrowed as immutable

Cause: reader and writer overlap.

```rust
fn main() {
    let mut s = String::from("hello");
    let r = &s;
    // let w = &mut s;         // error
    println!("{}", r);
}
```

Fix: end the immutable borrow before mutable borrow.

```rust
fn main() {
    let mut s = String::from("hello");
    let r = &s;
    println!("{}", r);       // last use of r
    let w = &mut s;
    w.push('!');
    println!("{}", s);
}
```

Error: cannot borrow as mutable more than once at a time

Cause: two mutable borrows overlap.

```
fn main() {
    let mut x = 0;
    let a = &mut x;
    // let b = &mut x;         // error
    *a += 1;
    println!("{}", x);
}
```

Fix: make the first borrow end before the second begins.

```
fn main() {
    let mut x = 0;
    {
        let a = &mut x;
        *a += 1;
    } // a ends here
    let b = &mut x;
    *b += 2;
    println!("{}", x);
}
```

Error: cannot return reference to local variable

Cause: reference would outlive the owned data.

```
fn bad_ref() -> &String {
    let s = String::from("temp");
    &s // error
}

fn main() {}
```

Fix: return ownership or borrow from an input.

```
fn make() -> String {
    String::from("owned")
}

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

fn main() {
    let s = make();
    println!("{}", s);

    let x = "abcd";
    let y = "xy";
    println!("{}", pick(x, y));
}
```

Windows workflow (PowerShell)

```
cargo new ch03_borrowing
cd ch03_borrowing
```

```
cargo run
cargo build --release
```

Borrowing is the mechanism that makes references safe by construction: you can express efficient, zero-copy APIs, while the compiler enforces that references are valid and that mutation is exclusive.

Lifetimes: Compile-Time Memory Contracts

4.1 What Lifetimes Really Are

A lifetime in Rust is not a runtime value. It is a **compile-time relationship** that describes how long references are guaranteed to be valid.
Key idea:

> **Lifetimes do not change how long data lives.** They describe and constrain **how references relate to that data**.

Ownership determines when data is dropped. Lifetimes ensure that references never outlive the data they point to.

Example: invalid reference to local data

```rust
fn bad() -> &String {
    let s = String::from("temp");
    &s                    // error: returns reference to local data
}

fn main() {}
```

The compiler rejects this because the returned reference would outlive ‘s‘, which is dropped at the end of ‘bad()‘.

Example: valid reference tied to caller’s data

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

fn main() {
    let name = String::from("Ayman");
    let ch = first_char(&name);
    println!("{}", ch);
}
```

Here, the returned reference is guaranteed to be valid because it refers to the caller’s data, not local temporary data.

4.2 Lifetime Annotations

When multiple references are involved, the compiler sometimes needs explicit lifetime annotations to understand relationships.
Lifetime parameters describe **how input and output references relate to each other**.

Example: choosing the longer string slice

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

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

Meaning:

- Both parameters must live at least as long as lifetime ‘’a‘.
- The returned reference lives no longer than ‘’a‘.
- The output cannot outlive the shorter of the two inputs.

The annotation does not extend lifetimes. It expresses constraints the compiler must enforce.

Example: struct holding a reference

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

fn main() {
    let s = String::from("hello");
    let h = Holder { data: &s };
    println!("{}", h.data);
}
```

The struct cannot outlive the data it borrows.

4.3 Lifetime Elision Rules

In common cases, Rust infers lifetimes automatically. This is called **lifetime elision**.

The compiler applies three main rules:

1. Each reference parameter gets its own lifetime.
2. If there is exactly one input lifetime, it is assigned to output lifetimes.
3. If there are multiple input lifetimes but one is `&self` or `&mut self`, the lifetime of `self` is assigned to outputs.

Example: single input reference

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

This is equivalent to:

```
fn len<'a>(s: &'a str) -> usize {
    s.len()
}
```

Example: method with &self

```
struct Person {
    name: String,
}

impl Person {
    fn name(&self) -> &str {
        &self.name
    }
}

fn main() {
    let p = Person { name: "Ayman".to_string() };
    println!("{}", p.name());
}
```

The returned reference is automatically tied to ‘&self‘.

Example requiring explicit annotation

```
fn choose(x: &str, y: &str) -> &str {
    // compiler error: cannot infer which input lifetime to return
    x
}
```

Fix with explicit lifetime:

```
fn choose<'a>(x: &'a str, y: &'a str) -> &'a str {
    x
}
```

4.4 Why Other Languages Cannot Enforce This

Most mainstream languages do not encode lifetime relationships in their type systems.

C / C++

References and pointers carry no lifetime metadata. The compiler does not enforce that returned references outlive their targets.

```
#include <string>

const std::string& bad() {
    std::string s = "temp";
    return s;              // undefined behavior: reference to destroyed object
}
```

This compiles, but is invalid at runtime.

Garbage-Collected Languages

GC languages prevent use-after-free for memory, but they do not express lifetime relationships between references at compile time. The runtime ensures memory remains allocated while reachable, but there is no static proof tying returned references to input lifetimes.

Rust's Difference

Rust combines:

- Deterministic destruction (ownership),
- Compile-time borrow checking,
- Explicit lifetime relationships in types.

The result:

- References cannot outlive the data they point to.
- Aliasing rules are enforced at compile time.
- No garbage collector is required.
- No runtime cost for lifetime checking in normal safe code.

Windows workflow (PowerShell)

```
cargo new ch04_lifetimes
cd ch04_lifetimes
cargo run
cargo build --release
```

Lifetimes are not syntax decoration. They are static contracts that make reference validity a provable property of the program, closing one of the largest historical gaps in systems programming.

Ownership and Concurrency

5.1 Data Races in Traditional Systems

A **data race** occurs when:

- two or more threads access the same memory location concurrently,
- at least one access is a write,
- there is no proper synchronization establishing ordering and exclusivity.

In many traditional systems languages, the type system does not prevent data races. The program may compile and appear correct, yet be invalid under the language memory model.

C++ example: unsynchronized shared increment

```cpp
#include <thread>
#include <iostream>

int main() {
    int x = 0;

    std::thread t1([&]{ for (int i=0;i<1'000'000;i++) ++x; });
    std::thread t2([&]{ for (int i=0;i<1'000'000;i++) ++x; });
    t1.join(); t2.join();
    std::cout << x << "\n";  // undefined behavior: data race
}
```

The problem is not “sometimes wrong output”; the behavior is not defined by the language. Optimizers may assume it cannot happen and transform code unpredictably.

Why this is hard

Even experienced engineers create races when:

- shared state spreads across multiple modules,
- invariants rely on undocumented “who writes when”,
- locks are taken inconsistently, or not at all on some paths,
- reference/pointer aliasing makes ownership unclear.

Rust approaches concurrency by making safe sharing explicit and provable.

5.2 Send and Sync

Rust uses two core marker traits to express thread-safety:

- **Send:** a type can be transferred (moved) to another thread safely.
- **Sync:** a type can be shared by reference across threads safely (`&T` is safe to use from multiple threads).

These are enforced at compile time by trait bounds used throughout the standard library. Most plain data types are `Send` and `Sync`. Types that enable unsynchronized mutation are typically **not** `Sync`.

Example: moving ownership into a thread (Send)

```rust
use std::thread;

fn main() {
    let v = vec![1, 2, 3, 4];

    let h = thread::spawn(move || {
        let sum: i32 = v.iter().sum();
        println!("sum={}", sum);
    });

    h.join().unwrap();
}
```

Here, `move` transfers ownership into the thread. After spawning, the original thread cannot use `v` anymore.

Example: why some types cannot be sent/shared

`Rc<T>` is for single-threaded reference counting. Trying to send it to another thread is rejected.

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

fn main() {
    let x = Rc::new(5);

    // let h = thread::spawn(move || println!("{}", x));
    // error: `Rc<i32>` cannot be sent between threads safely

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

The correct multi-threaded shared ownership primitive is `Arc<T>`.

5.3 Fearless Concurrency

Rust concurrency is called "fearless" because safe Rust code prevents common race shapes by construction:

- You cannot mutate shared data through multiple aliases without synchronization.
- Shared ownership across threads is explicit (`Arc`).
- Shared mutation requires explicit synchronization (`Mutex`, `RwLock`, atomics).

Example: shared mutation with `Arc<Mutex<T»`

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

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

    for _ in 0..4 {
        let c = Arc::clone(&counter);
        let h = thread::spawn(move || {
            for _ in 0..100_000 {
                let mut g = c.lock().unwrap();
                *g += 1;
            }
        });
        handles.push(h);
    }
    for h in handles {
        h.join().unwrap();
    }

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

Key points:

- `Arc` provides thread-safe shared ownership.
- `Mutex` provides exclusive access for mutation.
- The type system ensures you cannot forget the lock and still mutate the inner value.

Example: read-heavy workloads with `RwLock`

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

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

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

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

    r.join().unwrap();
    w.join().unwrap();
    println!("final={:?}", *data.read().unwrap());
}
```

Example: lock-free counter with atomics

```
use std::sync::atomic::{AtomicU64, Ordering};
use std::sync::Arc;
use std::thread;

fn main() {
    let x = Arc::new(AtomicU64::new(0));
    let mut hs = Vec::new();

    for _ in 0..4 {
        let a = Arc::clone(&x);
        hs.push(thread::spawn(move || {
            for _ in 0..100_000 {
                a.fetch_add(1, Ordering::Relaxed);
            }
        }));
    }

    for h in hs { h.join().unwrap(); }
    println!("x={}", x.load(Ordering::Relaxed));
}
```

Atomics still require correct ordering choices; Rust prevents data races but does not remove the need for correct synchronization semantics.

5.4 Shared State vs Message Passing

Rust supports two primary concurrency styles:

Shared state (synchronized memory)

Use `Arc` + `Mutex`/`RwLock` or atomics when threads must access shared structures. This style is natural for caches, shared queues, global registries, and coordination state.

Message passing (ownership transfer)

Instead of sharing memory, send owned data between threads. This often simplifies reasoning because ownership moves and prevents aliasing.

Example: channel-based work distribution

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

fn main() {
    let (tx, rx) = mpsc::channel::<String>();

    let producer = thread::spawn(move || {
        for i in 0..5 {
            tx.send(format!("job-{i}")).unwrap();
        }
        // tx dropped here -> receiver will eventually stop
    });
    let consumer = thread::spawn(move || {
        while let Ok(job) = rx.recv() {
            println!("got {}", job);
        }
        println!("receiver done");
    });

    producer.join().unwrap();
    consumer.join().unwrap();
}
```

Here, each message is **owned** by the receiver. No shared mutable state is required.

Choosing between the two

- Prefer **message passing** when ownership can flow naturally with the data.
- Prefer **shared state** when multiple workers must query/update a shared structure efficiently.
- Combine them: e.g., send tasks by channel, store results in synchronized shared state.

Windows workflow (PowerShell)

```
cargo new ch05_concurrency
cd ch05_concurrency
cargo run
cargo build --release
```

Rust does not magically solve concurrency, but it makes unsafe sharing patterns difficult (or impossible) in safe code: ownership controls who may access data, borrowing controls aliasing, and Send/Sync ensure thread boundaries respect those rules.

Zero-Cost Safety

6.1 No Garbage Collector

Rust has no garbage collector. Memory is reclaimed deterministically when owners go out of scope. This is the same practical benefit as C++ RAII, but enforced as a language rule across the ecosystem.

Deterministic release of heap memory

```rust
fn main() {
    let v = vec![1u32, 2, 3, 4, 5]; // heap allocation
    println!("len={}", v.len());
} // v dropped here -> heap freed deterministically
```

Deterministic release of OS resources

```rust
use std::fs::File;
use std::io::{Read, Result};

fn main() -> Result<()> {
    let mut f = File::open("input.bin")?; // OS handle owned by f
    let mut buf = Vec::new();
    f.read_to_end(&mut buf)?;
    println!("bytes={}", buf.len());
    Ok(())
} // f dropped here -> file handle closed deterministically
```

Early release when needed

```
use std::fs::File;

fn main() {
    let f = File::open("input.bin").unwrap();
    drop(f); // close immediately, before end of scope
    println!("closed early");
}
```

6.2 No Runtime Borrow Checking

Rust's borrow rules (`&T` vs `&mut T`) are enforced at compile time in normal safe code. There is no runtime "borrow table" and no runtime checks for ordinary borrows.

Compile-time enforcement (no runtime cost)

```
fn main() {
    let mut s = String::from("hello");

    let r1 = &s;
    let r2 = &s;
    println!("{} {}", r1, r2);

    let w = &mut s;          // OK: r1/r2 are no longer used
    w.push('!');
    println!("{}", w);
}
```

If you truly need dynamic borrow behavior (shared mutation in a single thread), Rust provides **explicit** runtime-checked mechanisms such as `RefCell`. This is not the default; you opt in and pay only when you choose it.

Explicit runtime checking with `RefCell` (opt-in)

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

fn main() {
    let x = RefCell::new(0);

    {
        let mut a = x.borrow_mut();
        *a += 1;
    } // mutable borrow ends

    let b = x.borrow();
    println!("{}", *b);
}
```

6.3 Compile-Time Guarantees

Rust safety comes primarily from static guarantees:

- **No dangling references in safe code:** a reference cannot outlive its referent.
- **No double free in safe code:** ownership is unique unless explicitly shared.
- **No data races in safe code:** shared mutation across threads requires synchronization.
- **No implicit aliasing with mutation:** the borrow rules prevent "reader+writer overlap".

Guarantee: cannot return reference to local data

```rust
fn bad_ref() -> &String {
    let s = String::from("temp");
    &s // compile error: reference would outlive s
}

fn main() {}
```

Guarantee: move prevents accidental double free

```rust
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;            // move
    // println!("{}", s1);   // compile error: moved value
    println!("{}", s2);
}
```

Guarantee: shared mutation across threads must be synchronized

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

fn main() {
    let x = Arc::new(Mutex::new(0i32));
    let mut hs = Vec::new();

    for _ in 0..2 {
        let x2 = Arc::clone(&x);
        hs.push(thread::spawn(move || {
            for _ in 0..100_000 {
                let mut g = x2.lock().unwrap();
                *g += 1;
            }
        }));
    }

    for h in hs { h.join().unwrap(); }
    println!("{}", *x.lock().unwrap());
}
```

The compiler is not "guessing". It uses strict type and trait rules (ownership/borrowing + Send/Sync) to prevent unsafe shapes at compile time.

6.4 Performance Compared to C and C++

Rust is designed as a zero-cost abstraction language: high-level constructs compile down to efficient machine code, typically comparable to C/C++ for equivalent algorithms.
Performance basics in Rust for this booklet's scope:

- **No GC pauses:** no collector running in the background.
- **Deterministic destruction:** predictable cleanup points.
- **Borrowing is free:** references are plain pointers under the hood.
- **Bounds checks exist but optimize well:** the compiler can eliminate checks when provably safe.

Example: slice-based APIs avoid allocation and copies

```
fn sum(xs: &[u64]) -> u64 {
    let mut s = 0;
    for &x in xs { s += x; }
    s
}

fn main() {
    let v = vec![1u64, 2, 3, 4, 5];
    println!("{}", sum(&v)); // borrow slice, no copy of v
}
```

Example: iterator style can be optimized similarly to loops

```
fn sum_iter(xs: &[u64]) -> u64 {
    xs.iter().copied().sum()
}

fn main() {
    let v = vec![1u64, 2, 3, 4, 5];
    println!("{}", sum_iter(&v));
}
```

Example: avoiding unnecessary clones

A common beginner performance mistake is cloning owned values instead of borrowing.

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

fn main() {
    let s = String::from("hello");
    print_len(&s);      // borrow: no allocation, no copy
    // print_len(s.clone().as_str()); // unnecessary work
}
```

Release builds matter (like C/C++)

Rust debug builds prioritize fast compilation and checks. For performance comparisons, use release mode.

```
cargo run
cargo run --release
cargo build --release
```

Practical comparison summary

- Rust can match C/C++ for many workloads because it compiles to native code and has no GC.
- Safety is achieved primarily at compile time, not via runtime tracking.
- You still choose algorithms, data structures, and synchronization correctly; Rust prevents many unsafe shapes but does not remove engineering responsibility.

In short: Rust's ownership and borrowing rules add constraints at compile time, not a garbage collector at runtime. That is the core meaning of "zero-cost memory safety" in Rust.

Advanced Ownership Patterns

7.1 Interior Mutability (`Cell`, `RefCell`)

Rust's normal rule is: if you have `&T`, you cannot mutate `T`. **Interior mutability** is the controlled exception: you can mutate through an immutable reference, but only using specific types that enforce safety rules.

`Cell<T>`: copy-in / copy-out mutation

`Cell<T>` is for small `Copy` types (no borrows to the interior). It provides mutation by value (set/get/replace) without exposing references to the inner data.

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

fn main() {
    let x = Cell::new(10u32);

    x.set(11);
    let v = x.get();
    println!("v={}", v);

    let old = x.replace(99);
    println!("old={} new={}", old, x.get());
}
```

Typical use: caching counters, flags, or statistics inside an object that is otherwise shared immutably.

`RefCell<T>`: runtime-checked borrowing (single-threaded)

`RefCell<T>` enforces Rust's borrowing rules at **runtime**:

- many immutable borrows are allowed, or
- exactly one mutable borrow is allowed,

and violating that causes a panic.

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

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

    {
        let mut w = data.borrow_mut(); // one writer
        w.push(4);
    } // writer ends here

    let r = data.borrow();             // readers ok now
    println!("{:?}", *r);
}
```

Demonstrating the runtime rule (will panic if uncommented)

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

fn main() {
    let x = RefCell::new(0);
    let _a = x.borrow_mut();
    // let _b = x.borrow_mut(); // panic: already borrowed mutably

    println!("done");
}
```

Guideline:

- Prefer normal borrowing (`&T` / `&mut T`) when possible (compile-time).
- Use `RefCell` only when the design requires mutation behind shared references **within one thread**.

Interior mutability for a cache

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

struct Cache {
    hits: RefCell<u64>,
}

impl Cache {
    fn new() -> Self { Self { hits: RefCell::new(0) } }

    fn record_hit(&self) {
        *self.hits.borrow_mut() += 1;
    }

    fn hits(&self) -> u64 {
        *self.hits.borrow()
    }
}
```

```
fn main() {
    let c = Cache::new();
    c.record_hit();
    c.record_hit();
    println!("hits={}", c.hits());
}
```

7.2 Smart Pointers (Box, Rc, Arc)

Rust makes ownership explicit; smart pointers are the standard tools for expressing common ownership patterns.

Box<T>: single-owner heap allocation

Box<T> allocates T on the heap with a single owner. It is often used for:

- large values you do not want to move on the stack,
- recursive types,
- trait objects (Box<dyn Trait>).

```
fn main() {
    let b = Box::new([0u8; 1024]); // heap allocation
    println!("size={}", b.len());
} // dropped -> heap freed
```

Recursive type example:

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

```rust
}

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

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

`Rc<T>` provides reference-counted shared ownership for a single thread. Cloning an `Rc` increments the count (cheap); dropping decrements.

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

fn main() {
    let a = Rc::new(String::from("shared"));
    let b = Rc::clone(&a);
    let c = Rc::clone(&a);

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

Important: `Rc<T>` is not thread-safe and cannot be sent across threads.

Arc<T>: shared ownership (multi-threaded)

`Arc<T>` is an atomic reference-counted pointer, safe to share across threads. Shared mutation still requires synchronization (e.g., `Mutex`, `RwLock`).

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

fn main() {
    let s = Arc::new(String::from("hello"));
    let mut hs = Vec::new();

    for _ in 0..2 {
        let s2 = Arc::clone(&s);
        hs.push(thread::spawn(move || {
            println!("{}", s2);
        }));
    }

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

Shared mutation pattern:

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

fn main() {
    let x = Arc::new(Mutex::new(0u64));
    let mut hs = Vec::new();

    for _ in 0..4 {
        let x2 = Arc::clone(&x);
        hs.push(thread::spawn(move || {
            for _ in 0..50_000 {
                *x2.lock().unwrap() += 1;
            }
        }));
    }

    for h in hs { h.join().unwrap(); }
    println!("{}", *x.lock().unwrap());
}
```

Choosing the right pointer

- `Box<T>`: single owner, heap allocation.
- `Rc<T>`: shared ownership, one thread.
- `Arc<T>`: shared ownership, many threads (atomic refcount).

7.3 When and Why `unsafe` Exists

Rust's safe subset enforces strict rules that eliminate many bug classes. However, systems programming sometimes needs operations the compiler cannot verify. For that, Rust provides `unsafe` as an explicit boundary.

What `unsafe` means

`unsafe` does not turn off the borrow checker globally. It allows a specific set of actions that require the programmer to uphold additional invariants, such as:

- dereferencing raw pointers,
- calling `unsafe` functions (FFI, intrinsics),
- accessing or mutating `static mut`,
- implementing `unsafe` traits,
- certain low-level memory operations.

Example: raw pointer dereference

```
fn main() {
    let mut x = 10i32;
    let p: *mut i32 = &mut x;

    unsafe {
        *p += 1;               // programmer must ensure p is valid
    }

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

Example: FFI call shape (Windows-friendly concept)

The key idea: calling foreign code is `unsafe` because Rust cannot verify external invariants.

```
#[allow(non_camel_case_types)]
type c_int = i32;

extern "C" {
    fn abs(x: c_int) -> c_int;
}

fn main() {
    let x = -7;
    let y = unsafe { abs(x) }; // unsafe boundary
    println!("{}", y);
}
```

Safe abstraction built on unsafe

Most real-world `unsafe` should be contained inside a small module that exports a safe API. Example: a function that returns a slice from raw parts is unsafe internally, but can be wrapped safely.

```
fn safe_first_byte(buf: &[u8]) -> Option<u8> {
    buf.first().copied()
}

fn unsafe_first_byte(ptr: *const u8, len: usize) -> Option<u8> {
    if ptr.is_null() || len == 0 { return None; }
    unsafe {
        let s = std::slice::from_raw_parts(ptr, len); // unsafe: ptr must be valid for len
        ↪  bytes
        s.first().copied()
    }
}

fn main() {
    let v = vec![1u8, 2, 3];
    println!("{:?}", safe_first_byte(&v));
    println!("{:?}", unsafe_first_byte(v.as_ptr(), v.len()));
}
```

Why `unsafe` is not a failure

`unsafe` exists because:

- low-level hardware/OS interfaces require raw pointers and external invariants,
- some optimizations require manual proof beyond the compiler's current reasoning,
- interoperability (C ABI) is essential for real systems.

The Rust discipline is:

- keep `unsafe` small,
- document invariants precisely,
- expose safe APIs to the rest of the codebase.

Windows workflow (PowerShell)

```
cargo new ch07_advanced_patterns
cd ch07_advanced_patterns
cargo run
cargo build --release
```

These patterns are “advanced” because they make trade-offs explicit: interior mutability opts into runtime checks, smart pointers opt into specific ownership models, and `unsafe` opts into programmer-proven invariants when the compiler cannot prove them.

Conclusion — A Paradigm Shift in Memory Safety

From Discipline to Enforcement

For decades, systems programming relied on discipline, conventions, and tooling to control memory lifetime and aliasing. C and C++ provide power and performance, but correctness depends heavily on human reasoning.
Garbage-collected languages removed entire classes of memory errors, but at the cost of runtime systems, non-deterministic destruction, and limited compile-time guarantees about aliasing and concurrency.
Rust introduces a structural shift:

Memory safety as a compile-time property, without a garbage collector.

Ownership answers:

- Who owns this value?
- When is it destroyed?
- Can there be multiple owners?

Borrowing answers:

- Who can read?
- Who can write?
- For how long?

Lifetimes answer:

- How are references related to the data they point to?
- Can a reference ever outlive its referent?

These are not guidelines. They are enforced invariants.

Safety Without a Garbage Collector

Rust achieves memory safety primarily through:

- Unique ownership by default,
- Explicit moves instead of implicit aliasing,
- Strict borrow rules (one writer or many readers),
- Deterministic destruction via `Drop`,
- Trait-based concurrency guarantees (`Send`, `Sync`).

There is:

- No tracing garbage collector,
- No background compaction,

- No hidden reference counting by default,
- No runtime borrow table in safe code.

References are plain pointers at runtime. The safety reasoning happens at compile time.

Eliminating Entire Bug Families

In safe Rust:

- A value cannot be freed twice.
- A reference cannot outlive the data it points to.
- Two mutable references cannot exist simultaneously.
- Shared mutation across threads requires explicit synchronization.

Consider again a pattern common in other systems:

```cpp
#include <memory>
#include <iostream>

int main() {
    auto u = std::make_unique<int>(5);
    int* raw = u.get();
    u.reset();
    std::cout << *raw << "\n"; // undefined behavior
}
```

In Rust, the corresponding invalid shape is rejected at compile time:

```
fn main() {
    let r: &String;
    {
        let s = String::from("hello");
        r = &s;
    }
    // println!("{}", r); // compile error: r would outlive s
}
```

The shift is not syntactic. It is structural: unsafe lifetime shapes are not representable in safe Rust.

Performance Is Not the Trade-Off

Rust compiles to native code using LLVM. Ownership and borrowing are static analyses. They do not add runtime overhead in ordinary safe code.

```
fn sum(xs: &[u64]) -> u64 {
    xs.iter().copied().sum()
}

fn main() {
    let v = vec![1u64, 2, 3, 4, 5];
    println!("{}", sum(&v)); // borrow, no copy
}
```

Release build on Windows:

```
cargo build --release
cargo run --release
```

Safety is not implemented through runtime tracking. It is implemented through type and lifetime constraints.

When Unsafe Is Required

Systems programming sometimes requires operations that cannot be statically verified: raw pointers, FFI, hardware access, custom allocators.
Rust does not hide these realities. It isolates them.

```rust
fn main() {
    let mut x = 10;
    let p = &mut x as *mut i32;

    unsafe {
        *p += 1; // programmer must uphold invariants
    }

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

The discipline becomes:

- Unsafe blocks are explicit.
- Invariants are local and auditable.
- Safe abstractions wrap unsafe internals.

This sharply reduces the surface area where undefined behavior can originate.

A New Default for Systems Programming

Rust does not eliminate the need for engineering judgment. Algorithms, synchronization strategies, and architecture still matter.
But Rust changes the baseline:

- Memory lifetime is explicit.
- Aliasing rules are enforced.
- Concurrency constraints are encoded in types.
- Safety is opt-out (`unsafe`), not opt-in.

That is the paradigm shift.
Instead of asking:

“Did we remember to make this safe?”

Rust asks:

“Can this program even compile if it is unsafe?”

Ownership and borrowing are not features layered on top of a language. They are the language’s memory model.
That is why Rust represents not an incremental improvement, but a structural redesign of how memory safety is achieved without a garbage collector and without sacrificing performance.

Appendices

Appendix A — Mapping C++ Concepts to Rust

This appendix maps common C++ concepts to Rust equivalents, focusing on **ownership, borrowing, lifetimes, and concurrency**. The goal is conceptual alignment: **what is guaranteed**, **what is explicit**, and **what is prevented by default**.

A.1 Core Concepts at a Glance

- **RAII:** Both languages use deterministic destruction. Rust additionally enforces single ownership by default.
- **References:** C++ references/pointers do not encode lifetime or exclusivity; Rust references encode both.
- **Aliasing:** C++ permits aliasing + mutation; Rust forbids unsafe aliasing shapes in safe code.
- **Concurrency:** C++ relies on discipline; Rust encodes thread-safety through types and traits (`Send`, `Sync`).

A.2 Ownership vs C++ Object Lifetime

C++: lifetime is managed by scope (stack) and RAII wrappers (heap). **Rust:** lifetime is tied to ownership; moves are enforced and previous bindings become unusable.

C++: unique ownership with `std::unique_ptr`

```
#include <memory>
#include <iostream>

int main() {
    auto p = std::make_unique<int>(7);
    std::cout << *p << "\n";
} // freed here
```

Rust: unique ownership with `Box<T>`

```
fn main() {
    let p = Box::new(7);
    println!("{}", p);
} // dropped here
```

Difference:

- In C++, raw pointer aliases can outlive ownership if misused.
- In Rust safe code, references cannot outlive their owner.

A.3 Move Semantics: Stronger Default Safety

C++ move: source remains valid-but-unspecified. **Rust move:** the old binding becomes unusable.

```rust
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;             // move
    // println!("{}", s1);  // compile error
    println!("{}", s2);
}
```

Explicit deep copy:

```rust
fn main() {
    let s1 = String::from("hello");
    let s2 = s1.clone();
    println!("{} {}", s1, s2);
}
```

A.4 References and Borrowing vs C++ References/Pointers

C++: `T&` and `T*` do not encode exclusivity. **Rust:**

- `&T` = shared read access,
- `&mut T` = exclusive write access.

Many readers

```rust
fn main() {
    let v = vec![1, 2, 3];
    let a = &v;
    let b = &v;
    println!("{} {}", a[0], b[2]);
}
```

One writer

```
fn main() {
    let mut v = vec![1, 2, 3];
    let w = &mut v;
    w.push(4);
    println!("{:?}", w);
}
```

Reader + writer overlap (blocked)

```
fn main() {
    let mut v = vec![1, 2, 3];
    let r = &v[0];
    // v.push(4);            // compile error
    println!("{}", r);
}
```

This prevents classic C++ vector-reallocation invalidation bugs.

A.5 Lifetimes vs Dangling References

C++: returning `T&` can compile even if it refers to dead storage. **Rust:** lifetime relationships must be provable.

C++: compiles but invalid

```
#include <string>

const std::string& bad() {
    std::string s = "temp";
    return s; // dangling
}
```

Rust: rejected

```rust
fn bad_ref() -> &String {
    let s = String::from("temp");
    &s // compile error
}
fn main() {}
```

Explicit lifetime relationship:

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

A.6 Shared Ownership: `shared_ptr` vs `Rc/Arc`

C++: `shared_ptr` handles reference counting; thread-safety of the pointee is separate. **Rust:**

- `Rc<T>` for single-threaded shared ownership,
- `Arc<T>` for multi-threaded shared ownership.

`Rc<T>`

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

fn main() {
    let x = Rc::new(String::from("shared"));
    let a = Rc::clone(&x);
    println!("count={}", Rc::strong_count(&x));
    println!("{} {}", x, a);
}
```

Arc<T>

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

fn main() {
    let x = Arc::new(String::from("shared"));
    let x2 = Arc::clone(&x);

    let h = thread::spawn(move || {
        println!("{}", x2);
    });

    h.join().unwrap();
    println!("{}", x);
}
```

A.7 Mutex and Locking

Both ecosystems use mutexes; Rust encodes safe access via guard types.

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

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

    let x2 = Arc::clone(&x);
    let h = thread::spawn(move || {
        *x2.lock().unwrap() += 1;
    });

    h.join().unwrap();
    println!("{}", *x.lock().unwrap());
}
```

You cannot mutate the inner value without acquiring the lock, because access is mediated by the guard.

A.8 `const` Correctness vs Borrowing

Mapping:

- C++ `const T&` ≈ Rust `&T`
- C++ `T&` ≈ Rust `&mut T`

Difference: `&mut T` implies exclusivity, not just mutability.

A.9 Interior Mutability vs `mutable`

C++ allows mutation through `mutable` and `const_cast`. Rust uses controlled types:

- `Cell<T>` for small `Copy` values,
- `RefCell<T>` for runtime-checked borrows,
- `Mutex<T>` / `RwLock<T>` for thread-safe mutation.

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

fn main() {
    let x = RefCell::new(vec![1, 2, 3]);

    {
        let mut w = x.borrow_mut();
        w.push(4);
    }

    println!("{:?}", *x.borrow());
}
```

A.10 Raw Pointers and `unsafe`

Both languages allow raw pointers. Rust requires explicit `unsafe` blocks.

```
fn main() {
    let mut x = 10i32;
    let p: *mut i32 = &mut x;

    unsafe {
        *p += 1;
    }

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

Best practice: keep `unsafe` small, document invariants, and expose safe abstractions.

A.11 Windows workflow (PowerShell)

```
cargo new appendix_a_cpp_to_rust
cd appendix_a_cpp_to_rust
cargo run
cargo build --release
```

Rust does not merely mirror RAII. It enforces ownership, borrowing, and lifetime relationships as type-level invariants, making many invalid states unrepresentable in safe code while preserving native performance.

Appendix B — Ownership Error Reference Guide

This appendix is a compact cheat sheet for the most common **ownership/borrowing/lifetime** errors. For each error family you get: **what it means**, **why Rust rejects it**, and **standard fixes**.

B.1 Use of Moved Value

Meaning: ownership was moved (non-Copy type) and the old binding was used again.

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;              // move
    println!("{}", s1);       // error: use of moved value
}
```

Fix: borrow (no move), clone (two owners), or move in/out via return.

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

fn main() {
    let s1 = String::from("hello");
    print_len(&s1);           // borrow
    let s2 = s1.clone();      // explicit deep copy
    println!("{} {}", s1, s2);
}
```

B.2 Cannot Borrow as Mutable Because It Is Also Borrowed as Immutable

Meaning: a reader (`&T`) and a writer (`&mut T`) overlap.

```
fn main() {
    let mut s = String::from("hello");
    let r = &s;
    let w = &mut s;           // error: mutable borrow while immutable borrow exists
    w.push('!');
    println!("{}", r);
}
```

Fix: end the read borrow before writing (reorder or add a scope).

```
fn main() {
    let mut s = String::from("hello");
    let r = &s;
    println!("{}", r);        // last use of r
    let w = &mut s;
    w.push('!');
    println!("{}", s);
}
```

B.3 Cannot Borrow as Mutable More Than Once at a Time

Meaning: two `&mut` borrows overlap.

```
fn main() {
    let mut x = 0;
    let a = &mut x;
    let b = &mut x;           // error: second mutable borrow
    *a += 1;
    *b += 2;
    println!("{}", x);
}
```

Fix: sequence the borrows, or borrow disjoint parts safely.

```
fn main() {
    let mut x = 0;
    { let a = &mut x; *a += 1; } // a ends
```

```rust
    { let b = &mut x; *b += 2; } // b ends
    println!("{}", x);

    let mut arr = [10, 20, 30, 40];
    let (l, r) = arr.split_at_mut(2); // proven disjoint
    l[0] += 1;
    r[0] += 2;
    println!("{:?}", arr);
}
```

B.4 Cannot Return Reference to Local Variable

Meaning: the returned reference would outlive the local value.

```rust
fn bad() -> &String {
    let s = String::from("temp");
    &s // error
}
fn main() {}
```

Fix: return ownership, or return a borrow tied to an input lifetime.

```rust
fn make() -> String { String::from("owned") }

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

fn main() {
    let s = make();
    println!("{}", s);

    let a = String::from("abcd");
    let b = String::from("xy");
    println!("{}", longest(&a, &b));
}
```

B.5 Borrowed Value Does Not Live Long Enough

Meaning: a reference was stored beyond the owner's scope.

```
fn main() {
    let r: &str;
    {
        let s = String::from("hello");
        r = &s;              // error: r would outlive s
    }
    println!("{}", r);
}
```

Fix: extend the owner's lifetime, or store an owned value instead of a reference.

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

    let owned: String;
    {
        let t = String::from("move-me");
        owned = t;           // move ownership out
    }
    println!("{}", owned);
}
```

B.6 Container Borrow Blocks Mutation (Reallocation Hazard Pattern)

Meaning: you hold a reference into a container, then try to mutate the container.

```rust
fn main() {
    let mut v = vec![1, 2, 3];
    let first = &v[0];
    // v.push(4);              // error: cannot mutably borrow while `first` exists
    println!("{}", first);
}
```

Fix: end the borrow before mutation, or copy out `Copy` values.

```rust
fn main() {
    let mut v = vec![1i32, 2, 3];

    { let first = &v[0]; println!("{}", first); } // borrow ends
    v.push(4);

    let first_copy = v[0];   // Copy, no borrow kept
    v.push(5);

    println!("first_copy={} v={:?}", first_copy, v);
}
```

B.7 Thread Boundary Errors: Not Send / Not Sync

Meaning: a type is not safe to move/share across threads (e.g., `Rc<T>`).

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

fn main() {
    let x = Rc::new(5);
    // thread::spawn(move || println!("{}", x)); // error: Rc<T> is not Send
    println!("{}", x);
}
```

Fix: use `Arc<T>` for cross-thread shared ownership (and add `Mutex`/`RwLock` for mutation).

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

fn main() {
    let x = Arc::new(Mutex::new(0i32));

    let x2 = Arc::clone(&x);
    let h = thread::spawn(move || {
        *x2.lock().unwrap() += 1;
    });

    h.join().unwrap();
    println!("{}", *x.lock().unwrap());
}
```

B.8 Quick Fix Checklist

When the borrow checker rejects code, try:

- Borrow instead of move: `&T`, `&mut T`, `&str`, `&[T]`.
- Shorten a borrow: reorder code or add a scope block `{ }`.
- Copy out `Copy` values instead of borrowing.
- Borrow disjoint parts: `split_at_mut`, `chunks_mut`.
- Need shared ownership: `Rc` (single-thread) or `Arc` (multi-thread).
- Need shared mutation: `RefCell` (single-thread) or `Mutex`/`RwLock` (threads).
- Need low-level: isolate `unsafe` and wrap it in a safe API.

B.9 Windows workflow (PowerShell)

```
cargo new appendix_b_errors
cd appendix_b_errors
cargo run
cargo build --release
```

References

Official Rust Language Books and Specifications

- **The Rust Programming Language ("The Book")**
  Primary official introduction and the canonical baseline for ownership, borrowing, lifetimes, and fearless concurrency. This text explicitly tracks modern Rust and the Rust 2024 Edition toolchain assumptions.

- **The Rust Reference**
  The authoritative specification-style document for Rust syntax and semantics. Use it to confirm precise rules: moves, drops, borrows, lifetimes, coercions, trait rules, and unsafe semantics.

- **The Rust Standard Library Documentation (`std`)**
  The definitive reference for ownership-centric APIs and core types: `String`, `Vec`, `Option`, `Result`, and the concurrency and synchronization primitives.

Ownership, Borrowing, and Lifetimes Deep Dives

- **The Rustonomicon (Unsafe Rust)**
  Companion-level material for understanding the edges of ownership and lifetimes when

you must cross into `unsafe`. Covers invariants behind references, aliasing, destructors, and the rules that safe Rust relies on.

- **Rust 2024 Edition Guide**
  Explains edition-driven language and idiom changes, migration concerns, and how modern Rust is intended to be written (especially relevant if your codebase spans editions).

Concurrency and Memory Safety Guarantees

- **`Send` and `Sync` in the Standard Library Docs**
  The formal definitions of thread-safety at the type level, including: `T: Sync` iff `&T: Send`, and the meaning of "safe to share references" vs "safe to move ownership."

- **The Rustonomicon: `Send` and `Sync`**
  Explains why these traits are `unsafe` to implement, what assumptions unsafe code can make about them, and how they connect to Rust's data-race prevention in safe code.

Editions and Stability Notes

- **Rust Blog: Rust 1.85.0 and Rust 2024**
  Release announcement and rationale for Rust 2024 Edition stability and edition mechanics.

- **Rust 2024 Edition Guide (release version notes)**
  Edition-level changes and the intended 2024-era idioms for writing Rust.

Windows Practical Workflow Reference

- **Cargo and toolchain usage (official documentation)**

Use the official docs as the source of truth for: `cargo new`, `cargo run`, `cargo build -release`, dependency management, and edition selection.

Minimal PowerShell Commands Used Throughout This Booklet

```
# Create a new example project
cargo new example_project
cd example_project

# Run (debug)
cargo run

# Build optimized (release)
cargo build --release

# Run optimized
cargo run --release
```