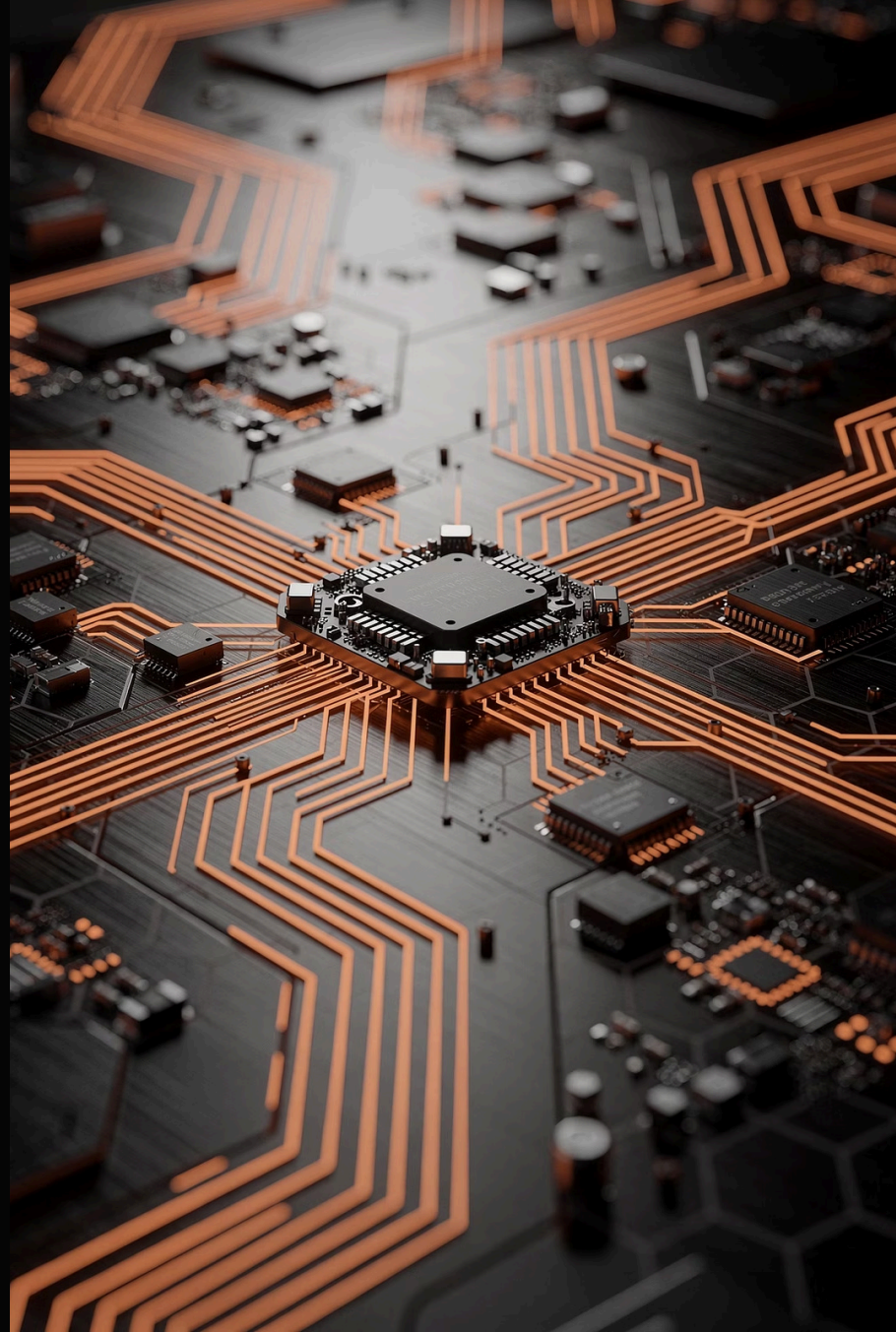
Rust Programming: Systems Reimagined

Module 2: Ownership & Borrowing

The Compiler's Guardian – The Borrow Checker

Chandrashekar Babu · <training@chandrashekar.info>

https://www.chandrashekar.info/ | https://www.slashprog.com/



Module 2 Overview

This module is structured as four focused learning blocks, taking you from flow control fundamentals all the way through to Rust's legendary ownership system and safe borrowing mechanics.

1

Part 2.1 – Flow Control & Pattern Matching

45 min · `if` expressions, `loop`, `while`, `for`, and `match` introduction

2

Part 2.2 – Ownership System Deep Dive

60 min · Ownership rules, memory allocation, Move/Clone/Copy semantics, C++ comparison

3

Part 2.3 – Borrowing & References

45 min · Shared `&T` and mutable `&mut T` references, borrow checker rules, MIR deep dive

4

Part 2.4 – Slices & Lab

30 min · String slices, array slices, and hands-on lab fixing ownership errors

PART 2.1

Flow Control & Pattern Matching

Before diving into Rust's ownership model, we build confidence with flow control — the constructs that direct program execution. Rust's approach is distinctly **expression-based**, making it more functional, composable, and less verbose than C-family languages.

Statements vs. Expressions

This distinction is foundational to understanding how Rust code works. Unlike C or Java, Rust treats most constructs as **expressions** that produce values — not just statements that perform side effects.

Statements

- Perform an action
- Do **not** return a value
- End with a semicolon `;`
- Example: `let x = 5;`

Expressions

- Produce a value
- Can be part of statements
- **No** trailing semicolon
- Examples: `5`, `x + 1`, `if {}`

```
fn main() {
    let x = {       // statement
        let y = 3;   // statement (inside block)
        y + 1       // EXPRESSION — block returns this value
    };              // statement ends
}
```

Rust is expression-based → More functional, less verbose, and highly composable.

The `if` Expression

In Rust, `if` is not just a statement — it is a full expression that returns a value. This enables concise, readable conditional assignments without a ternary operator.

```
fn main() {
    let number = 7;

    // if as an EXPRESSION — returns a value
    let description = if number % 2 == 0 {
        "even"   // no semicolon = expression value
    } else {
        "odd"    // no semicolon = expression value
    };

    println!("{} is {}", number, description);

    // ERROR: both branches must return the SAME type!
    let invalid = if true {
        5        // i32
    } else {
        "five"   // &str — COMPILE ERROR!
    };
}
```

Rule 1

All branches must return the **same type**

Rule 2

The `else` branch is **required** when used as an expression

Rule 3

Each branch's **last expression** determines the return value

The `loop` — Infinite Loop with Control

The `loop` keyword creates an infinite loop that runs until an explicit `break`. Uniquely, Rust allows `break` to carry a return value — making `loop` useful for retry logic and value-producing loops.

```
fn main() {
    // Returning a value from loop
    let mut counter = 0;
    let result = loop {
        counter += 1;
        if counter == 10 {
            break counter * 2;  // Returns value from loop
        }
    };
    println!("Result: {}", result);  // 20
}
```

Retry Logic

Loop until a network request succeeds or a resource becomes available

Event Loops

Continuously listen for and process incoming events in server applications

Game Loops

Run the render–update cycle indefinitely until the player quits

Loop Labels — Breaking Outer Loops

Rust supports **loop labels** — named anchors prefixed with a single quote (`'label`) — allowing you to target a specific loop with `break` or `continue`. This eliminates the need for flag variables or awkward restructuring.

```
fn main() {
    let mut count = 0;
    'outer: loop {
        'inner: loop {
            count += 1;
            if count == 3 { break 'inner; }  // exits only inner
            if count == 5 { break 'outer; }  // exits both loops
        }
        println!("Still in outer loop");
    }
    println!("Exited at count = {}", count);  // 5
}
```

Without Labels

`break` only exits the **innermost** loop. Reaching outer loops requires boolean flags or function calls — both are messy.

With Labels

`break 'label` exits any **specific** labelled loop instantly. Clean, structured, and expressive — like a safe `goto`.

The `while` Loop — Conditional Looping

The `while` loop checks its condition before every iteration and continues as long as it evaluates to `true`. It is the right choice when the number of iterations is unknown at compile time.

```
fn main() {
    let mut number = 3;
    while number != 0 {
        println!("{}!", number);
        number -= 1;
    }
    println!("LIFTOFF!!!");
}
```

> **Equivalent using `loop`:** You can always rewrite a `while` as a `loop` with an inner `if/break`, but `while` expresses the intent more clearly when the condition is known upfront.

When iterating over collections with `while`, you must manually manage the index — which is error-prone. That is exactly why Rust provides the `for` loop as the preferred alternative for collection traversal.

The `for` Loop — The Preferred Way

The `for` loop is Rust's most idiomatic iteration construct. It eliminates off-by-one errors, works with any type implementing `Iterator`, and is zero-cost — compiling down to raw loops.

```
fn main() {
    let arr = [10, 20, 30, 40, 50];

    for element in arr { println!("{}", element); }

    for number in 1..4  { /* 1, 2, 3    exclusive */ }
    for number in 1..=4 { /* 1, 2, 3, 4 inclusive */ }

    for number in (1..4).rev() { /* 3, 2, 1 */ }

    // With index
    for (index, value) in arr.iter().enumerate() {
        println!("Index {}: {}", index, value);
    }
}
```

No off-by-one

Range syntax is unambiguous

Expressive

Intent is always clear

Universal

Works with any iterator

Zero-cost

Optimises to raw loops

The `for` Loop — Under the Hood

The `for` loop is syntactic sugar over Rust's iterator protocol. Understanding the desugaring helps you appreciate how Rust achieves **zero-cost abstractions** — you get expressive, safe code with no runtime penalty.

High-level syntax

```
for element in collection {
    // do something
}
```

Desugars to (approximately)

```
let mut iter =
  IntoIterator::into_iter(collection);
loop {
    match iter.next() {
        Some(e) => { /* do something */ },
        None    => break,
    }
}
```

→ No runtime overhead

Zero-cost abstraction — the compiler eliminates the iterator machinery entirely in optimised builds.

→ Lazy evaluation

Iterators compute values on demand, enabling efficient chaining of operations like `.filter().map()`.

→ LLVM-optimised

The compiler can often vectorise the resulting loop for maximum hardware throughput.

`match` — Pattern Matching Introduced

`match` is one of Rust's most powerful constructs. It is **exhaustive** — the compiler requires every possible value to be handled — and each arm is itself an expression, making it suitable for both control flow and value production.

```
fn main() {
    let number = 3;

    // match as an expression
    let description = match number {
        1 => "one",
        2 => "two",
        3 => "three",
        _ => "many",   // wildcard — catches everything else
    };

    // Multiple patterns in one arm
    match number {
        1 | 2     => println!("One or two"),
        3..=5     => println!("Three through five"),
        _         => println!("Other"),
    }
}
```

Exhaustive

Every possible value must be covered — no forgotten cases

Expression-based

Each arm returns a value; the whole `match` can be assigned

Rich patterns

Ranges, OR patterns, structs, enums, guards — all supported

`match` with Enums — Destructuring Data

`match` shines brightest when paired with enums. It can **destructure** enum variants and bind their payload to local variables in the same step — eliminating the boilerplate getters common in other languages.

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(String),  // carries a state name
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny        => { println!("Lucky penny!"); 1 },
        Coin::Nickel       => 5,
        Coin::Dime         => 10,
        Coin::Quarter(state)=> { println!("From {}!", state); 25 },
    }
}

fn main() {
    let coin = Coin::Quarter("California".to_string());
    println!("{} cents", value_in_cents(coin));
}
```

Pattern matching **destructures** data — the `state` variable is automatically bound to the `String` inside `Quarter`. No manual extraction needed.

`match` with `Option<T>` — Eliminating Null

Rust has no `null`. Instead, optional values are represented by `Option<T>` — an enum with two variants: `Some(T)` and `None`. The compiler **forces** you to handle both, making null pointer exceptions a compile-time impossibility.

```
fn double_if_exists(x: Option<i32>) -> Option<i32> {
    match x {
        Some(value) => Some(value * 2),
        None        => None,
    }
}

fn main() {
    println!("{:?}", double_if_exists(Some(5)));  // Some(10)
    println!("{:?}", double_if_exists(None));      // None

    // Safe extraction — default to 0 if absent
    let result = match Some(5) {
        Some(v) => v,
        None    => 0,
    };

    // This would PANIC at runtime:
    // let dangerous = Some(5).unwrap(); // avoid in production
}
```

Key insight: `match` forces you to handle the `None` case — the compiler will not let you ignore it. Null pointer exceptions are structurally impossible.

`if let` — Concise Single-Pattern Matching

`if let` is syntactic sugar for a `match` that only cares about **one pattern**. It is not safer than `match`, but it is significantly less ceremonial when you need to handle exactly one case and discard all others.

```
fn main() {
    let some_value = Some(7);

    // Verbose match
    match some_value {
        Some(7) => println!("Lucky seven!"),
        _       => (),
    }

    // Concise if let — same behaviour
    if let Some(7) = some_value {
        println!("Lucky seven!");
    }

    // With else branch
    if let Some(7) = some_value {
        println!("Lucky seven!");
    } else {
        println!("Not lucky seven");
    }
}
```

Use `match` when...

You need **exhaustiveness checking** — all possible values must be accounted for. Critical for safety-sensitive logic.

Use `if let` when...

Only **one pattern matters** and you want to discard the rest cleanly. Reduces boilerplate without sacrificing clarity.

Flow Control Summary

A quick-reference overview of all flow control constructs covered so far. The key principle: **Rust prefers expressions over statements**, enabling more functional, composable, and safer code throughout the language.

Construct	Best Use Case	Returns a Value?
`if`	Conditional branching, inline value selection	✓ When used as expression
`loop`	Infinite loops, retry patterns, value-returning loops	✓ Via `break value`
`while`	Condition-driven iteration (unknown count)	✗
`for`	Collection iteration, ranges — preferred default	✗
`match`	Exhaustive pattern matching on any type	✓
`if let`	Single-pattern match — concise sugar for `match`	✗

Design principle: Rust's expression-oriented design means flow constructs can appear inside assignments, function arguments, and return positions — leading to denser, more declarative code.

PART 2.2

The Ownership System

Ownership is Rust's most distinctive feature — a compile-time memory management system that guarantees safety **without** a garbage collector and without manual memory management. Understanding it is the key to mastering Rust.

What Is Ownership?

Ownership is the mechanism by which Rust guarantees memory safety at compile time. Every value has exactly one owner; when that owner goes out of scope, the value is automatically freed. No runtime GC, no manual `free()`.

Rule 1

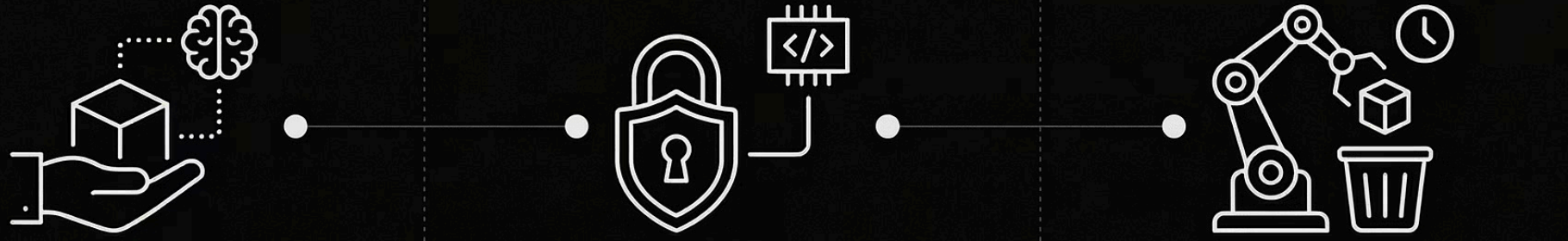
Each value in Rust has an **owner** — a single variable responsible for it

Rule 2

There can only be **one owner at a time** — ownership cannot be shared

Rule 3

When the owner goes **out of scope**, the value is automatically **dropped**



Manual
(C)

**Full control.
Error-prone.
Developer manages memory.**

Rust
(Ownership)

**Compiler-enforced.
Zero-cost.
Safe by default.**

Garbage Collection
(Java/Go)

**Automatic.
Convenient.
Runtime pauses and latency.**

Variable Scope — The Foundation of Ownership

Scope is the region of code where a variable is valid and accessible. In Rust, scope and ownership are intimately linked: a value's lifetime is tied directly to the scope of its owner. When the scope ends, the value is dropped.

```
fn main() {
    // s not valid here — not yet declared

    let s = "hello";  // s is valid from this point forward

    // do stuff with s
    println!("{}", s);

} // scope ends — s is no longer valid, memory freed

// Nested scopes
fn main() {
    let x = 5;
    {
        let y = 10;
        println!("x={}, y={}", x, y);  // Both accessible
    }   // y is DROPPED here
    // println!("{}", y);  // ERROR: y not in scope
    println!("x={}", x);   // OK — x still valid
}
```

Key insight: Scope determines *where* a variable is accessible. Ownership determines *when* the underlying value is freed. These two concepts work together seamlessly.

Memory & Allocation — Stack vs. Heap

Understanding where data lives is essential to understanding ownership. Rust distinguishes between stack-allocated and heap-allocated values, and the ownership system handles cleanup for both — automatically and correctly.

Stack Allocation

```
let x = 5;
```

- Fixed size, known at compile time
- Extremely fast — just move the stack pointer
- Automatic cleanup (LIFO order)
- Examples: integers, booleans, chars, fixed arrays

Heap Allocation

```
let s = String::from("hello");
```

- Flexible, growable size
- Slower — requires OS allocation request
- Requires explicit deallocation in other languages
- Examples: `String`, `Vec`, `Box`

Rust's innovation: Heap memory is automatically freed when the owning variable goes out of scope — with *zero* runtime overhead and *zero* risk of double-free or use-after-free bugs.

Ownership in Action — The Move Semantic

When you assign a heap-allocated value to a new variable, Rust performs a **move** — transferring ownership. The original variable is immediately invalidated. This prevents double-free errors without any runtime cost.

Stack (Copy)

`let x = 5; let y = x;` → Both valid. Bitwise copy, no ownership transfer.

Heap (Move)

`let s2 = s1;` → s1 invalidated. One owner, one free.

```
fn main() {
    // Heap-allocated: ownership moves
    let s1 = String::from("hello");
    let s2 = s1;  // s1 is MOVED — no longer valid
    // println!("{}", s1);  // COMPILE ERROR: value moved
    println!("{}", s2);     // OK
} // s2 dropped, heap freed — no double-free possible!
```

1

s1 created

s1 owns heap data "hello"

2

let s2 = s1

ownership moves to s2; s1 invalidated

3

Scope ends

s2 dropped, heap freed. No double-free.

The compiler statically tracks ownership transfers. After a move, the original variable is erased from the compiler's mental model — any attempt to use it is a compile-time error, not a runtime crash.

The `clone()` Operation — Explicit Deep Copy

When you genuinely need two independent copies of heap data, use `.clone()`. This performs a full deep copy — allocating new heap memory and copying the contents. It is intentionally **explicit** because it is **expensive**.

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1.clone();  // deep copy — new heap allocation

    // Both are valid now!
    println!("s1={}, s2={}", s1, s2);
}

// Custom types: derive Clone
#[derive(Clone)]
struct Person {
    name: String,
    age: u32,
}

fn main() {
    let p1 = Person { name: "Alice".to_string(), age: 30 };
    let p2 = p1.clone();  // deep copies ALL fields including String
}
```

Move (default)	Clone (explicit)
Transfers ownership. No heap copy. Original invalidated. **Free.**	Deep copies all heap data. Both values valid. **Costly — allocates.**

The `Copy` Trait — Stack-Only Types

Simple, fixed-size types that live entirely on the stack implement the `Copy` trait. When you assign or pass them, a **bitwise copy** happens implicitly — no move, no invalidation. No heap means no ownership complexity.

```
fn main() {
    let x = 5;
    let y = x;  // COPY, not move
    println!("x={}, y={}", x, y);  // Both valid!
}
```

Types that implement Copy

- All integers: `i32`, `u64`, etc.
- Floats: `f32`, `f64`
- Booleans: `bool`
- Characters: `char`
- Tuples of Copy types: `(i32, i32)`
- Fixed arrays of Copy types: `[i32; 5]`

Types that do NOT implement Copy

- `String` — owns heap memory
- `Vec<T>` — owns heap memory
- Any type that implements `Drop`
- Any struct containing non-Copy fields

Ownership & Functions — Arguments and Returns

Passing a value to a function follows **exactly the same rules** as variable assignment: heap types are moved, Copy types are copied. Functions are first-class owners.

```
fn main() {
    let s = String::from("hello");
    take_ownership(s);         // s MOVED into function
    // println!("{}", s);      // ERROR: s no longer valid

    let x = 5;
    make_copy(x);              // x COPIED (i32 is Copy)
    println!("x still: {}", x); // OK
}

fn take_ownership(some_string: String) {
    println!("{}", some_string);
} // some_string dropped — memory freed here

fn make_copy(some_integer: i32) {
    println!("{}", some_integer);
} // some_integer dropped — stack, nothing special
```

The same rules apply whether you are assigning to a variable, passing to a function, or returning from a function. Ownership is a **universal concept**, not special syntax.

Return Values and Ownership Transfer

Functions can **return ownership** to the caller. This pattern — taking ownership in and giving it back — works but is verbose. It directly motivates the need for **references**, covered in the next section.

```
fn gives_ownership() -> String {
    let s = String::from("yours");
    s  // ownership moves to caller
}

fn takes_and_gives_back(s: String) -> String {
    s  // received ownership — returns it to caller
}

fn main() {
    let s1 = gives_ownership();         // gets ownership
    let s2 = String::from("hello");
    let s3 = takes_and_gives_back(s2);  // s2 moved, s3 gets it
    // println!("{}", s2);              // ERROR: s2 moved
    println!("{}", s3);                // OK
}
```

This ownership-passing pattern is **correct but cumbersome**. Having to return every string you borrow from a function is impractical. The solution is **references** — coming up next.

Rust Move vs. C++ Move Semantics

Both Rust and C++ have move semantics, but they differ in a crucial way: Rust's moves are **enforced by the compiler**, while C++'s are advisory at best. This seemingly small difference eliminates an entire class of use-after-move bugs.

Rust — Compiler-Enforced

```
let s1 = String::from("hello");
let s2 = s1;  // MOVE
// Using s1 → COMPILE ERROR
// Cannot proceed — safe by default
```

C++ — Programmer's Responsibility

```
std::string s1 = "hello";
std::string s2 = std::move(s1);
// s1 is "valid but unspecified"
// Using s1 is ALLOWED — but dangerous!
```

Rust: Safe by default

The compiler statically invalidates moved-from variables. Use-after-move is a compile error, not a runtime surprise.

C++: Unsafe by default

Moved-from objects remain in a "valid but unspecified" state. The programmer must remember not to use them — the compiler will not stop them.

Rust Copy vs. C++ Copy Semantics

C++ copies implicitly and often silently, leading to unexpected performance cliffs when large objects are passed by value. Rust takes the opposite stance: **copies are opt-in and explicit**. If it could allocate, you have to say `.clone()`.

C++ — Implicit Deep Copy

```
std::string s1 = "hello";
std::string s2 = s1;  // silent deep copy
// Passing to function:
auto f = [](std::string s) {};
f(s1);  // COPIES — easy to miss!
```

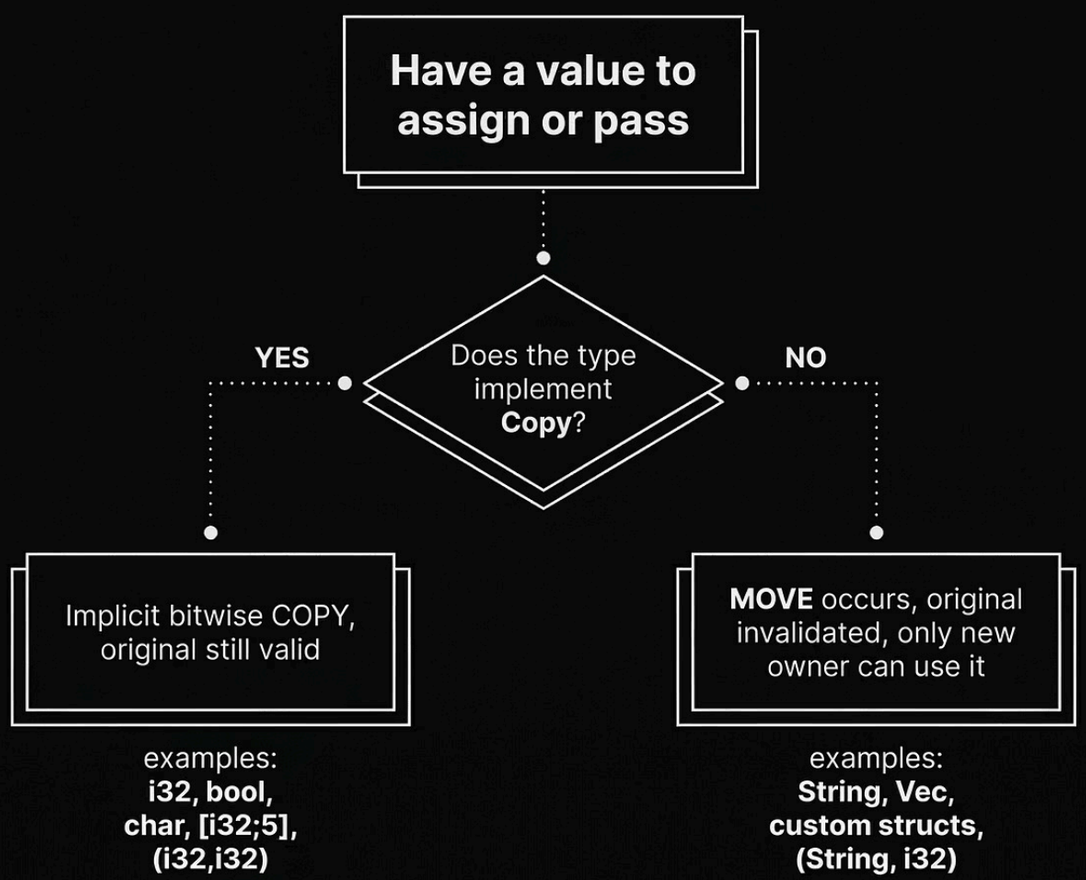
Rust — Explicit Cost

```
let s1 = String::from("hello");
let s2 = s1.clone(); // explicit deep copy
// let s2 = s1;      // MOVES — cheaper!
fn f(s: String) {}
// f(s1);   // moves unless you .clone()
```

Rust's philosophy: Make expensive operations — copies, heap allocations — **visible in the code**. If you see `.clone()`, you know a heap allocation is happening. If you don't see it, you know it's free.

Ownership Decision Tree

When you assign or pass a value, Rust applies a simple decision process based on whether the type implements `Copy`. Internalising this decision tree will make ownership intuitive rather than mysterious.



The rule of thumb: if a type manages heap memory or has a custom `Drop` implementation, it moves. Everything else copies. When in doubt, check whether the type implements the `Copy` trait in the documentation.

Common Ownership Pitfalls

Most beginners make the same four mistakes when learning ownership. Recognising these patterns early will save hours of compiler frustration.

Pitfall 1 — Assuming all types move

`i32`, `bool`, and other primitives are `Copy`. `let y = x;` does *not* invalidate `x` when `x` is an integer. This trips up developers coming from GC languages.

Pitfall 2 — Forgetting `.clone()`

If you need to use a `String` after passing it somewhere, clone *before* the move. You cannot retroactively clone a value that has already been moved away.

Pitfall 3 — Thinking moves are expensive

Moving a `Vec` is cheap — it only copies the small stack struct (pointer, length, capacity). Only `.clone()` copies heap data. Fear of moves is unfounded.

Pitfall 4 — Scope confusion with references

A reference cannot outlive the data it points to. Taking a reference to a local variable and storing it outside that variable's scope is a compile error — the borrow checker will catch it.

PART 2.3

Borrowing & References

References allow you to **use a value without taking ownership**. This is Rust's primary tool for avoiding the verbose ownership-passing pattern seen in the previous section. The borrow checker ensures all references are always valid.

The Problem References Solve

Without references, every function that needs to read a value must take full ownership — and then return it back to the caller. This is both tedious and error-prone. References eliminate the problem entirely.

Without references — ownership headache

```
fn calculate_length(s: String)
    -> (String, usize) {
    let len = s.len();
    (s, len)  // must return to give back ownership
}

fn main() {
    let s1 = String::from("hello");
    let (s2, len) = calculate_length(s1);
    println!("'{}' len is {}", s2, len);
}
```

With references — clean and natural

```
fn calculate_length(s: &String) -> usize {
    s.len()   // use without owning
}

fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
    // s1 still valid!
    println!("'{}' len is {}", s1, len);
}
```

The `&` operator creates a reference — a pointer to the value that does *not* transfer ownership. The function borrows the value for its duration, then the caller regains full access automatically when the function returns.

Immutable References — `&T`

An immutable reference (`&T`) lets you read a value without owning it. Rust allows **any number** of simultaneous immutable references — because multiple readers can never conflict. Attempting to mutate through an immutable reference is a compile error.

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

    // Multiple immutable references — all OK!
    let r1 = &s;
    let r2 = &s;
    let r3 = &s;  // Any number allowed
    println!("{}, {}, {}", r1, r2, r3);

    // Read-only — cannot modify through &T
    // r1.push_str(" world");  // ERROR: immutable reference
}

fn print_length(s: &String) {
    println!("Length: {}", s.len());
    // s.push_str("!");  // ERROR: immutable reference
}
```

Rule: You can have **any number** of immutable (`&T`) references simultaneously. They are all read-only views — no mutation possible, no conflicts possible.

Mutable References — `&mut T`

A mutable reference (`&mut T`) allows you to modify a value you do not own. The critical constraint: you can have **exactly one** mutable reference to a value at any given time — and no immutable references at the same time.

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

    let r1 = &mut s;
    r1.push_str(" world");
    println!("{}", r1);  // OK

    // Only ONE mutable reference at a time!
    let mut t = String::from("test");
    let r2 = &mut t;
    // let r3 = &mut t;  // ERROR: cannot borrow as mutable
                         // more than once at a time
}

fn modify(s: &mut String) {
    s.push_str("!");  // Can mutate through &mut T
}
```

Immutable `&T`

Many readers simultaneously. Read-only access. No conflicts.

Mutable `&mut T`

Exactly one writer at a time. No concurrent readers allowed.

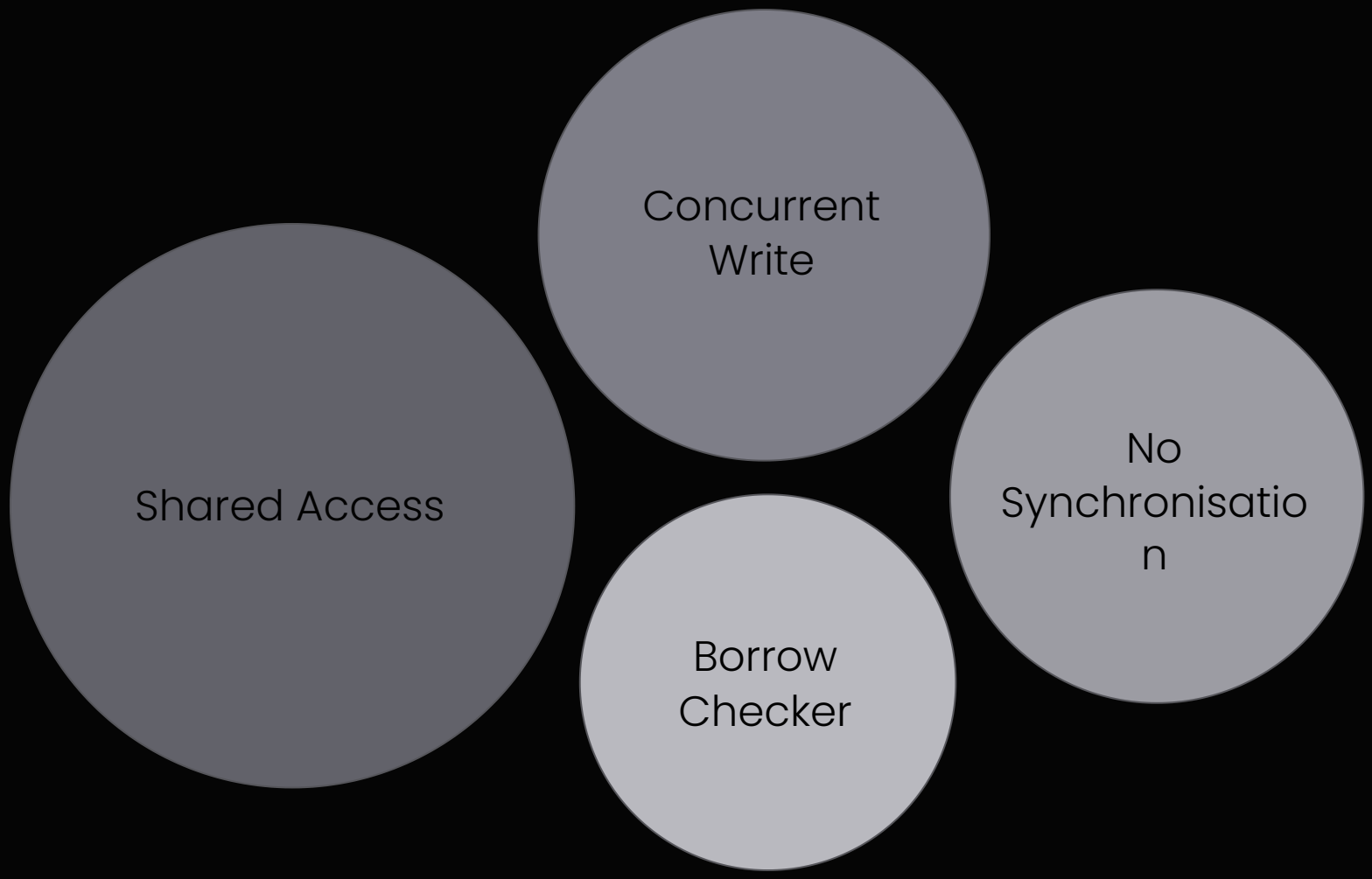
The Three Borrowing Rules

The borrow checker enforces three simple rules that together guarantee freedom from data races and dangling pointers — at compile time, with zero runtime cost.

1	2	3
One mutable OR many immutable	**References must always be valid**	**References cannot outlive their data**
At any point in time, you may have *either* **one mutable reference** (`&mut T`) *or* **any number of immutable references** (`&T`) to a value — but never both at the same time.	No **dangling references** are permitted. A reference's lifetime must never exceed the lifetime of the data it refers to. The compiler tracks this statically.	The compiler ensures that a reference's **scope** is always smaller than or equal to the **owner's scope**. You cannot return a reference to a local variable.

Why These Rules? — Preventing Data Races

Rust's borrowing rules are not arbitrary restrictions — they are a precise, minimal set of constraints that make data races **structurally impossible**. A data race requires simultaneous unsynchronised access where at least one access is a write. The rules eliminate exactly that.



Because the borrow checker enforces that you can never simultaneously hold a mutable reference alongside any other reference to the same data, the three conditions for a data race can never co-exist. This check happens entirely at compile time — there is no mutex, no atomic, no runtime overhead involved in the guarantee.

> Rust's borrowing rules provide **thread-safety guarantees for single-threaded code** that extend naturally to multi-threaded scenarios via the `Send` and `Sync` traits.

Dangling References — What Rust Prevents

A **dangling reference** points to memory that has already been freed. In C and C++, this is a silent runtime hazard. In Rust, it is a compile-time error — the borrow checker proves statically that every reference points to live data.

```
// This would create a dangling reference (Rust rejects it)
fn dangle() -> &String {      // ERROR: missing lifetime
    let s = String::from("hello");
    &s  // s dropped here — reference would be dangling!
}

// CORRECT: Transfer ownership instead
fn no_dangle() -> String {
    let s = String::from("hello");
    s  // ownership moves to caller — valid!
}

fn main() {
    let s = no_dangle();
    println!("{}", s);  // safe — s owns the String
} // s dropped here — clean
```

> The compiler error message even tells you *why* it fails and *how* to fix it. Rust's error messages are famously helpful — they are part of the language's design.

Borrowing Rules Illustrated

The following examples show the three canonical scenarios: multiple immutable borrows (always valid), one mutable borrow (valid when alone), and mixing mutable with immutable (compile error).

✓ Valid — Multiple immutable borrows

```
let s = String::from("hello");
let r1 = &s;  // immutable
let r2 = &s;  // immutable
let r3 = &s;  // immutable — fine!
println!("{} {} {}", r1, r2, r3);
```

✓ Valid — One mutable borrow, alone

```
let mut s = String::from("hello");
let r1 = &mut s;  // mutable — sole reference
r1.push_str(" world");
println!("{}", r1);
```

✗ Invalid — Mixing mutable and immutable

```
let mut s = String::from("hello");
let r1 = &s; // immutable borrow
let r2 = &mut s; // ERROR: mutable borrow
 // while immutable exists!
println!("{}", r1);
```

The Borrow Checker — Rust's Secret Weapon

The borrow checker is the component of the Rust compiler responsible for enforcing all ownership and borrowing rules. It operates at compile time — meaning every check it performs adds **zero overhead** to your running programme.

Tracks Lifetimes

Follows the lifetime of every reference through every code path — including branches and loops

Enforces Exclusivity

Guarantees the "one mutable OR many immutable" rule holds at every point in the programme

Prevents Data Races

Makes concurrent mutation of shared state impossible to express — at compile time

Zero Runtime Cost

All checks are resolved before the binary is produced. No runtime guards, no locks, no overhead

Borrow Checker — Lifetime Regions Visualised

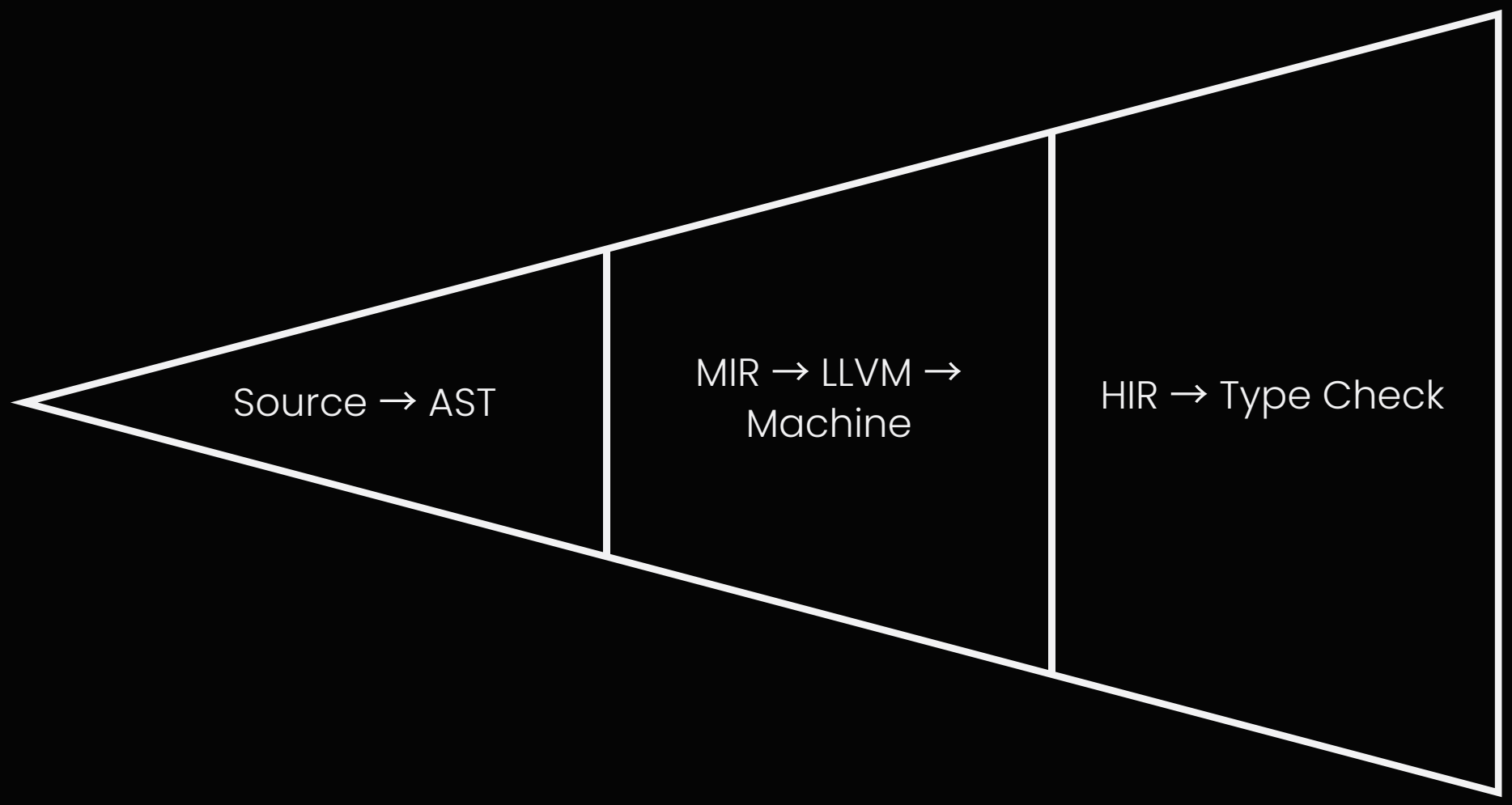
The borrow checker reasons about **regions** — contiguous spans of code where a borrow is active. A mutable borrow can only begin after all previously active borrows have ended. The compiler determines this through **Non-Lexical Lifetimes** (NLL): a borrow ends at its *last use*, not at the end of the syntactic block.

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

    let r1 = &s;      // immutable borrow R1 starts ──┐
    let r2 = &s;      // immutable borrow R2 starts ──┼──┐
    println!("{} {}", r1, r2);  // R1, R2 last use ──┘  │
                          // R1 and R2 END here ──┘

    // Now it is safe to create a mutable borrow
    let r3 = &mut s;  // mutable borrow R3 starts ──────┐
    r3.push_str(" world");                             │
    println!("{}", r3);  // R3 last use ──────────────────┘
} // s dropped
```

Non-Lexical Lifetimes (NLL): Modern Rust uses control-flow analysis rather than lexical scopes to determine borrow end points. This means borrows end at their last *use*, not at the closing `}` of their block — resulting in far fewer false positive errors.

Compiler Deep Dive — MIR

Rust's borrow checking does not happen on the raw source code. It operates on **MIR** (Mid-level Intermediate Representation) — a simplified, explicit control-flow graph that makes ownership and lifetimes unambiguous for the compiler to analyse.




```
// Simplified MIR for: let mut s = String::from("hello");
bb0: {
    _1 = String::from("hello") -> [return: bb1];
}
bb1: {
    _2 = &_1;       // immutable borrow
    _3 = &_1;       // immutable borrow
    _4 = &mut _1;   // BORROW CHECKER FLAGS: _2 and _3 still live!
}
```

At the MIR level, every variable, reference, and control-flow edge is explicit. The borrow checker builds a **borrow graph** over this representation and checks for conflicts. Because MIR is already desugared, complex Rust features like closures and iterators reduce to simple, checkable operations.

How the Borrow Checker Validates Rules

The borrow checker performs **flow-sensitive analysis** — it tracks which borrows are active at each point in the control-flow graph, not just within lexical scopes. This allows it to reason correctly about borrows that end mid-block.

```
let mut x = 5;
let r1 = &x;      // Loan1: immut, active region R1
let r2 = &x;      // Loan2: immut, active region R2
println!("{}", r1); // R1 ends here (last use of r1)
// R2 is still active...
println!("{}", r2); // R2 ends here (last use of r2)
// NOW safe to create a mutable borrow
let r3 = &mut x;  // Loan3: mut — R1 and R2 have ended ✓
```

01

Record the loan

For each borrow, record its kind (mutable/immutable) and the variable it borrows from

02

Track the region

Determine the region of code where the borrow is active — from creation to last use

03

Check for conflicts

For every point where a new loan is created, verify no conflicting active loans exist for the same variable

04

Report errors

If a conflict is found, emit a precise error message pointing to the conflicting borrows

Common Borrow Checker Errors

These are the four most frequent borrow checker errors that beginners encounter. Each has a clear cause and a straightforward fix — the compiler's error messages will guide you directly to the solution.

Error 1 — Multiple mutable borrows

```
let mut s = String::from("hello");
let r1 = &mut s;
let r2 = &mut s;  // ERROR: two mutable borrows
println!("{}", r1);
```

Error 2 — Mutable + immutable borrows mixed

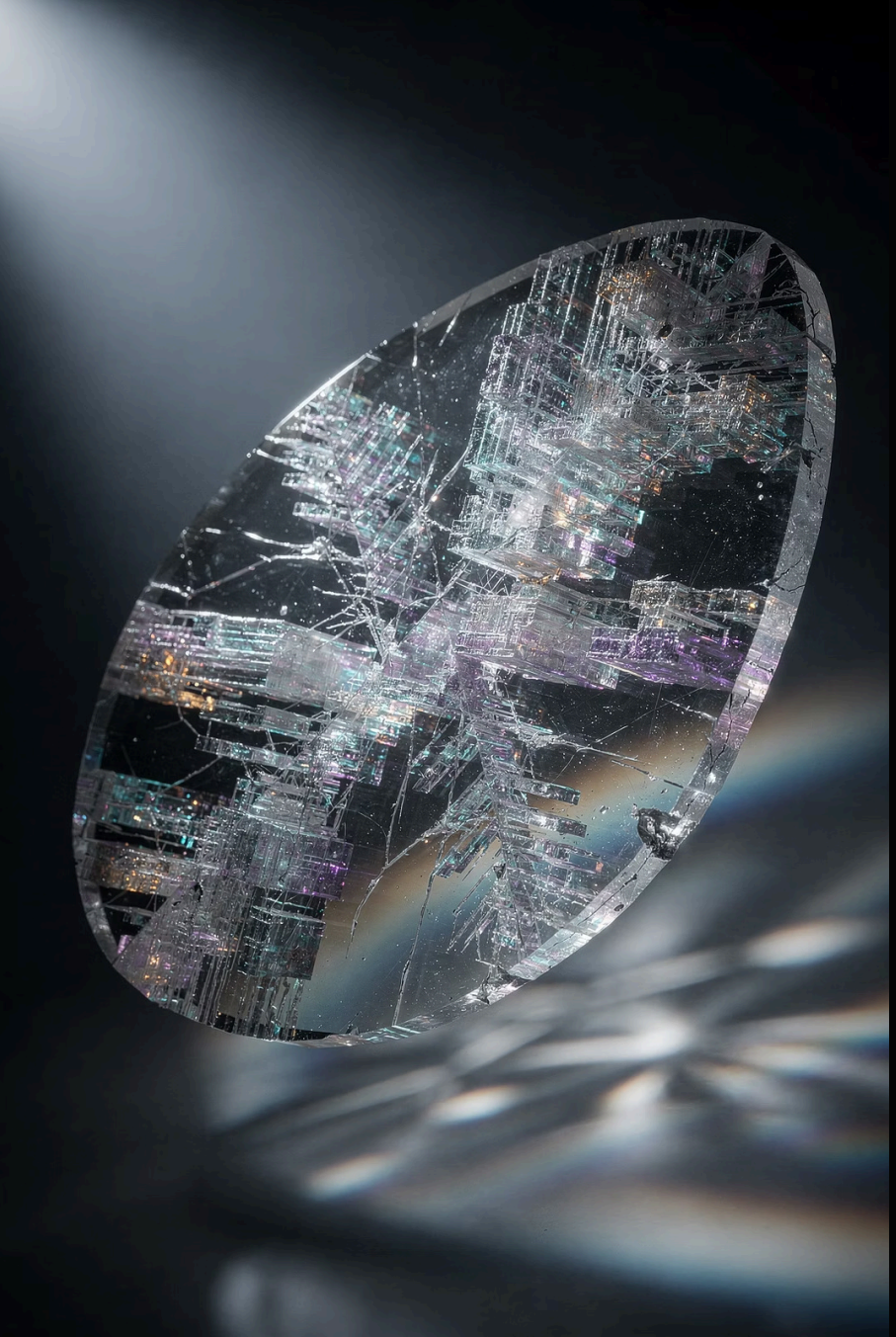
```
let mut s = String::from("hello");
let r1 = &s;      // immutable
let r2 = &mut s;  // ERROR: mutable while immutable exists
println!("{}", r1);
```

Error 3 — Non-obvious active borrow

```
let mut s = String::from("hello");
let r1 = &s;
println!("{}", r1);  // r1 ends here (NLL)
let r2 = &mut s;     // OK — r1 no longer active
```

Error 4 — Dangling reference from function

```
fn dangling() -> &String {
 let s = String::from("hello");
 &s // ERROR: returns ref to local — s is dropped!
}
```



PART 2.4

Slices — Safe Views into Collections

A **slice** is a reference to a contiguous portion of a collection — an array, a vector, or a string. Slices give you a window into data without copying it and without transferring ownership. They are always borrowed references.

What Are Slices?

Slices are **fat pointers** — they carry both a pointer to the start of the data and the length of the slice. This means slices know their own bounds, enabling safe indexing without runtime overhead from separate length tracking.

```
let arr = [1, 2, 3, 4, 5];
//        ↑  ↑  ↑  ↑  ↑
//        0 1 2 3 4  (indices)

let slice = &arr[1..4];  // References indices 1, 2, 3
// slice = [2, 3, 4]     // Does NOT copy data
```

No Ownership

A slice borrows the collection — it never owns or frees the underlying data

Always Borrowed

Slices are references and obey all borrowing rules — borrow checker applies fully

Knows Its Length

The length is stored in the fat pointer — no separate variable needed, bounds are always known

Typed

`&[T]` for array/vector slices; `&str` for string slices

String Slices — &str

String slices (&str) are references into the UTF-8 byte sequence of a String or string literal. Notably, **string literals in Rust are &str slices** — they point directly into the programme's read-only binary data.

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

    let hello = &s[0..5];   // "hello"
    let world = &s[6..11];  // "world"

    // Shorthand syntax
    let from_start = &s[..5];   // "hello"
    let to_end     = &s[6..];   // "world"
    let entire     = &s[..];    // full string

    println!("{} {}", hello, world);

    // String literals ARE &str slices
    let literal: &str = "hello world";
    // Points into the binary — 'static lifetime
}
```

String

Owned, heap-allocated, growable UTF-8 string. Has ptr, len, and capacity on the stack.

&str

Borrowed slice into any string data. Has only ptr and len. Immutable. Zero cost to create.

Array Slices — `&[T]`

Array slices work identically to string slices but for any element type. They are the primary way to write functions that accept arrays, vectors, and other contiguous sequences without committing to a specific collection type.

```
fn main() {
    let arr = [1, 2, 3, 4, 5];
    let slice = &arr[1..4];  // type: &[i32]

    println!("{:?}", slice);   // [2, 3, 4]
    println!("{}", slice.len()); // 3

    fn print_first(slice: &[i32]) {
        if let Some(first) = slice.first() {
            println!("First: {}", first);
        }
    }

    print_first(&arr[..]);        // full array as slice
    print_first(&[10, 20, 30]);   // array literal as slice
    print_first(slice);           // existing slice

    // Fat pointer size: ptr + len = 16 bytes on 64-bit
    println!("{}", std::mem::size_of::<&[i32]>());  // 16
    println!("{}", std::mem::size_of::<&i32>());    // 8
}
```

Writing functions that accept `&[T]` rather than `&Vec<T>` or `&[T; N]` makes them **universally applicable** — they work with arrays, vectors, slices, and anything that coerces to a slice.

String Slices — The Right Approach

The classic example of getting string slices right is the `first_word` function. Returning a slice instead of an index creates a **compile-time link** between the slice and its source string — the borrow checker prevents the source from being mutated while the slice is alive.

```
// BAD: returns an index — can become invalid if String changes
fn first_word_bad(s: &String) -> usize {
    // ...returns index of first space
}

// GOOD: returns a slice — borrow checker enforces validity
fn first_word(s: &str) -> &str {  // accepts &str — more flexible!
    let bytes = s.as_bytes();
    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' { return &s[0..i]; }
    }
    &s[..]
}

fn main() {
    let s = String::from("hello world");
    let word = first_word(&s);
    // s.clear();  // ERROR: can't mutate while slice alive!
    println!("{}", word);  // word guaranteed valid
}
```

Using `&str` as the parameter type rather than `&String` makes the function work with both `String` values and string literals — a strictly more general and idiomatic API.

Slice Rules and the Borrow Checker

Slices are borrows, so all borrowing rules apply. The most important practical consequence: you cannot modify a `String` while a slice into it is active. This prevents a whole class of iterator invalidation bugs that are common in C++.

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

    let word = first_word(&s); // immutable borrow begins
    // s.clear();  // ERROR: cannot mutate s while borrowed
    println!("{}", word); // borrow ends here

    // After last use of slice, mutation is permitted again
    s.clear(); // OK — no outstanding borrows
}

// Mutable slices allow modification through the slice
fn main() {
    let mut arr = [1, 2, 3, 4, 5];
    let slice = &mut arr[1..4];  // &mut [i32]
    slice[0] = 10;
    println!("{:?}", arr);  // [1, 10, 3, 4, 5]
}
```

This pattern eliminates the C++ anti-pattern of calling `erase()` or `clear()` on a container while holding iterators or pointers into it — a common source of undefined behaviour in C++ codebases.

Other Slice Types

Slices are not limited to arrays and strings. Any contiguous heap or stack allocation supports slicing — vectors, custom types stored in arrays, and even empty slices. Mutable slices allow in-place modification through the view.

```
fn main() {
 // Vector slices
 let v = vec![1, 2, 3, 4, 5];
 let slice = &v[1..4];
 println!("{:?}", slice); // [2, 3, 4]

 // Custom type slices
 #[derive(Debug)]
 struct Point(i32, i32);
 let points = [Point(1,1), Point(2,2), Point(3,3)];
 let slice = &points[1..];
 println!("{:?}", slice); // [Point(2, 2), Point(3, 3)]

 // Mutable slice modification
 let mut arr = [1, 2, 3, 4, 5];
 let slice = &mut arr[1..4];
 slice[0] = 10;
 println!("{:?}", arr); // [1, 10, 3, 4, 5]

 // Empty slice
 let empty: &[i32] = &[];
}
```

LAB

Lab Session — Become a Borrow Checker Whisperer

The following exercises are designed to build genuine intuition for Rust's ownership model. Each one presents a broken program — your job is to identify the root cause and apply the correct fix. There is usually more than one valid solution.

Lab Exercise 1 — Fix the Move Error

The most common first encounter with the borrow checker: accidentally using a value after it has been moved. There are three valid approaches, each with different trade-offs.

```
// BROKEN: value used after move
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;
    println!("{}", s1); // ERROR: value borrowed after move
}
```

Solution 1 — Clone

```
let s1 = String::from("hello");
let s2 = s1.clone(); // deep copy
println!("{} {}", s1, s2); // both valid
```

Use when you genuinely need two independent owned values

Solution 2 — Borrow

```
let s1 = String::from("hello");
let s2 = &s1; // borrow, not move
println!("{} {}", s1, s2); // both valid
```

Use when `s2` only needs read access — cheaper than clone

Solution 3 — Reorder

```
let s1 = String::from("hello");
println!("{}", s1); // use before move
let s2 = s1; // move is now safe
```

Use when you only need `s1` before the move and `s2` after

Lab Exercise 2 — Fix the Borrow Error

This exercise tackles the "mixing mutable and immutable borrows" error — one of the most frequently encountered borrow checker errors in real Rust code. The fix involves adjusting the **order** of operations.

```
// BROKEN: immutable borrow still active when mutable borrow created
fn main() {
    let mut s = String::from("hello");
    let r1 = &s;      // immutable borrow
    let r2 = &mut s;  // ERROR: cannot borrow as mutable
    println!("{}", r1);
}
```

Solution 1 — Use r1 before mutable borrow

```
let mut s = String::from("hello");
let r1 = &s;
println!("{}", r1);  // r1's last use — ends here
let r2 = &mut s;     // mutable borrow: OK now!
r2.push_str(" world");
println!("{}", r2);
```

Solution 2 — Explicit scope

```
let mut s = String::from("hello");
{
 let r1 = &s; // immutable borrow
 println!("{}", r1);
} // r1 dropped here — scope ends
let r2 = &mut s; // mutable borrow: OK!
r2.push_str(" world");
println!("{}", r2);
```

Lab Exercise 3 — Fix the Dangling Reference

Attempting to return a reference to a local variable is one of the clearest examples of the borrow checker preventing undefined behaviour. All three solutions transfer the *data* rather than a pointer to data that no longer exists.

```
// BROKEN: returns reference to local variable
fn dangle() -> &String {
    let s = String::from("hello");
    &s  // s is dropped when function returns — dangling!
}
```

Solution 1 — Transfer ownership

```
fn no_dangle() -> String {
    let s = String::from("hello");
    s  // ownership moves to caller
}
```

The most common fix — return the owned value directly

Solution 2 — Tie lifetime to input

```
fn use_string(s: &String) -> &String {
    s  // lifetime tied to input parameter
}
```

Valid when the reference came from outside the function

Solution 3 — Static lifetime

```
fn greeting() -> &'static str {
    "hello"  // lives for the entire programme
}
```

Only for string literals or data with truly global lifetime

Lab Exercise 4 — Write a Safe Function

This open-ended exercise combines everything: slices, references, borrowing, and edge case handling. The goal is to write a function that extracts the first two words from a string — returning slices, not copies.

```
// TASK: Return the first and second words from a string.
// Requirements:
// - Return ("", "") for an empty string
// - Return (first, "") for a single-word string
// - Return (first, second) for two or more words

fn first_two_words(s: &str) -> (&str, &str) {
    // Your implementation here...
}

#[test]
fn test_first_two_words() {
    assert_eq!(first_two_words(""),            ("", ""));
    assert_eq!(first_two_words("hello"),       ("hello", ""));
    assert_eq!(first_two_words("hello world"), ("hello", "world"));
    assert_eq!(first_two_words("hello world rust"), ("hello", "world"));
}

// BONUS: Ensure it works with both &String and &str as input
// Hint: accepting &str already handles both via Deref coercion!
```

This exercise tests lifetime reasoning: the returned `&str` slices borrow from the input `s`. The compiler infers that the output lifetimes are tied to the input — making the function safe without any explicit lifetime annotations in most cases.

Lab Exercise 4 — Solution

The solution uses a single linear pass over the byte representation of the string, tracking word boundaries. Returning slices tied to the input avoids any heap allocation and keeps the function zero-cost.

```
fn first_two_words(s: &str) -> (&str, &str) {
    if s.is_empty() { return ("", ""); }

    let bytes = s.as_bytes();
    let mut first_end: Option<usize> = None;

    for (i, &byte) in bytes.iter().enumerate() {
        if byte == b' ' {
            if first_end.is_none() {
                first_end = Some(i);
            } else {
                // Found space after second word
                let fe = first_end.unwrap();
                return (&s[0..fe], &s[fe+1..i]);
            }
        }
    }

    match first_end {
        None     => (s, ""),          // single word
        Some(fe) => (&s[0..fe], &s[fe+1..]), // two words, no trailing space
    }
}
```

The returned slices borrow from the input `s`. No allocations occur. The borrow checker ensures the caller cannot mutate or drop the source string while these slices are alive — catching bugs that would cause crashes in C or C++.

Module 2 — Key Takeaways & Resources

Core Concepts Mastered

→ Flow Control

`if`, `loop`, `while`, `for` are all expressions — they can return values

→ Pattern Matching

`match` is exhaustive and powerful; `if let` is its concise single-pattern sibling

→ Ownership

One owner, one lifetime, automatic drop. Move by default; Copy for stack-only types

→ Borrowing

Many `&T` OR one `&mut T`. References never outlive their data. Borrow checker enforces this at compile time

→ Slices

Fat-pointer views into collections — safe, zero-cost, always borrowed

Further Reading

- **The Rust Book** — Chapters 3, 4 & 5 (Flow Control, Ownership, References)
- **Rust by Example** — Flow Control, Ownership sections
- **Rustlings** — exercises: `move_semantics`, `borrow`, `lifetimes`

Interactive Tools

- **Rust Playground** — play.rust-lang.org · experiment safely in-browser
- **Rust Visualizer** — visualise stack/heap/ownership state step by step
- **Compiler Explorer** — godbolt.org · inspect MIR and generated assembly

Next up: **Module 3 — Structs, Enums & Error Handling.** The ownership concepts from this module underpin everything that follows — they will feel more natural with each programme you write.