Rust Programming: Systems Programming Reimagined

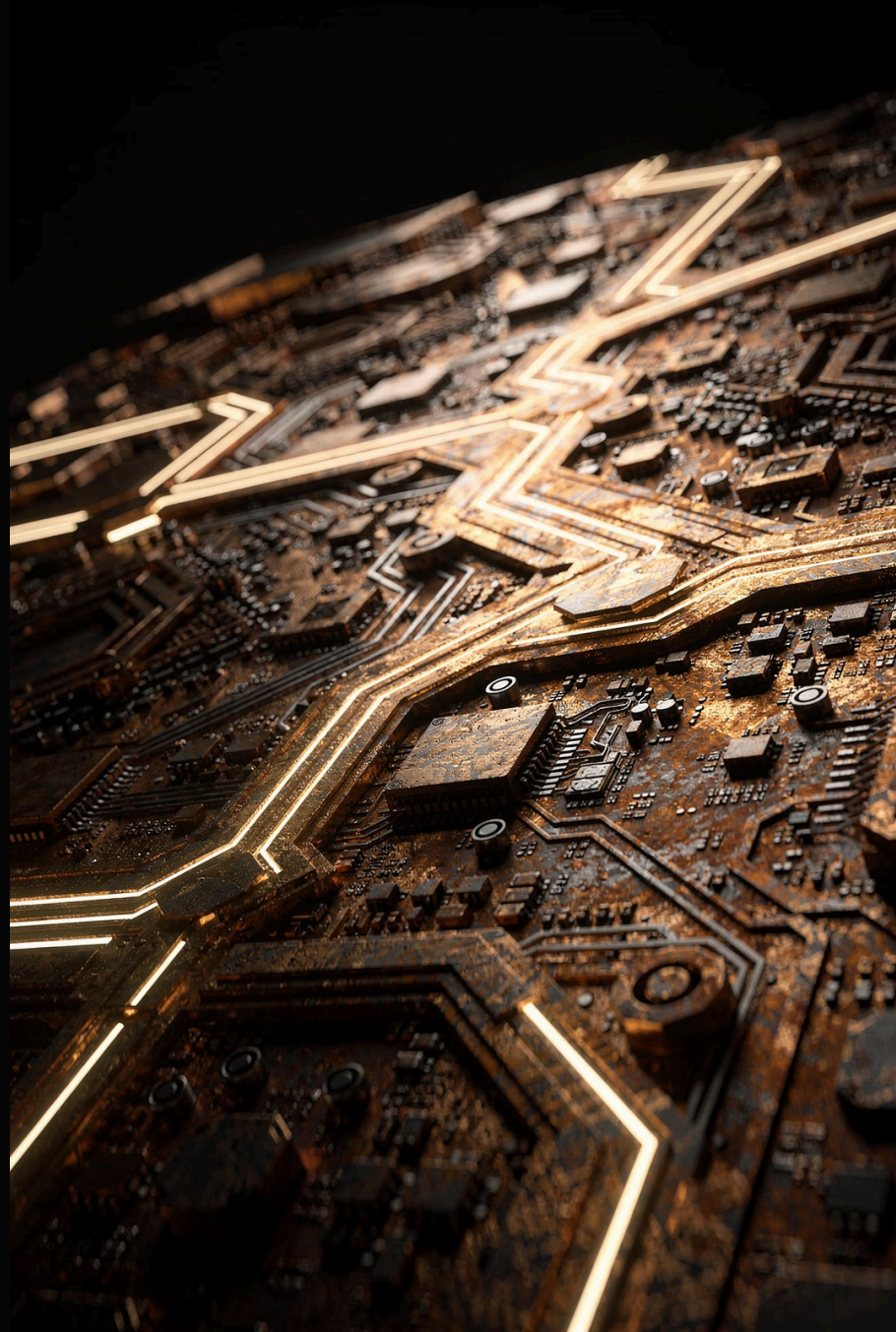
A comprehensive, lab-driven course for experienced C/C++ engineers ready to embrace memory safety, fearless concurrency, and zero-cost abstractions — without sacrificing performance.

Instructor: **Chandrashekar Babu <training@chandrashekar.info>**

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

INTERMEDIATE / ADVANCED

17.5 HOURS · 5 MODULES



Course at a Glance

5	3.5	17.5	100%
Modules	Hours / Day	Total Hours	Lab-Integrated
One per day, each building on the last	Instructor-led training per session	Including hands-on labs	Every concept reinforced through code

This intensive programme is structured as five full days of instructor-led training, each module pairing conceptual depth with immediately applicable coding exercises. The pacing ensures engineers leave each session with working, tested Rust code — not just theory.

Who Is This Course For?

Systems Engineers

Developers building new, safe, and performant systems software who want to leverage Rust's guarantees from day one.

C/C++ Practitioners

Engineers seeking to modernise their tooling and eliminate entire classes of bugs — data races, use-after-free, buffer overflows.

Architects & Tech Leads

Technical leaders evaluating new paradigms for safe concurrency, correctness-by-construction, and long-term system design.

Compiler-Curious Devs

Developers who want to understand how compiler-level design enforces application safety and correctness at zero runtime cost.

Prerequisites

This is an intermediate-to-advanced track. Participants should arrive with the following foundations already in place:

1 Programming Fundamentals

Comfort in at least one language — C, C++, Java, Python, or Go. Ability to write, compile, and debug a multi-function programme independently.

2 Structured & OOP Concepts

Solid understanding of functions, control flow, and basic data structures. Familiarity with classes or structs as compound types is expected.

3 Memory & Compilation Basics

A working conceptual model of stack versus heap allocation, and an understanding of what a compiler does — even if not how it works internally.

Course Objectives

Upon completing this programme, participants will be able to:

01

Articulate Rust's Design Philosophy

Compare and contrast Rust with C/C++ across safety, performance, and developer ergonomics dimensions.

02

Manage the Rust Toolchain

Use rustup and Cargo to build, test, benchmark, and deploy projects with professional-grade tooling.

03

Master Ownership & Lifetimes

Write memory-safe code without a garbage collector by internalising ownership, borrowing, and lifetime annotations.

04

Build Robust Abstractions

Leverage Rust's type system, traits, and pattern matching to create expressive, reusable, and correct APIs.

05

Implement Safe Concurrency

Write concurrent programmes using Rust's threading and message-passing primitives with compile-time safety guarantees.

06

Structure Large Projects

Organise code using Rust's module and crate system for maintainability and team scalability.

07

Understand the Compiler Pipeline

Navigate the front-end, MIR, borrow-checking, and LLVM backend stages and explain how they enforce guarantees.

What Is Out of Scope

To maintain depth and focus, the following topics are explicitly excluded from this programme. Dedicated follow-on courses cover each area.

Third-Party Frameworks

Async runtimes such as `tokio`, web frameworks like `actix-web`, and GUI toolkits such as `gtk-rs` are not covered. The course focuses on the standard library and language fundamentals.

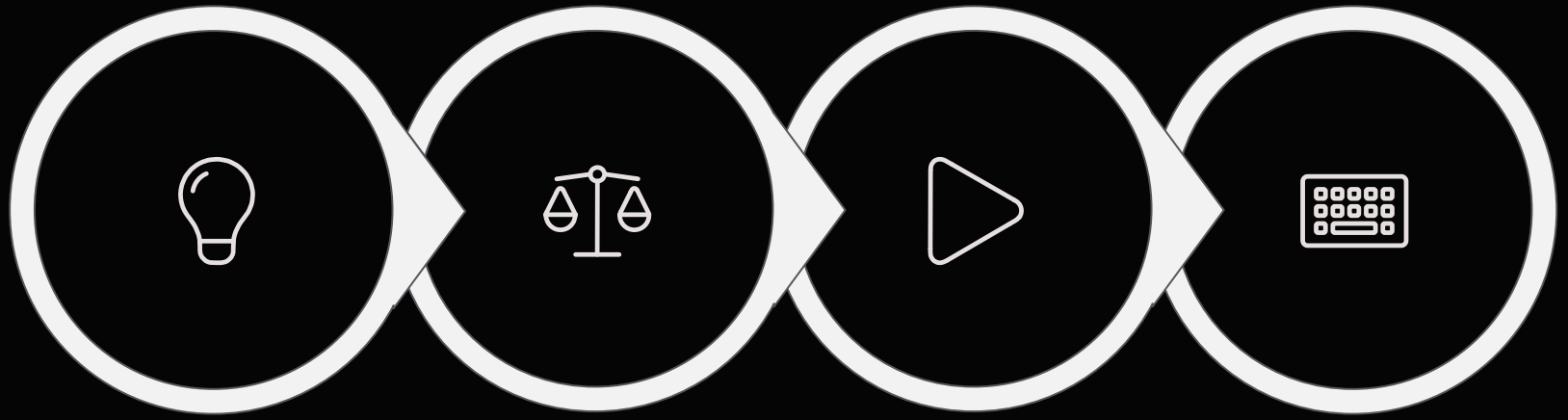
Domain-Specific Applications

Web development, native GUI applications, and deep FFI/C interop beyond basic `unsafe` blocks and `extern` declarations are out of scope for this programme.

Embedded / no_std Rust

Bare-metal development, `#![no_std]` environments, and microcontroller targets require a separate specialised course and are not addressed here.

Training Methodology



Concept Compare Demonstrate Practice

Every topic in this course follows a consistent four-stage learning loop. Concepts are never left abstract — each one is immediately contrasted with equivalent C/C++ patterns, demonstrated in live code by the instructor, and reinforced through a focused lab exercise participants complete during the session.

Lab Requirements

Each participant must arrive with a personal laptop configured and verified before Day 1 begins. The following setup is required:

Rust Toolchain via rustup

Install via the stable channel using the official script:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Verify with `rustc --version` and `cargo --version`.

IDE / Editor

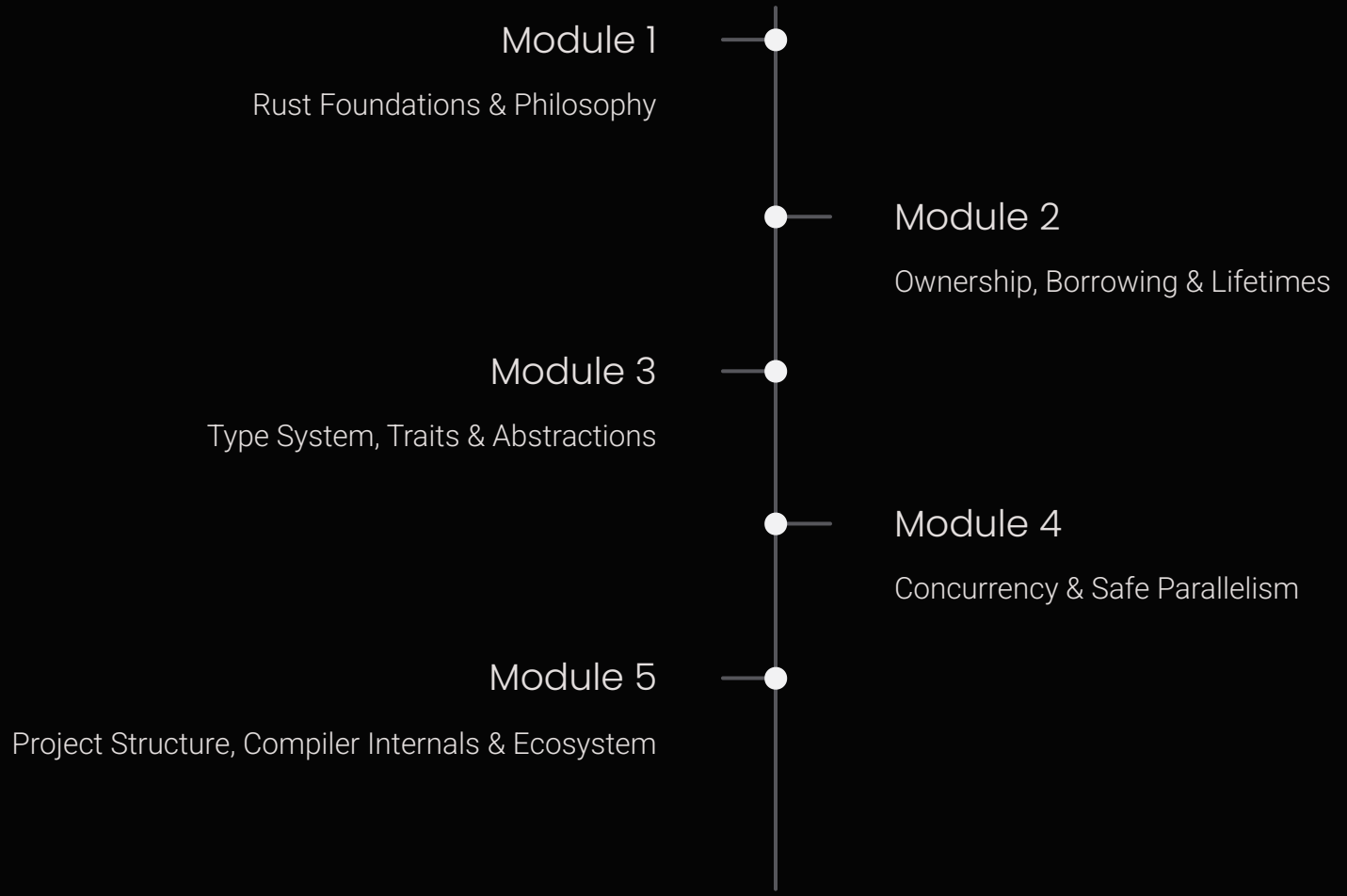
Visual Studio Code with the **rust-analyzer** extension is **strongly recommended**. It provides inline type hints, error diagnostics, and auto-completion that significantly accelerate learning. Any programmer-friendly editor is acceptable as a fallback.

Terminal / Shell

A standard command-line environment is required. Bash, Zsh, or PowerShell (Windows) are all compatible. Participants should be comfortable navigating directories and running commands from the terminal.

COURSE AGENDA

Course Agenda



MODULE 1

Rust Foundations & The Rust/C/C++ Philosophy

The opening module establishes the intellectual foundation for the entire course. Rather than jumping straight into syntax, we begin by asking *why* Rust exists and what specific, measurable problems it solves — grounded in the lived experience of C and C++ engineers.

Part A

Getting Started with Rust

Part B

Core Syntax & Memory Model Preview

MODULE 1 · PART A

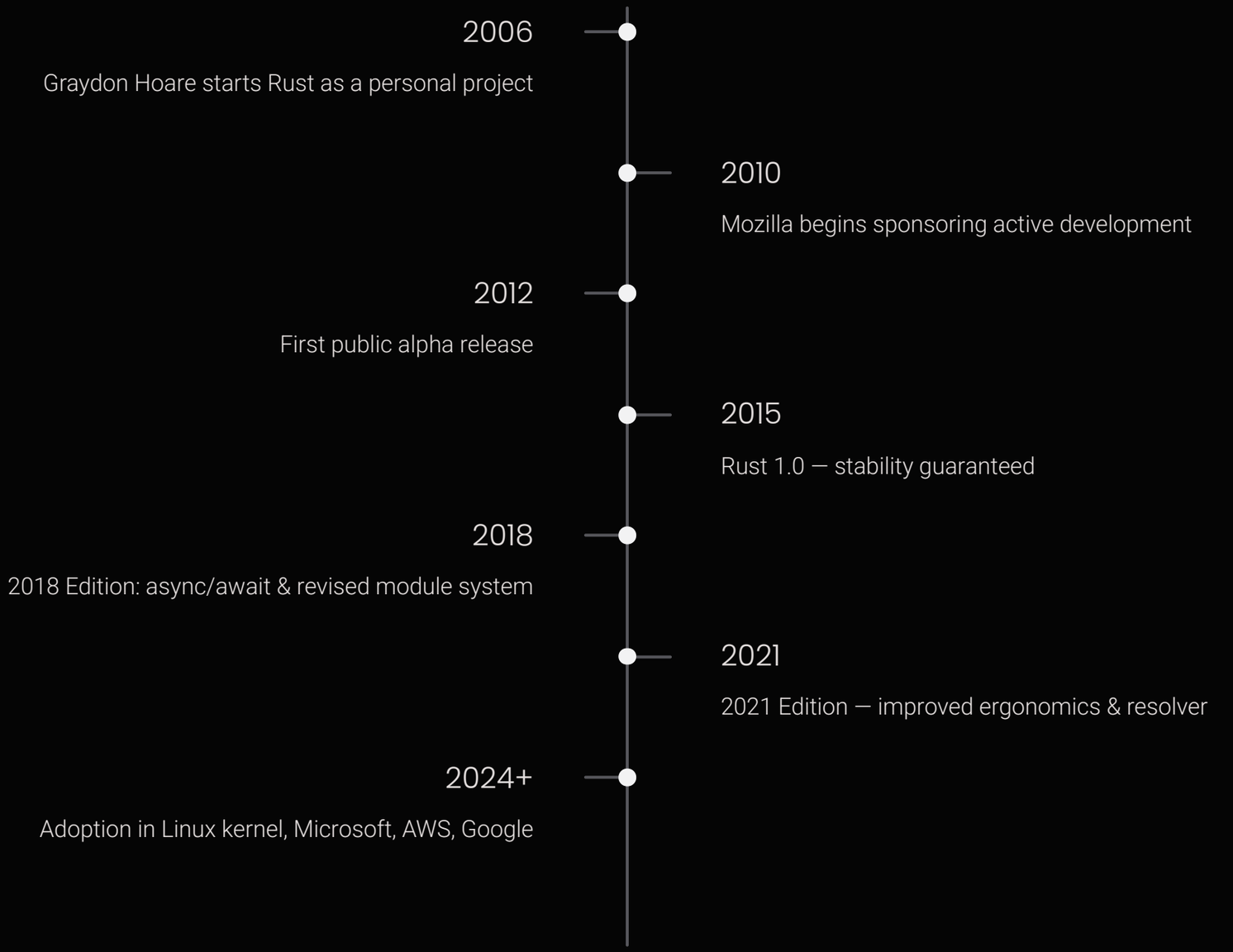
Rust's Genesis & Design Goals

Rust originated as a personal project at Mozilla Research in 2006, motivated by a browser engine bug rooted in memory unsafety. It became Mozilla's answer to a fundamental question: *can we build a systems language that is as fast as C++ but categorically safe?*

The three primary design goals — **Safety, Speed, and Concurrency** — are not independent features bolted together. They are consequences of a single core insight: if the type system tracks ownership and aliasing, the compiler can statically verify memory and thread safety at zero runtime cost. Every other feature in Rust flows from this insight.

Rust's Genesis — A Brief History

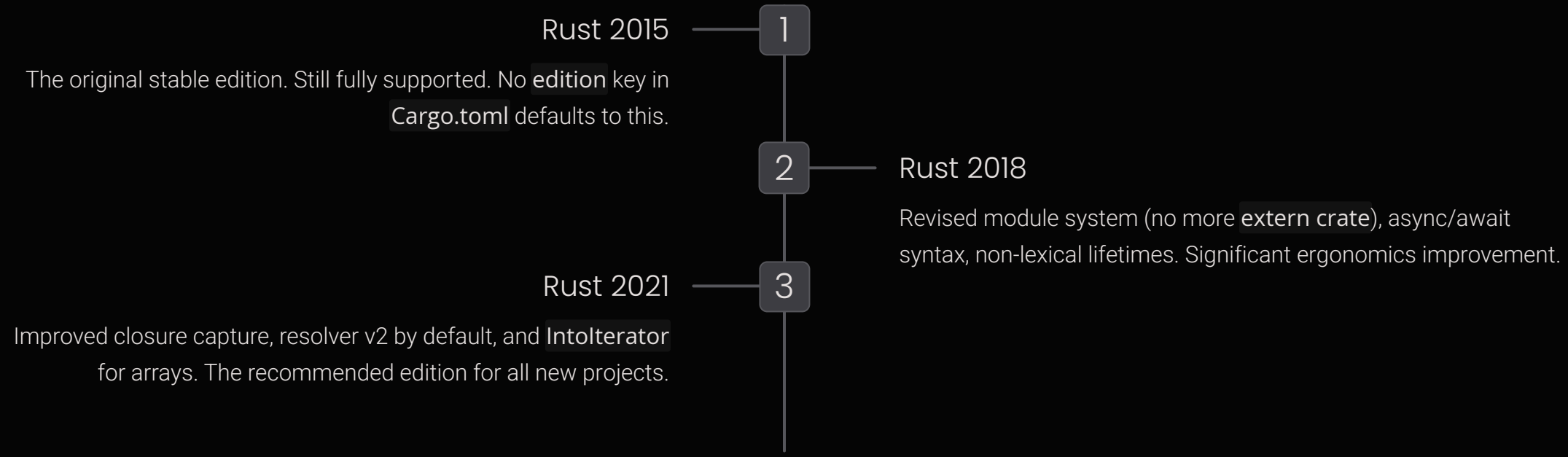
Rust began as a personal research project and grew into one of the most influential systems languages of our era. Each milestone reflects a deliberate commitment to correctness and stability.



> "Rust was born out of frustration with the trade-offs in existing systems languages."

Rust Editions Explained

Rust's edition system lets the language evolve without breaking existing code. Every crate declares its edition independently — old and new editions can interoperate within the same project.



Use `cargo fix --edition` to automatically migrate most code between editions. The toolchain handles nearly all breaking changes mechanically — you rarely need to touch anything by hand.

Why Rust? — Industry Adoption in 2024

Rust is no longer a curiosity. It is shipping in production at some of the largest software organisations in the world, in contexts where correctness and performance are non-negotiable.

Linux Kernel

Rust became the second officially supported language in the Linux kernel (v6.1, 2022). New driver code may now be written in Rust.

Microsoft

Rewrites of core Windows components in Rust are underway. Microsoft has publicly endorsed Rust as the preferred replacement for C/C++ in security-sensitive code.

AWS

Firecracker VMM (the engine behind AWS Lambda and Fargate) is written entirely in Rust. AWS is a major sponsor of the Rust Foundation.

Google

Rust is an officially supported language for Android. Google reports a 21× reduction in memory safety vulnerabilities in new Rust-written Android components.

Rust vs. C++ — A Practical Comparison

For experienced C++ developers, the most useful frame is not "Rust instead of C++" but "Rust resolves the problems that make C++ dangerous at scale." The semantics map closely; the safety does not.

Feature	C++	Rust
Memory management	Manual / RAII / smart pointers	Ownership & borrow checker
Null safety	Raw pointers can be null	`Option<T>` — no null
Data races	Detected at runtime (if at all)	Compile-time prevention
Error handling	Exceptions or error codes	`Result<T, E>` — exhaustive
Undefined behaviour	Possible, often silent	Eliminated in safe Rust
Build system	CMake, Make, Bazel…	Cargo — unified, opinionated
Package manager	vcpkg, Conan (not universal)	crates.io — first-class
Runtime / GC	No GC, optional runtime	No GC, minimal runtime

The Three Pillars of Rust

Every design decision in Rust traces back to one or more of these three foundational goals. They are not independent — they reinforce each other at the language level.

🛡️ Safety

Memory safety **without** a garbage collector. No null pointer dereferences, no dangling pointers, no buffer overflows — all enforced at compile time.

⚡ Speed

Zero-cost abstractions and LLVM-powered code generation deliver performance on par with hand-tuned C and C++. You pay only for what you use.

🔀 Concurrency

Fearless concurrency means the ownership model prevents data races at compile time. Write parallel code with confidence — if it compiles, it is race-free.

> "The Holy Trinity of Systems Programming — and for the first time, you can have all three."

Design Goal #1 — Safety

Rust's safety model eliminates entire categories of bugs that have plagued C and C++ for decades. Crucially, these guarantees come with **zero runtime overhead** — all checking happens at compile time.

Rust Guarantees at Compile Time

- No null pointer dereferencing
- No dangling pointers
- No buffer overflows
- No use-after-free errors
- No double-free errors

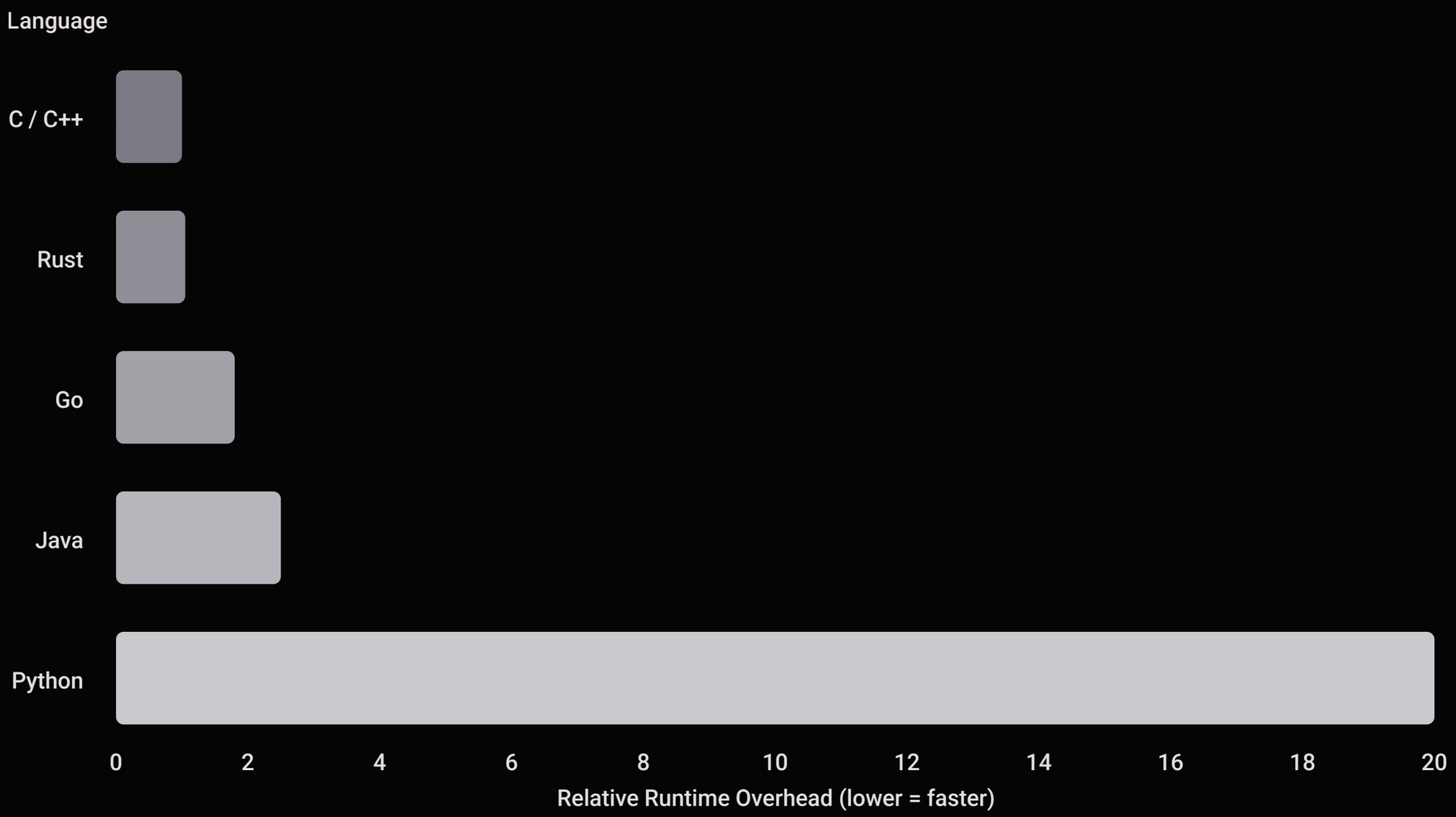
Traditional Approaches & Their Costs

- **Java:** GC pauses at unpredictable times
- **Python:** Runtime checks slow execution
- **C/C++:** Undefined behaviour, no guarantees
- **Go:** Runtime bounds checks add overhead
- **JavaScript:** JIT complexity obscures bugs

> "Safe code doesn't mean slow code. Safe code means *correct* code."

Design Goal #2 — Speed

Rust achieves performance comparable to C and C++ by eliminating the runtime overhead found in managed languages. There is no garbage collector, no interpreter, and no virtual machine standing between your code and the hardware.



Key enablers: **zero-cost abstractions**, no runtime or GC, **LLVM optimisation backend**, and full control over memory layout.

Design Goal #3 — Fearless Concurrency

Concurrency bugs are notoriously difficult to reproduce and fix. Rust relocates the entire class of **data-race errors** from runtime to compile time, so you catch them before a single byte of machine code is executed.

Traditional Problems

- Data races — silent corruption
- Deadlocks from incorrect locking
- Race conditions — nondeterministic output
- Heisenbugs that vanish under debuggers
- Complex, error-prone manual synchronisation

Rust's Solution

- Ownership prevents mutable shared state
- Compiler checks thread safety statically
- `Send` and `Sync` traits enforced at compile time
- Standard library types: `Arc`, `Mutex`, channels

> "Fearless concurrency means your code is free of data races — not necessarily free of deadlocks, but the compiler has your back on the hard part."

The C/C++ Problem Space

Rust's design was explicitly motivated by the categories of bugs that plague production C and C++ systems. Understanding this problem space is essential context for every concept in the course.

Memory Errors

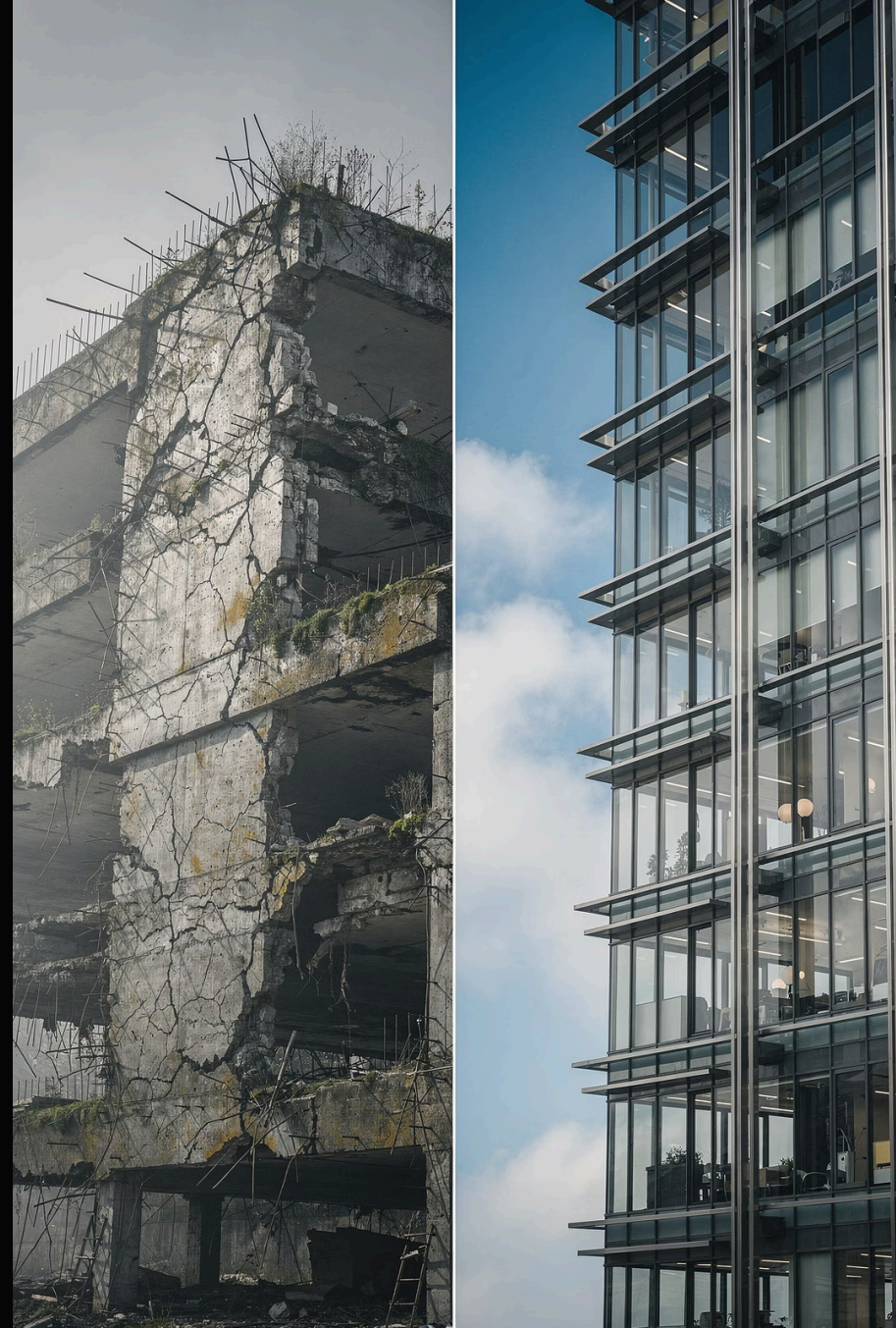
Use-after-free, double-free, null pointer dereferences, and buffer overflows — responsible for the majority of CVEs in systems software.

Data Races

Concurrent access to shared mutable state without synchronisation produces undefined behaviour that is notoriously difficult to reproduce and diagnose.

Undefined Behaviour

C/C++ specifications permit the compiler to optimise aggressively around UB, leading to subtle, non-deterministic failures in production.



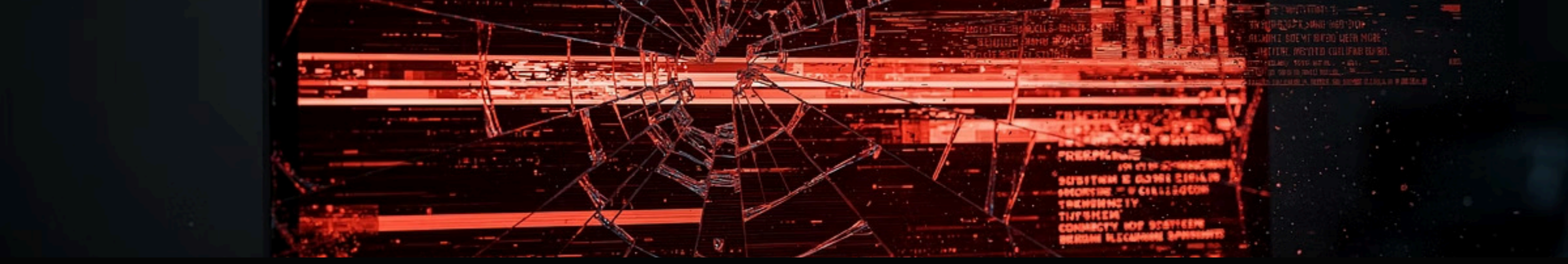
How Rust Responds to These Challenges

The C/C++ Approach

- Manual `malloc`/`free` or `new`/`delete`
- Programmer discipline as the safety layer
- Sanitisers and static analysers as post-hoc tools
- Data race detection at runtime (e.g., ThreadSanitizer)
- Undefined behaviour silently exploited by optimisers

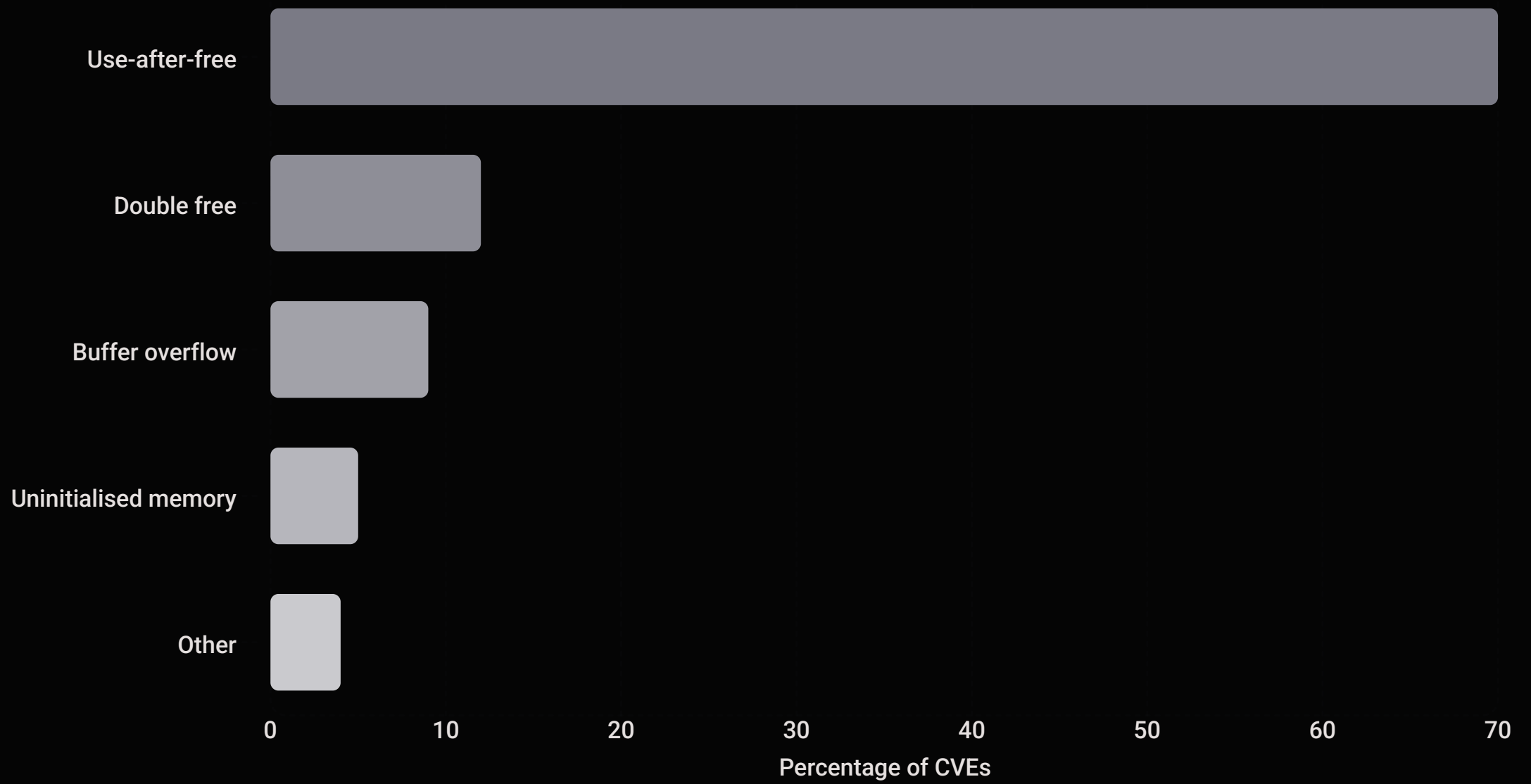
The Rust Approach

- Ownership system manages allocation and deallocation automatically
- Borrow checker enforces aliasing rules at compile time
- No separate tooling needed — safety is built into the compiler
- Data races are a **compile-time error**, not a runtime event
- No undefined behaviour in safe Rust code



The C/C++ Problem Space — Memory Errors

A landmark Microsoft study in 2019 found that roughly **70% of all security vulnerabilities** in their systems software were caused by memory safety issues. These are not edge cases — they are endemic to the C/C++ memory model.



Rust eliminates all five of these categories through its ownership and type system — at compile time, with no runtime cost.

DEMO

The Classic C Bug — Use After Free

This snippet compiles cleanly with no warnings under `gcc` by default. Yet it exhibits **undefined behaviour**: the freed memory may be reallocated, overwritten, or still readable depending on the allocator state. In production it becomes a security vulnerability.

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

int main() {
    int *ptr = (int*)malloc(sizeof(int));
    *ptr = 42;
    free(ptr);

    // UNDEFINED BEHAVIOUR — compiler won't warn you
    printf("Value: %d\n", *ptr);
    return 0;
}
```

Crash Immediately

Segfault on some platforms

Silent Corruption

Reads garbage or overwrites live data

"Works" in Testing

Fails only in production under load

Security Exploit

Enables heap-spraying attacks

DEMO

The C/C++ Problem Space — Data Races

Two threads incrementing a shared counter without synchronisation is a textbook race condition. The final value is **nondeterministic**, the programme compiles without any warning, and the bug may not reproduce under a debugger.

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

int counter = 0; // Shared mutable state — danger!

void increment() {
    for (int i = 0; i < 1_000_000; i++)
        counter++; // RACE CONDITION: read-modify-write is not atomic
}

int main() {
    std::thread t1(increment);
    std::thread t2(increment);
    t1.join(); t2.join();
    // Expected: 2,000,000  Actual: ???
    std::cout << counter << std::endl;
}
```

On a modern multicore CPU this program will almost never print 2,000,000. The result varies across runs, across compilers, and across optimisation levels — making it extremely hard to diagnose.

RUST SOLUTION

Rust's Answer — Ownership Prevents the Bug

Rust does not merely detect the race at runtime — it **refuses to compile** code that could produce one. The borrow checker statically verifies that mutable state is never shared across threads without explicit synchronisation.

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

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

    let handles: Vec<_> = (0..2).map(|_| {
        let c = Arc::clone(&counter);
        thread::spawn(move || {
            let mut num = c.lock().unwrap();
            *num += 1_000_000;
        })
    }).collect();

    for h in handles { h.join().unwrap(); }
    println!("Result: {}", *counter.lock().unwrap()); // Always correct
}
```

C/C++ Approach

"Here be dragons" → runtime crashes, undefined behaviour, security CVEs

Rust Approach

"Here be the compiler" → compile-time errors, forced correctness, zero surprises

The Rust Toolchain

Rust ships as a tightly integrated set of tools. Understanding the role of each component helps you navigate the ecosystem confidently from day one.

rustup — Toolchain Manager

Installs and updates Rust releases. Switches between `stable`, `beta`, and `nightly` channels. Manages cross-compilation targets with a single command.

rustc — The Compiler

Parses and type-checks your code, then hands off to LLVM for optimised machine code. Famous for its **detailed, actionable error messages** that point directly to the problem and suggest fixes.

cargo — Build System & Package Manager

Fetches dependencies from **crates.io**, compiles your project, runs tests, generates documentation, and publishes packages — all with a single unified interface.

MODULE 1 · PART A

Installing & Configuring the Toolchain

rustup — The Toolchain Manager

`rustup` is the official installer and version manager for Rust. It manages multiple toolchain versions side-by-side, making it trivial to pin a project to a specific Rust version or switch to nightly for experimental features.

```
curl --proto '=https' --tlsv1.2 -sSf \
  https://sh.rustup.rs | sh

rustup update stable
rustup show
```

rustc & cargo

`rustc` is the Rust compiler. You will rarely invoke it directly — `cargo`, Rust's build system and package manager, wraps it with a rich set of commands:

- `cargo new` — scaffold a new project
- `cargo build` — compile the project
- `cargo test` — run all tests
- `cargo run` — build and execute
- `cargo doc` — generate HTML documentation

INSTALLATION

Getting Rust Installed — Quick Setup

The official installer, `rustup`, handles everything: the compiler, Cargo, the standard library, and documentation. A single command is all you need on Unix-like systems.

```
# Unix / Linux / macOS
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

# Windows: download the installer from
# https://rustup.rs
```

After installation, open a new terminal and verify:

```
rustc --version    # rustc 1.93.0 (stable)
cargo --version    # cargo 1.93.0
rustup --version   # rustup 1.93.0

rustup update      # keep everything up to date
rustup doc         # browse docs offline in your browser
```

The entire toolchain — compiler, package manager, standard library source, and offline documentation — is installed in `~/.rustup` and `~/.cargo`. No system-wide changes required.

Toolchain Management with rustup

Production teams commonly pin to `stable`, while library authors test against `beta` and `nightly` to catch breaking changes early. `rustup` makes switching between them trivial.

```
# Install additional channels
rustup install stable
rustup install beta
rustup install nightly

# Set the global default
rustup default stable

# Override for a single project directory
cd my_project
rustup override set nightly

# Add a cross-compilation target
rustup target add arm-unknown-linux-gnueabihf
```

stable

Production use. New release every 6 weeks. Backwards-compatible.

beta

Preview of the next stable. Test your crate against upcoming changes.

nightly

Experimental features. Required for some procedural macro and compiler internals work.

The Rust Ecosystem

With over **100,000 crates** on **crates.io**, the Rust ecosystem covers everything from embedded firmware to web servers. The tooling around the language is equally mature.

rustfmt

Opinionated, automatic code formatting. Enforces a single consistent style across every Rust project — no style debates needed.

clippy

A rich linter with hundreds of checks. Catches common mistakes, non-idiomatic patterns, and performance pitfalls beyond what the compiler flags.

rust-analyzer

LSP server providing completions, inline type hints, go-to-definition, and refactoring across VS Code, Neovim, Emacs, and more.

cargo-edit / cargo-watch

`cargo-edit` adds `cargo add` for managing dependencies. `cargo-watch` re-runs checks automatically on every file save.

DEMO

Your First Cargo Project

Cargo scaffolds a fully working project in seconds. The generated structure follows community conventions so every Rust developer can navigate your codebase immediately.

```
# Create a new binary project
cargo new hello_rust
cd hello_rust

# Resulting structure:
# .
# ├── Cargo.toml   ← package manifest
# └── src
#     └── main.rs  ← entry point

cargo build         # compile (debug mode)
cargo run           # compile and execute
cargo check         # type-check only (fastest feedback)
cargo build --release  # optimised production build
```

`cargo check` skips code generation and is dramatically faster than a full build. Use it as your primary compile-check loop during development.

Understanding Cargo.toml

The `Cargo.toml` manifest is the single source of truth for your project's identity, dependencies, and build configuration. It uses the TOML format — human-readable and easy to diff.

```
[package]
name    = "hello_rust"
version = "0.1.0"
edition = "2021"          # Rust edition: 2015 | 2018 | 2021
authors = ["Your Name <email@example.com>"]

[dependencies]
serde = { version = "1.0", features = ["derive"] }
rand  = "0.8"

[dev-dependencies]
pretty_assertions = "1.0"  # used only in tests

[profile.release]
opt-level = "z"  # optimise for binary size
lto       = true # link-time optimisation
```

- **Editions** let the language evolve without breaking old code — your crate picks its edition independently.
- **Semantic versioning** is enforced by Cargo; `"1.0"` means any compatible 1.x release.
- **Profiles** give precise control over optimisation, debug info, and overflow behaviour per build type.

src/main.rs — Your First Look at Rust Code

Even a minimal Rust file illustrates several key language properties: immutability by default, explicit opt-in to mutation, and the distinction between macros (`println!`) and functions.

```
/// Documentation comment — supports Markdown, appears in `cargo doc`
fn main() {
    println!("Hello, world!"); // println! is a macro, not a function

    // Variables are IMMUTABLE by default
    let x = 5;
    // x = 6; ← COMPILE ERROR: cannot assign twice to immutable variable

    // Opt in to mutation explicitly
    let mut y = 5;
    y = 6; // OK

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

let

Binds a name to a value. Immutable unless `mut` is present.

println!

A macro that formats and prints. The `!` signals macro invocation.

Type Inference

The compiler infers `x: i32` automatically — no annotation needed.

☕ Break Time

10 minutes — stretch, hydrate, and ask any questions so far.

When we return: **Core Syntax & the Memory Model** — where Rust begins to feel genuinely different from C and C++.

MODULE 1 · PART B

Variables, Mutability & Shadowing

Rust makes immutability the default — a deliberate inversion of C/C++ convention. This forces engineers to be explicit about which state is intended to change, reducing a significant class of accidental mutation bugs.

Immutable by Default

```
let x = 5;
// x = 6; ← compile error
```

Variables declared with `let` are immutable. Any attempt to reassign them is a compile-time error.

Explicit Mutability

```
let mut y = 5;
y = 6; // OK
```

Adding `mut` opts into mutability. The `mut` keyword is visible at every use site, making mutable state explicit in the type signature.

Shadowing

```
let z = 5;
let z = z + 1; // new binding
let z = z * 2; // 12
```

Shadowing rebinds a name to a new value — even a new type — without `mut`. Useful for transforming values through a pipeline of steps.

SECTION 1.6

Variables and Mutability — Immutable by Default

This is one of Rust's most deliberate design decisions. Making immutability the default is not an inconvenience — it is a correctness aid. The compiler rejects accidental mutations before your code ever runs.

```
fn main() {
    // IMMUTABLE — the default
    let x = 5;
    // x = 6; ← COMPILE ERROR

    // MUTABLE — explicit opt-in
    let mut y = 5;
    y = 6; // fine

    // CONSTANTS — always immutable, type annotation required
    const MAX_POINTS: u32          = 100_000;
    const THREE_HOURS_IN_SECS: u32 = 3 * 60 * 60;
}
```

Prevents Accidental Changes

A variable that should never change cannot be changed — guaranteed.

Aids Reasoning

Immutable values are easier to track across complex call graphs.

Enables Optimisations

The compiler can cache, inline, and reorder immutable values freely.

Shadowing — Not the Same as Mutability

Shadowing lets you re-use a name in the same scope by introducing a **brand-new variable**. Crucially, the type can change — something impossible with `mut`. The original binding is simply hidden, not overwritten.

```
fn main() {
    let x = 5;
    let x = x + 1;        // shadows the first x

    {
        let x = x * 2;
        println!("Inner x: {}", x); // 12
    }
    println!("Outer x: {}", x);     // 6 — inner shadow gone

    // Type can change via shadowing — impossible with mut
    let spaces = "   ";    // &str
    let spaces = spaces.len(); // usize — totally different type!
}
```

Shadowing

- Creates a new binding
- Can change the type
- Original value is unchanged

Mutability (mut)

- Modifies the same binding
- Type must stay the same
- Value is overwritten in place

MODULE 1 · PART B

Scalar & Compound Data Types

Scalar Types

- **Integers:** `i8`, `i16`, `i32`, `i64`, `i128`, `isize` (signed); `u8`–`usize` (unsigned). Overflow is a compile-time error in debug builds.
- **Floats:** `f32` and `f64`. IEEE 754.
- **Boolean:** `bool` — strictly `true` or `false`. No integer coercion.
- **Character:** `char` — a Unicode scalar value (4 bytes), not a C `char`.

Compound Types

- **Tuples:** Fixed-length, heterogeneous. `let t: (i32, f64, bool) = (1, 2.0, true);` — accessed by index `t.0`.
- **Arrays:** Fixed-length, homogeneous. `let a: [i32; 5] = [1, 2, 3, 4, 5];` — stack-allocated. Bounds checks enforced at runtime in safe code.

Unlike C, Rust has no implicit type coercions between numerics. All conversions must be explicit using the `as` operator or `From`/`Into` traits.

Scalar Data Types

Rust's scalar types map directly to hardware primitives, giving you full control over size, signedness, and precision — a critical requirement for systems programming.

Integers

`i8` / `u8` · `i16` / `u16` · `i32` / `u32` · `i64` / `u64` · `i128` / `u128`

Architecture-sized: `isize` / `usize`. Default inferred integer is `i32`.

Floating Point

`f32` — 32-bit IEEE 754
`f64` — 64-bit IEEE 754 (default)

Both support the full set of arithmetic operators. Use `f64` unless code size or speed on embedded targets demands `f32`.

Boolean & Character

`bool`: `true` or `false` — one byte in memory.

`char`: a **Unicode scalar value**, always 4 bytes. Not a C `char` — it represents any Unicode codepoint, not just ASCII.

Compound Data Types — Tuples and Arrays

Both tuples and arrays are **stack-allocated** and **fixed in length** at compile time, making them efficient and predictable. They differ in whether the element types must be homogeneous.

Tuples — Heterogeneous, Fixed

```
let tup: (i32, f64, char) = (500, 6.4, 'z');

// Destructuring
let (x, y, z) = tup;

// Index access
let first = tup.0; // 500

// Unit tuple — zero-size, used as "void"
let unit = ();
```

Arrays — Homogeneous, Fixed

```
let arr = [1, 2, 3, 4, 5];
let first = arr[0]; // 1

// Explicit type + length
let typed: [i32; 5] = [1, 2, 3, 4, 5];

// Initialise all elements to same value
let zeros = [0; 10]; // ten zeroes
```

For dynamically-sized collections, use `Vec<T>` (heap-allocated). Arrays and tuples are ideal for fixed-size data where lifetime and size are known at compile time.

MODULE 1 · PART B

Functions, Statements & Expressions

One of Rust's most distinctive syntactic features — inherited from ML-family languages — is the distinction between **statements** (which do not return a value) and **expressions** (which do). Understanding this distinction is foundational for reading and writing idiomatic Rust.

Statements

Declarations and assignments are statements. `let x = 5;` does not produce a value — you cannot write `let y = (let x = 5);`. This eliminates a class of C bugs like `if (x = 5)`.

Expressions

Blocks, `if` constructs, and `match` arms are all expressions. A block's value is its final expression (no trailing semicolon). Functions implicitly return their final expression, making explicit `return` idiomatic only for early exits.

```
fn add_one(x: i32) -> i32 {
 x + 1 // expression — no semicolon, this is the return value
}
```

Functions in Rust

Rust uses **snake_case** by convention for function names. All parameter types are mandatory — there is no type inference at function boundaries, keeping call sites unambiguous.

```
fn add(x: i32, y: i32) -> i32 {
    x + y          // expression — no semicolon → this IS the return value
}

fn early_return(x: i32) -> i32 {
    if x < 0 {
        return 0;  // statement — explicit return, with semicolon
    }
    x * 2          // expression — implicit return
}

fn main() {
    let result = add(5, 3);    // 8
    let y = {
        let x = 3;
        x + 1      // block expression — evaluates to 4
    };
    println!("result={}, y={}", result, y);
}
```

Statements

Perform an action; produce no value. Terminated with `;`. Example: `let x = 5;`

Expressions

Evaluate to a value; **no trailing semicolon**. Blocks, `if`, `match`, and function bodies are all expressions.

Statements vs. Expressions — Deep Dive

Understanding this distinction is essential for reading idiomatic Rust. Because nearly everything is an expression, Rust code is more compositional than C or C++.

```
fn main() {
    // Blocks are expressions
    let y = {
        let x = 3;
        x + 1       // no semicolon → block evaluates to 4
    }; // y = 4

    // if is an expression — both branches must have the same type
    let condition = true;
    let number = if condition { 5 } else { 6 };

    // match is an expression
    let desc = match number {
        5 => "five",
        6 => "six",
        _ => "other",   // _ is the catch-all arm
    };

    println!("number={}, desc={}", number, desc);
}
```

Adding a `;` to the last line of a block turns an expression into a statement, changing the block's return type to `()` — the unit type. This is a common source of type-mismatch errors for newcomers.

Memory Model Preview — The Stack

The stack is the default home for most Rust values. Its LIFO discipline means allocation and deallocation are simply a pointer increment or decrement — effectively free at runtime.

Characteristics

- LIFO — last in, first out
- Size must be known at compile time
- Extremely fast allocation (pointer move only)
- Automatically reclaimed when scope ends

What Lives Here

- All scalar types: `i32`, `f64`, `bool`, `char`
- Fixed-size arrays and tuples
- Function call frames (local variables + return address)
- Pointers/references to heap data

In Rust

Most values live on the stack by default. Unlike C++, there is no implicit copying to the heap. You opt into heap allocation explicitly with `Box`, `Vec`, `String`, etc.

Memory Model Preview — The Heap

The heap handles data whose size is unknown at compile time or that needs to outlive the function that created it. Heap allocation is more flexible but more expensive than the stack.

```
fn main() {
    // String lives on the heap; the stack holds a "fat pointer"
    // (ptr, len, capacity)
    let s = String::from("hello");
    //  Stack                Heap
    //  ┌──────────────────────┐    ┌──────────────────────────────────┐
    //  │ ptr ─────────────────┼──── │ h e l l o        │
    //  │ len: 5       │    └──────────────────────────────────┘
    //  │ capacity: 5  │
    //  └──────────────────────┘

    let v: Vec<i32> = vec![1, 2, 3, 4, 5];
    // Same pattern: stack fat-pointer → heap buffer
}
```

String

Growable UTF-8 text. Stack holds the fat pointer; heap holds the bytes.

Vec<T>

Growable array. Heap buffer, stack metadata (ptr / len / cap).

Box<T>

Single heap allocation. Used to store large values or enable recursive types.

Memory Management — C/C++ vs. Java vs. Rust

Rust occupies a unique position: it offers the performance of manual management without its dangers, and the safety of garbage collection without its overhead.

Rust's ownership system is the key innovation: the compiler **statically tracks** every allocation and automatically inserts the correct deallocation — no human error possible, no GC pause possible.

OWNERSHIP PREVIEW

Preview — The Three Rules of Ownership

These three rules are the entire foundation of Rust's memory safety guarantees. We will explore them in depth in Module 2. For now, understand the mental model they create.

1

Each value has exactly one owner

A value belongs to precisely one variable at any point in time. There is no shared ownership by default.

2

Only one owner at a time

Ownership can be transferred (moved), but not duplicated. Two variables cannot simultaneously own the same heap data.

3

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

The compiler inserts a call to `drop()` automatically. No GC needed, no `free()` needed, no leak possible.

```
let s1 = String::from("hello");
let s2 = s1; // s1 is MOVED — it is no longer valid

// println!("{}", s1); // COMPILE ERROR: value borrowed after move
println!("{}", s2); // OK — s2 is the owner
```

Common Pitfalls to Avoid

Even experienced C/C++ developers trip over these early Rust behaviours. Reading the compiler output carefully — Rust's error messages are among the best in the industry — will resolve almost all of them immediately.

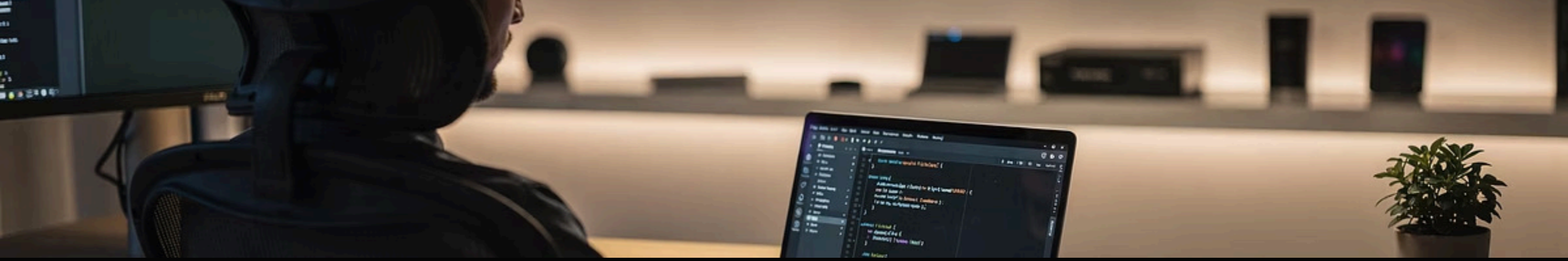
```
fn main() {
    // 1. Integer overflow: panic in debug, wrapping in release
    let mut x: u8 = 255;
    // x += 1; // debug → panic; release → wraps to 0

    // 2. Unused variables: prefix with _ to silence the warning
    let _unused = 5;

    // 3. Shadowing confusion — the type changed silently
    let x = 5;
    let x = x.to_string(); // x is now a String, not i32

    // 4. Array out-of-bounds: always a runtime panic (never UB)
    let arr = [1, 2, 3];
    // let bad = arr[5]; // thread 'main' panicked: index out of bounds
}
```

Unlike C/C++, an out-of-bounds array access in Rust is **never undefined behaviour**. It is always a deterministic panic that can be caught and handled — a significant safety improvement.



CHAPTER 2

Lab Session — Hands-On Practice

The next four exercises move from verifying your setup to observing Rust's ownership semantics in action. Work through them in order — each builds on the previous one.

01

Exercise 1

Setup Verification — confirm your toolchain and run your first project

02

Exercise 2

Variables and Types — experiment with scalars, tuples, arrays, and shadowing

03

Exercise 3

Functions — write functions using expression-style and early-return styles

04

Exercise 4

Memory Experiment — observe move semantics and `clone()` in practice

LAB — EXERCISE 1

Lab Exercise 1 — Setup Verification

Before writing real code, confirm every component of your toolchain is installed correctly and that Cargo can scaffold, build, and run a project cleanly.

```
# 1. Verify your toolchain versions
rustc --version
cargo --version

# 2. Create and enter a new project
cargo new lab_exercise
cd lab_exercise

# 3. Build and run the generated code
cargo run
# Expected output: Hello, world!

# 4. Modify src/main.rs — change the message to something personal
# 5. Run again and confirm your change appears
```

Explore

Open `Cargo.toml` and identify the `[package]` and `[dependencies]` sections.

Observe

Notice the `target/` directory created by the first build — this is where all compiled artefacts live.

LAB — EXERCISE 2

Lab Exercise 2 — Variables and Types

Experiment with Rust's type system hands-on. The goal is to internalise the difference between immutable bindings, mutable bindings, constants, and shadowing through direct observation.

```
fn main() {
    // TODO: Declare at least one of each:
    //   i32, u64, f64, bool, char

    // TODO: Declare a tuple representing a person
    // e.g. (name_as_char_initial, age, height_in_meters)

    // TODO: Declare an array of 5 temperatures (f32)

    // TODO: Use shadowing to transform spaces (&str) into its length (usize)

    // TODO: Try assigning to an immutable variable — read the error!

    println!("All variables created successfully!");
}
```

Challenge A

Create a tuple `(char, u32, f64)` for a person and destructure it into named variables.

Challenge B

Initialise an array of five temperatures using the `[value; count]` syntax, then change one with a `mut` binding.

LAB — EXERCISE 3

Lab Exercise 3 — Functions

Practice writing functions in idiomatic Rust. Focus on the difference between expression-style returns (no semicolon) and explicit `return` statements, and on using `if` as an expression.

```
// 1. Expression-style return: takes two i32, returns their sum
fn add(x: i32, y: i32) -> i32 { /* TODO */ }

// 2. Early return: takes f64, returns 0.0 if negative, else square
fn safe_square(n: f64) -> f64 { /* TODO */ }

// 3. if expression: takes bool, returns &"yes" or &"no"
fn yes_or_no(b: bool) -> &'static str { /* TODO */ }

fn main() {
    println!("{}", add(3, 4));          // 7
    println!("{}", safe_square(-2.0));  // 0.0
    println!("{}", yes_or_no(true));    // yes
}
```

Challenge A

Write `celsius_to_fahrenheit(c: f64) -> f64` using the formula `c * 9.0/5.0 + 32.0`.

Challenge B

Write `largest(a: i32, b: i32, c: i32) -> i32` returning the greatest value using nested `if` expressions.

LAB — EXERCISE 4

Lab Exercise 4 — Memory Experiment

This exercise makes move semantics *tangible*. Observe the difference between **Copy types** (cheap stack duplication) and **Move types** (heap ownership transfer), and learn when `clone()` is the right tool.

```
fn main() {
    // Copy types — duplicated on the stack, both vars remain valid
    let x = 42;
    let y = x;
    println!("Stack Copy: x={}, y={}", x, y); // both valid

    // Move types — heap ownership transferred, original invalidated
    let s1 = String::from("hello");
    let s2 = s1; // s1 is MOVED
    // println!("{}", s1); // ← uncomment to see the compile error
    println!("After move: s2={}", s2);

    // Explicit deep copy with .clone()
    let s3 = String::from("world");
    let s4 = s3.clone(); // heap data duplicated
    println!("After clone: s3={}, s4={}", s3, s4); // both valid
}
```

The key rule: types that implement the `Copy` trait (all scalars, tuples of Copy types, fixed arrays of Copy types) are duplicated silently. Everything else is moved.

Troubleshooting Common Compiler Errors

Rust's compiler errors are famously detailed. Rather than a cryptic one-liner, `rustc` shows you exactly where the problem is, what rule was violated, and often how to fix it. Learning to read these messages is a skill that pays dividends for your entire Rust career.

```
// Error 1: Mismatched types
let x: i32 = "hello"; // expected `i32`, found `&str`

// Error 2: Assign to immutable variable
let x = 5;
x = 6; // cannot assign twice to immutable variable `x`
       // help: consider making `x` mutable: `let mut x`

// Error 3: Use of moved value
let s = String::from("test");
let t = s;
println!("{}", s); // error: borrow of moved value: `s`

// Error 4: Index out of bounds (runtime panic — not a compile error)
let arr = [1, 2, 3];
let _x = arr[5]; // panics at runtime with a clear message
```

> "The Rust compiler is the best pair-programming partner you'll ever have. Read every error message from top to bottom — the fix is almost always right there."

Key Takeaways — Module 1

Here is everything you should be able to articulate after completing this module. These concepts are the foundation every subsequent module builds upon.

🛡️ Safety, Speed, Concurrency

Rust's three pillars — all enforced or enabled at the **language level**, not via runtime libraries.

🐛 C/C++ Problems Solved

Memory errors and data races — the source of ~70% of security CVEs — are eliminated at **compile time**.

🔧 The Toolchain

`rustup` manages versions, `rustc` compiles, `cargo` builds, tests, and manages packages.

📝 Variables & Functions

Immutable by default, explicit `mut`. Shadowing changes type. Functions use expressions for clean returns.

💾 Stack vs. Heap

Scalars and fixed-size types live on the stack. `String`, `Vec`, `Box` allocate on the heap. Fat pointers bridge both.

🔑 Ownership Preview

One owner, one scope. Moving transfers ownership. `clone()` makes deep copies. The compiler enforces all of this — no GC required.

Preview of Module 2 — Flow Control & Ownership

Module 2 is where Rust begins to feel genuinely different. We will move beyond syntax and into the borrow checker — the engine that makes Rust's guarantees possible.

1

Flow Control

`if`, `loop`, `while`, `for` — and Rust's uniquely powerful `match` with full pattern matching

2

Ownership in Depth

The three rules in full detail. Move semantics, the `Copy` trait, and when to `clone()`

3

References & Borrowing

Shared `&T` vs. exclusive `&mut T`. The borrow checker rules. Slices as borrowed views into contiguous data

4

Compiler Internals

A tour of MIR (Mid-level Intermediate Representation) — understanding what the borrow checker actually checks

📚 **Homework:** Install Rust, run through all four lab exercises, and experiment freely. The more you poke at the compiler's error messages, the faster Module 2 will click.

Q&A Session

No question is too basic — every experienced Rust programmer was a beginner once. If something felt unclear, there is almost certainly someone else in the room wondering the same thing.

Installation Issues

Trouble with `rustup`, PATH configuration, or running on Windows? Let's solve it together right now.

Syntax Questions

Puzzled by a semicolon, a borrow, or an unexpected compiler error? Bring it up — live debugging is the best way to learn.

Memory Model

Not sure whether something lives on the stack or the heap, or why the move happened? Stack/heap intuition takes time — ask now.

Anything Else

Comparison with C++? Performance questions? Real-world adoption stories? All fair game.

Additional Resources

These are the canonical, high-quality references used by the entire Rust community — from beginners to compiler contributors. Bookmark them all.

📚 Official Documentation

- **The Rust Book** — the definitive learning resource, free online
- **Rust by Example** — short, runnable code snippets for every concept
- **Standard Library Docs** — exhaustive API reference

🛠️ Tools & Playground

- **Rust Playground** — write and share Rust code in your browser, no install needed
- **Awesome Rust** — curated list of crates and learning materials

💬 Community

- **users.rust-lang.org** — the official help forum, friendly and active
- **Official Discord** — real-time chat with the community
- **This Week in Rust** — weekly newsletter covering new crates, blog posts, and RFCs

The Cargo Workflow — Day-to-Day Commands

Once you internalise these commands, your entire development loop runs through Cargo. There is no separate configure, make, and link step — Cargo orchestrates everything.

Develop

- `cargo new <name>` — scaffold project
- `cargo check` — type-check, no codegen (fastest)
- `cargo build` — debug build
- `cargo run` — build and execute

Test & Lint

- `cargo test` — run all unit and integration tests
- `cargo clippy` — run the linter
- `cargo fmt` — format the source tree
- `cargo bench` — run benchmarks (nightly)

Ship

- `cargo build --release` — optimised build
- `cargo doc --open` — generate and view docs
- `cargo publish` — publish crate to crates.io
- `cargo audit` — scan dependencies for CVEs

Control Flow — A Quick Preview

Rust's control flow constructs will be covered in depth in Module 2. Here is the shape of each one so that the lab code feels familiar rather than alien.

```
// if — also an expression
let x = if condition { 5 } else { 6 };

// loop — loops forever; use `break value` to return a value
let result = loop {
    counter += 1;
    if counter == 10 { break counter * 2; }
};

// while
while number != 0 { number -= 1; }

// for — preferred over index loops; safe, concise
for element in [10, 20, 30] {
    println!("{}", element);
}

// for with range
for i in 0..5 { println!("{}", i); } // 0,1,2,3,4
```

Rust's `for` loop works with anything that implements the `Iterator` trait — arrays, ranges, slices, collections, and custom types all work identically. There are no raw pointer-based for loops.

Pattern Matching with match

`match` is one of Rust's most powerful features. It is an expression, it is exhaustive (every possible case must be handled), and it supports rich patterns far beyond simple equality checks.

```
let number = 7;

let description = match number {
    1         => "one",
    2 | 3     => "two or three",    // OR pattern
    4..=6     => "four to six",     // inclusive range
    n if n % 2 == 0 => "even",      // guard condition
    _         => "something else",  // catch-all (required!)
};

// match on tuples
let point = (0, -2);
match point {
    (0, 0)  => println!("origin"),
    (x, 0)  => println!("on x-axis at {}", x),
    (0, y)  => println!("on y-axis at {}", y),
    (x, y)  => println!("at ({}, {})", x, y),
}
```

The compiler ensures all branches are covered — forgetting a case is a **compile error**, not a silent fallthrough. This property, known as *exhaustiveness checking*, eliminates an entire class of logic bugs common in `switch` statements in C/C++.

Structs and Enums — A Taste

Structs and enums are the primary tools for creating custom data types in Rust. Module 3 covers them comprehensively, but understanding their shape helps make sense of code you will encounter in the labs.

Structs — Named Fields

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

let p = Point { x: 1.0, y: 2.5 };
println!("{}", p.x);
```

Like C structs but with methods, trait implementations, and privacy control via `pub`.

Enums — Tagged Unions

```
enum Direction {
    North,
    South,
    East,
    West,
}

let d = Direction::North;
match d {
    Direction::North => println!("go north"),
    _ => println!("other"),
}
```

Rust enums can carry data in each variant — far more powerful than C enums.

Option<T> — Rust's Null Replacement

Rust has no `null` keyword. Instead, the standard library provides `Option<T>` — an enum that makes the possibility of absence **explicit in the type system**. You cannot accidentally use a None as a value; the compiler forces you to handle both cases.

```
fn divide(a: f64, b: f64) -> Option<f64> {
    if b == 0.0 { None } else { Some(a / b) }
}

fn main() {
    let result = divide(10.0, 2.0);

    // Must handle both variants — no null pointer exceptions possible
    match result {
        Some(v) => println!("Result: {}", v),
        None    => println!("Cannot divide by zero"),
    }

    // Shorthand with if let
    if let Some(v) = divide(6.0, 3.0) {
        println!("Got: {}", v);
    }
}
```

`Option<T>` is so fundamental that its variants `Some` and `None` are in the Rust prelude — you never need to import them explicitly.

Rust's Error Handling Philosophy

Rust makes errors **part of the type system**. Functions that can fail return `Result<T, E>`. The caller is forced to handle both the success and failure paths — there are no unchecked exceptions and no implicit error propagation.

```
use std::num::ParseIntError;

fn parse_and_double(s: &str) -> Result<i32, ParseIntError> {
    let n = s.trim().parse::<i32>()?; // ? propagates error upward
    Ok(n * 2)
}

fn main() {
    match parse_and_double("21") {
        Ok(v)  => println!("Success: {}", v), // 42
        Err(e) => println!("Error: {}", e),
    }
    match parse_and_double("abc") {
        Ok(v)  => println!("Success: {}", v),
        Err(e) => println!("Error: {}", e),   // "invalid digit"
    }
}
```

Result<T, E>

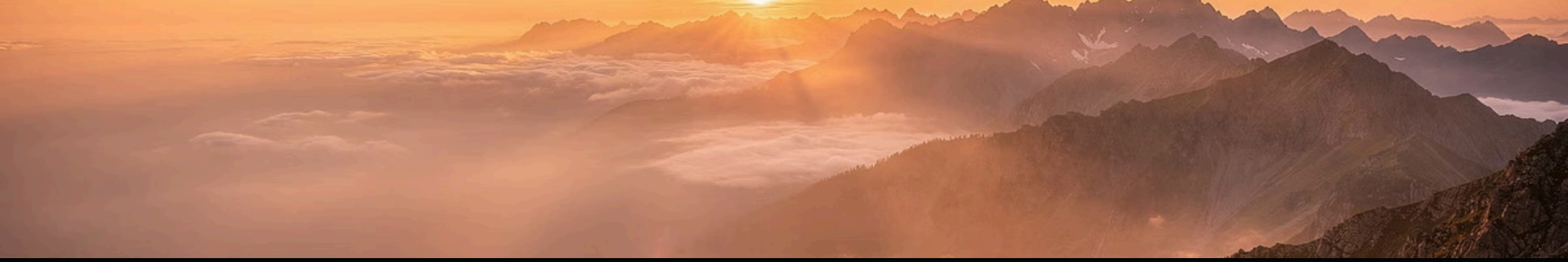
The standard return type for fallible operations. Forces error handling at every call site.

The ? Operator

Propagates an `Err` upward automatically — ergonomic without hiding errors.

panic!

For truly unrecoverable situations. Unwinds the stack cleanly rather than producing undefined behaviour.



Thank You — See You in Module 2!

You have covered the full breadth of Module 1: Rust's design philosophy, its relationship to C and C++, the toolchain, core syntax, and a first look at the ownership model that makes everything possible.

Next Session

Module 2: Flow Control & Ownership
Where Rust truly becomes Rust — the borrow checker, references, and fearless concurrency in action.

Homework

Complete all four lab exercises. Read Chapters 1–4 of *The Rust Book*. Try breaking things — the compiler's error messages are your best teacher.

Questions?

Reach out any time:
training@chandrashekar.info

> "The obstacle is the way. Every compiler error is Rust teaching you to write better systems software."