Rust Programming: Systems Reimagined

Module 5: Module System, Packaging & The Rust Compiler Pipeline

Part 1: Project Architecture with Modules & Crates · Part 2: Advanced Cargo & Publishing · Part 3: Compiler Internals · Part 4: Lab Session

Chandrashekar Babu · **training@chandrashekar.info**

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

Module 5 Overview

This module takes you from organising Rust source code into packages and modules, all the way through to how the compiler transforms that code into a safe, optimised binary. Four parts, progressively deeper.

1

5.1 Project Architecture

Packages, Crates, Modules, Paths · Visibility with `pub` · Splitting into files · External crates

2

5.2 Advanced Cargo

Workspaces · Build profiles · Publishing to crates.io · Semantic versioning

3

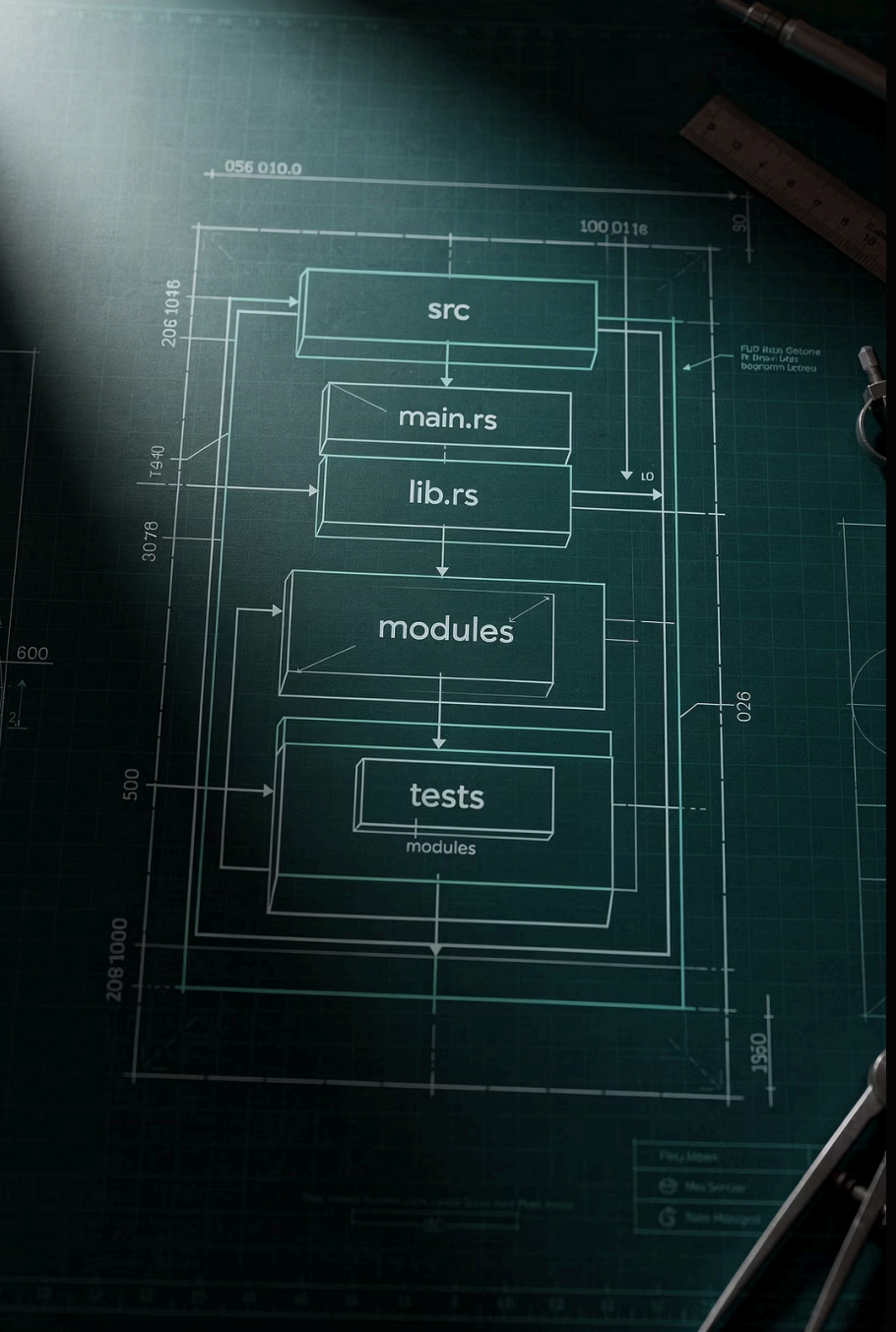
5.3 Compiler Internals

Full pipeline: Source → AST → HIR → MIR → LLVM → Binary · Safety enforcement · Zero-cost abstractions

4

5.4 Lab Session

Refactor a monolith into a workspace · Explore compiler output · Profile compilation

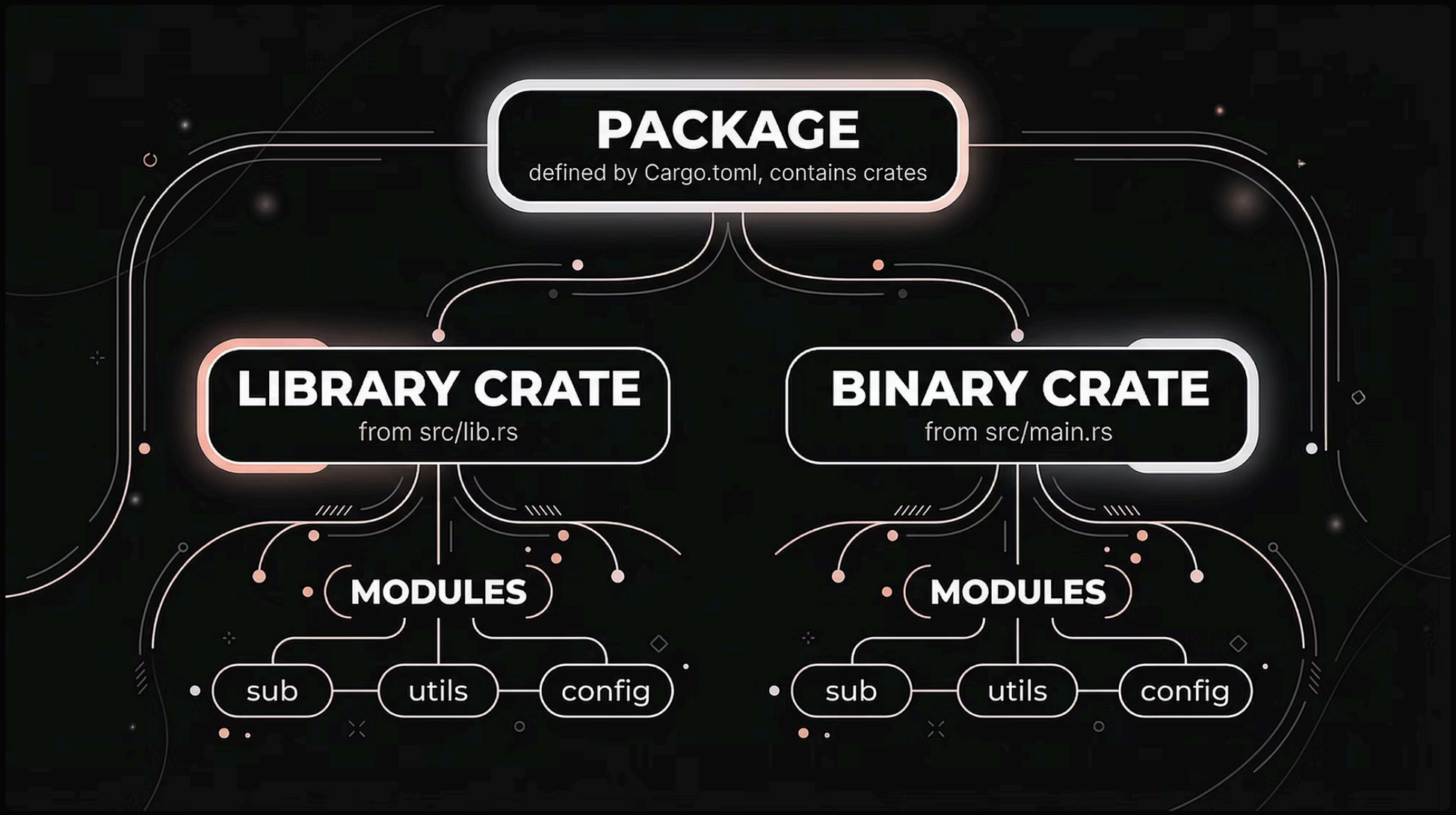


PART 5.1

Project Architecture with Modules & Crates

The Big Picture: Packages, Crates & Modules

Rust organises code through three nested concepts. Understanding how they relate is fundamental to structuring any non-trivial project.



Crate

The smallest compilation unit. Produces a binary or a reusable library.

Package

A collection of crates with shared metadata in `Cargo.toml`. At most one library crate.

Module

Namespaced organisation within a crate. Controls visibility and path resolution.

Creating a New Package

Cargo handles all scaffolding. Use `cargo new` for binary packages and add `--lib` for library crates. The generated `Cargo.toml` is the single source of truth for metadata and dependencies.

Binary Package

```
$ cargo new my_app
my_app/
 ├── Cargo.toml
 └── src/
     └── main.rs
```

Library Package

```
$ cargo new my_lib --lib
my_lib/
 ├── Cargo.toml
 └── src/
     └── lib.rs
```

Cargo.toml

```
[package]
name = "my_app"
version = "0.1.0"
edition = "2021"
authors = ["Your Name "]
description = "A sample Rust application"
license = "MIT OR Apache-2.0"

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

Module System Basics

Modules group related items and control visibility. Items are **private by default**; mark them `pub` to expose them. Nested modules and the `use` keyword keep call sites concise.

Defining a Module

```
mod math {
    fn add(a: i32, b: i32) -> i32 { a + b }

    pub fn multiply(a: i32, b: i32) -> i32 { a * b }

    pub mod advanced {
        pub fn square(x: i32) -> i32 { x * x }
    }
}
```

Consuming a Module

```
fn main() {
    // math::add(5, 3);       // ERROR – private
    let p = math::multiply(5, 3);
    let s = math::advanced::square(5);

    // Bring into scope
    use math::multiply;
    let r = multiply(5, 3);
}
```

Items inside a module are private by default. Use `pub` to expose functions, structs, and nested modules to callers outside the module boundary.

Module Trees and Paths

Rust supports both **absolute paths** (starting with `crate::`) and **relative paths** using the current identifier, `self`, or `super`. Use `pub use` to re-export items and present a cleaner public API.

Module Tree

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
        fn seat_at_table() {}      // private
    }
    mod serving {
        fn take_order() {}
        pub fn serve_order() {}
    }
}
```

Path Styles

```
// Absolute
crate::front_of_house::hosting::add_to_waitlist();
// Relative
front_of_house::hosting::add_to_waitlist();
// Parent (super)
super::front_of_house::hosting::add_to_waitlist();
```

Re-exporting with pub use

```
mod back_of_house {
 pub struct Breakfast {
  pub toast: String,
  seasonal_fruit: String, // private
 }
 impl Breakfast {
  pub fn summer(toast: &str) -> Breakfast {
   Breakfast {
    toast: String::from(toast),
    seasonal_fruit: String::from("peaches"),
   }
  }
 }
}

// Cleaner public API
pub use crate::back_of_house::Breakfast;
```

Visibility Rules: The `pub` Modifier

Rust's visibility system is fine-grained. Beyond simple `pub`, you can restrict visibility to the crate, a parent module, or any specific path — giving you precise control over your API surface.

`pub`

Fully public — accessible from anywhere that can name the item.

`pub(crate)`

Visible anywhere within the current crate, but not to external users.

`pub(super)`

Visible to the parent module only — useful for implementation helpers.

`pub(in path)`

Visible within a specific ancestor module path. Highly targeted.

(no modifier)

Private — only the defining module and its descendants can access it.

Enum variants always inherit the visibility of the enum itself. Struct fields must be individually annotated with `pub` — they are private by default even on a `pub struct`.

Splitting Code into Multiple Files

As projects grow, keeping everything in one file becomes impractical. Rust maps module declarations to the filesystem automatically — declare a module with `mod name;` and the compiler looks for `name.rs` or `name/mod.rs`.

Project Layout

```
my_app/
 ├── Cargo.toml
 └── src/
    ├── main.rs
    ├── math.rs       ← module file
    └── math/
       └── algebra.rs ← sub-module
```

src/math.rs

```
pub fn add(a: i32, b: i32) -> i32 { a + b }

pub mod algebra; // compiler finds math/algebra.rs
```

src/math/algebra.rs

```
pub fn multiply(a: i32, b: i32) -> i32 { a * b }
```

src/main.rs

```
mod math; // declare the module

fn main() {
 println!("{}", math::add(5, 3));
 println!("{}", math::algebra::multiply(5, 3));
}
```

Old Style vs New Style Module Files

Before Rust 2018, sub-modules required a `mod.rs` file. The new style uses a file named after the module itself, reducing clutter and improving IDE navigation. Prefer the new style for all Rust 2018+ projects.

Old Style (mod.rs)

```
src/
 ├── main.rs
 └── math/
     ├── mod.rs      ← declarations here
     ├── algebra.rs
     └── geometry.rs

// src/math/mod.rs
pub mod algebra;
pub mod geometry;
```

New Style (Rust 2018+) ✓ Preferred

```
src/
 ├── main.rs
 ├── math.rs         ← declarations here
 └── math/
     ├── algebra.rs
     └── geometry.rs

// src/math.rs
pub mod algebra;
pub mod geometry;
```

→ One fewer file — no `mod.rs` needed at each level

→ Clearer relationship between file name and module name

→ Better IDE and editor support for navigation and refactoring

Multi-File Module Organisation

Here is a complete worked example showing how the compiler resolves a multi-level module tree from independent source files. Each `mod name;` line is a declaration — the compiler does the rest.

src/main.rs

```
mod math;
mod utils;
use math::algebra;
use utils::strings;

fn main() {
    println!("{}", math::add(5, 3));
    println!("{}", algebra::multiply(5, 3));
    println!("{}", strings::reverse("Hello"));
}
```

src/math.rs

```
pub fn add(a: i32, b: i32) -> i32 { a + b }
pub mod algebra; // → math/algebra.rs
```

src/math/algebra.rs

```
pub fn multiply(a: i32, b: i32) -> i32 { a * b }
```

src/utils.rs

```
pub mod strings; // → utils/strings.rs
```

src/utils/strings.rs

```
pub fn reverse(s: &str) -> String {
 s.chars().rev().collect()
}
```

Using External Crates

Declare dependencies in `Cargo.toml` and Cargo downloads, compiles, and links them automatically. Crates can come from crates.io, a Git repository, or a local path — and you can restrict them to specific platforms.

Cargo.toml

```
[dependencies]
serde  = { version = "1.0", features = ["derive"] }
rand   = "0.8.5"
regex  = "1.5"

# From Git
tokio = { git = "https://github.com/tokio-rs/tokio",
          branch = "master" }
# Local path
my_local_lib = { path = "../my_local_lib" }

[target.'cfg(target_os = "linux")'.dependencies]
inotify = "0.9"

[target.'cfg(windows)'.dependencies]
winapi = "0.3"
```

src/main.rs

```
use serde::{Serialize, Deserialize};
use rand::Rng;
use regex::Regex;

#[derive(Serialize, Deserialize, Debug)]
struct User { name: String, age: u8 }

fn main() {
 let n: u8 = rand::thread_rng().gen();
 println!("Random: {}", n);

 let re = Regex::new(r"^\d{4}-\d{2}-\d{2}$").unwrap();
 println!("{}", re.is_match("2023-12-25"));

 let user = User { name: "Alice".to_string(), age: 30 };
 println!("{}", serde_json::to_string(&user).unwrap());
}
```

Managing Dependencies with Cargo

Cargo provides a rich set of subcommands for keeping dependencies up to date, auditing for vulnerabilities, and visualising the full dependency graph — an invaluable toolkit for maintaining healthy projects.

Day-to-Day Commands

```
$ cargo add rand
$ cargo add serde --features derive
$ cargo rm rand
$ cargo update          # update all (semver)
$ cargo update -p rand   # update one crate
```

Inspection

```
$ cargo tree           # full dep tree
$ cargo outdated       # show newer versions
$ cargo audit          # security scan
```

cargo tree Output

```
my_app v0.1.0
├── rand v0.8.5
│   ├── libc v0.2.126
│   ├── rand_chacha v0.3.1
│   │   ├── ppv-lite86 v0.2.16
│   │   └── rand_core v0.6.3
│   └── rand_core v0.6.3
└── serde v1.0.144
```

Features: Conditional Compilation

Cargo features let a crate expose optional capabilities. Consumers opt in at compile time, keeping binaries lean. Features can enable optional dependencies, toggle `no_std` support, or expose entire API layers.

Cargo.toml

```
[features]
default = ["std"]
std     = []
serde   = ["dep:serde"]   # optional dep
full    = ["std", "serde"]

[dependencies]
serde = { version = "1.0", optional = true }
```

Building with Features

```
$ cargo build --features "serde"
$ cargo build --no-default-features
$ cargo build --features "full"
```

src/lib.rs

```
#![cfg_attr(not(feature = "std"), no_std)]

#[cfg(feature = "serde")]
use serde::{Serialize, Deserialize};

#[cfg_attr(feature = "serde",
 derive(Serialize, Deserialize))]
pub struct Config {
 pub timeout: u32,
 pub retries: u8,
}

#[cfg(feature = "std")]
impl Config {
 pub fn from_file(path: &str)
 -> Result { todo!() }
}

#[cfg(not(feature = "std"))]
impl Config {
 pub fn default() -> Self {
 Config { timeout: 30, retries: 3 }
 }
}
```

PART 5.2

Advanced Cargo & Publishing

Cargo Workspaces

A workspace groups multiple related packages under a single `Cargo.lock` and a shared `target/` directory. This means consistent dependency versions across all member crates and significantly faster incremental builds.

Workspace Layout

```
my_workspace/
├── Cargo.toml      ← workspace definition
├── Cargo.lock      ← single lockfile
├── target/         ← shared build cache
├── crates/
│   ├── core/
│   │   ├── Cargo.toml
│   │   └── src/lib.rs
│   ├── utils/
│   │   ├── Cargo.toml
│   │   └── src/lib.rs
│   └── app/
│       ├── Cargo.toml
│       └── src/main.rs
└── examples/
    └── example.rs
```

Benefits at a Glance

- **Single lockfile** — all crates share consistent dependency versions
- **Shared target dir** — no duplicate compilation of common deps
- **Workspace deps** — declare once, reference everywhere
- **Unified commands** — `cargo test --workspace` covers all crates

Setting Up a Workspace

The root `Cargo.toml` lists all member crates and can declare shared dependencies that individual crates inherit with `workspace = true`. This eliminates version drift across your crate graph.

workspace/Cargo.toml

```
[workspace]
members = [
    "crates/core",
    "crates/utils",
    "crates/app",
]
resolver       = "2"
default-members = ["crates/app"]

[workspace.dependencies]
serde      = { version = "1.0", features = ["derive"] }
anyhow     = "1.0"
thiserror  = "1.0"
```

crates/core/Cargo.toml

```
[package]
name    = "core"
version = "0.1.0"
edition = "2021"

[dependencies]
serde     = { workspace = true }
thiserror = { workspace = true }
```

crates/app/Cargo.toml

```
[package]
name = "app"
version = "0.1.0"
edition = "2021"

[dependencies]
core = { path = "../core" }
utils = { path = "../utils" }
serde = { workspace = true }
anyhow = { workspace = true }
```

Working with Workspaces: Key Commands

Cargo commands accept `--workspace` to target all members, or `-p name` to target a single crate. Use these consistently in CI to catch integration issues early.

Build & Run

```
$ cargo build --workspace
$ cargo build -p core
$ cargo run -p app
```

Test & Check

```
$ cargo test --workspace
$ cargo check --workspace
$ cargo clippy --
workspace
```

Maintenance

```
$ cargo update
$ cargo clean
```

Publish (in dep order)

```
$ cargo publish -p core
$ cargo publish -p utils
$ cargo publish -p app
```

Build Profiles: Customising Compilation

Cargo provides two built-in profiles — `dev` for fast iteration and `release` for maximum performance. Every aspect of code generation is tunable, from optimisation level to panic strategy.

[profile.dev] — fast compile, easy debugging

```
opt-level       = 0     # No optimisations
debug           = true  # Debug info included
debug-assertions = true
overflow-checks  = true
lto             = false
codegen-units    = 256   # Parallel, faster compile
incremental      = true
panic           = 'unwind'
```

[profile.release] — maximum runtime performance

```
opt-level      = 3      # Max speed
debug          = false
lto            = true    # Link-time optimisation
codegen-units  = 1       # Slower compile, faster code
incremental    = false
panic          = 'abort' # Smaller binary
strip          = "symbols"
```

`panic = 'abort'` in release mode can meaningfully reduce binary size by removing unwinding machinery — important for embedded and WebAssembly targets.

Advanced Profile Configuration

Beyond the two built-in profiles, you can define custom profiles for specialised use cases — size-optimised builds for embedded targets, or per-crate overrides to tune a hot dependency without sacrificing debug ergonomics.

Custom Profiles

```
# Optimise for smallest binary
[profile.size]
inherits      = "release"
opt-level     = "z"
lto           = true
codegen-units = 1
panic         = "abort"

# Optimise for peak throughput
[profile.speed]
inherits      = "release"
opt-level     = 3
lto           = "fat"   # Full LTO
codegen-units = 1
```

Per-Package & Debug Overrides

```
# Override for one crate
[profile.release.package.my_crate]
opt-level = 2

# Dev build with light opts (profiling)
[profile.dev]
opt-level = 1
debug     = true
```

Usage & Comparison

```
$ cargo build --profile size
$ cargo build --profile speed
$ ls -lh target/release/my_app
$ ls -lh target/size/my_app
```

Publishing to **crates.io**

crates.io is Rust's official package registry. A well-prepared `Cargo.toml` with complete metadata ensures your crate is discoverable, documented, and trustworthy to potential users.

Required & Recommended Metadata

```
[package]
name         = "my_awesome_lib"
version      = "0.1.0"
edition      = "2021"
authors      = ["Your Name "]
description  = "A short, clear description"
license      = "MIT OR Apache-2.0"
repository   = "https://github.com/user/my_awesome_lib"
homepage     =
"https://github.com/user/my_awesome_lib#readme"
documentation = "https://docs.rs/my_awesome_lib"
readme       = "README.md"
keywords     = ["utility", "example", "learning"]
categories   = ["algorithms", "data-structures"]
exclude      = ["tests/*", "examples/*"]
include      = ["src/**/*", "README.md", "LICENSE"]
```

What Matters Most

- **License** — MIT or Apache-2.0 are the community standard; dual-licensing is common
- **Description** — shown in search results; keep it precise
- **Keywords** — maximum 5; drive discoverability
- **Categories** — maximum 5 from the official list at crates.io/category_slugs

Documentation for Publishing

Rust's toolchain generates HTML documentation directly from source comments. Inner doc comments (`//!`) document a module or crate; outer doc comments (`///`) document individual items. Embedded code examples are compiled and run as tests.

Crate-Level Docs (//!)

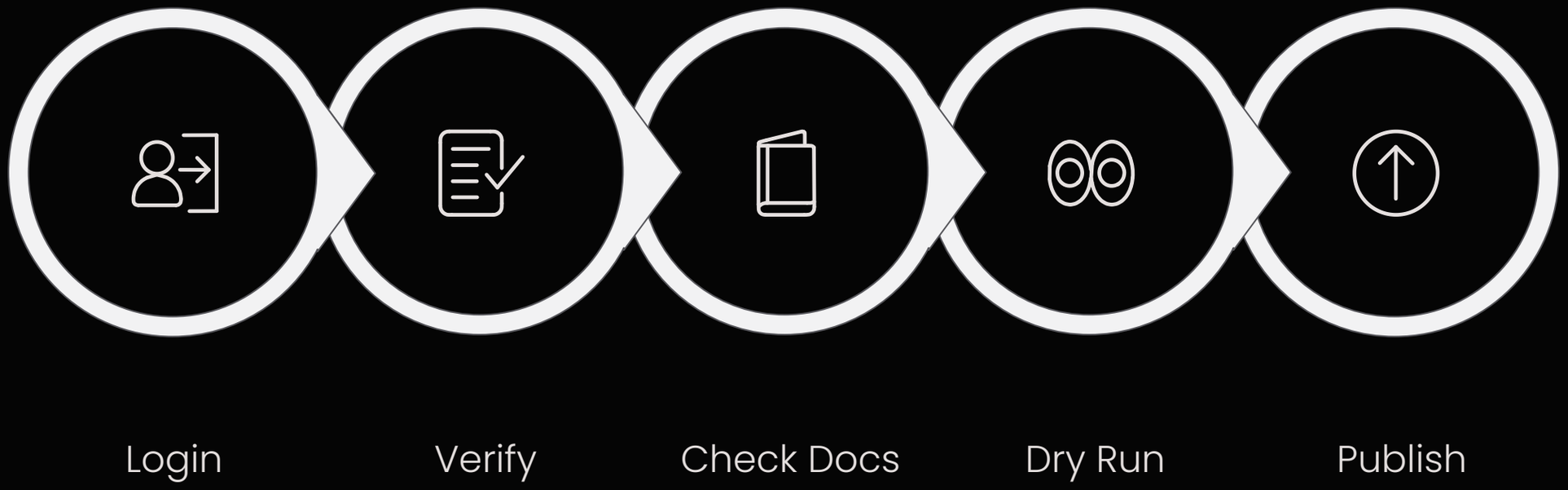
```
//! # My Awesome Library
//!
//! Utility functions that make common tasks easier.
//!
//! ## Example
//! ```
//! use my_awesome_lib::string_utils;
//! let rev = string_utils::reverse("hello");
//! assert_eq!(rev, "olleh");
//! ```
```

Item-Level Docs (///)

```
/// Reverses a string.
///
/// # Arguments
/// * `s` - The string to reverse
///
/// # Example
/// ```
/// use my_awesome_lib::reverse;
/// assert_eq!(reverse("hello"), "olleh");
/// ```
pub fn reverse(s: &str) -> String {
 s.chars().rev().collect()
}

/// # Errors
/// Returns an error if the string cannot be parsed.
pub fn parse_number(s: &str)
 -> Result {
 s.parse()
}
```

Publishing Workflow

Publishing is a one-way operation — once a version is live, it cannot be overwritten. Follow the steps below in order, and use `--dry-run` to catch metadata issues before they reach the registry.



Yanking a Release

```
$ cargo yank --vers 0.1.0
$ cargo yank --vers 0.1.0 --undo
```

After Publishing

```
# View on the registry
# https://crates.io/crates/my_awesome_lib
# Docs auto-generated at
# https://docs.rs/my_awesome_lib
```

Pre-Publishing Checklist

Rushing a publish is a common source of regret. Work through this checklist before every release to ensure your crate meets community standards and is ready for production use by others.

Metadata Complete

- name, version, authors, description
- license, edition, repository

Documentation

- README.md with examples
- Module docs (`//!`) and function docs (`///`)
- Doc tests passing (`cargo test --doc`)

Semantic Versioning

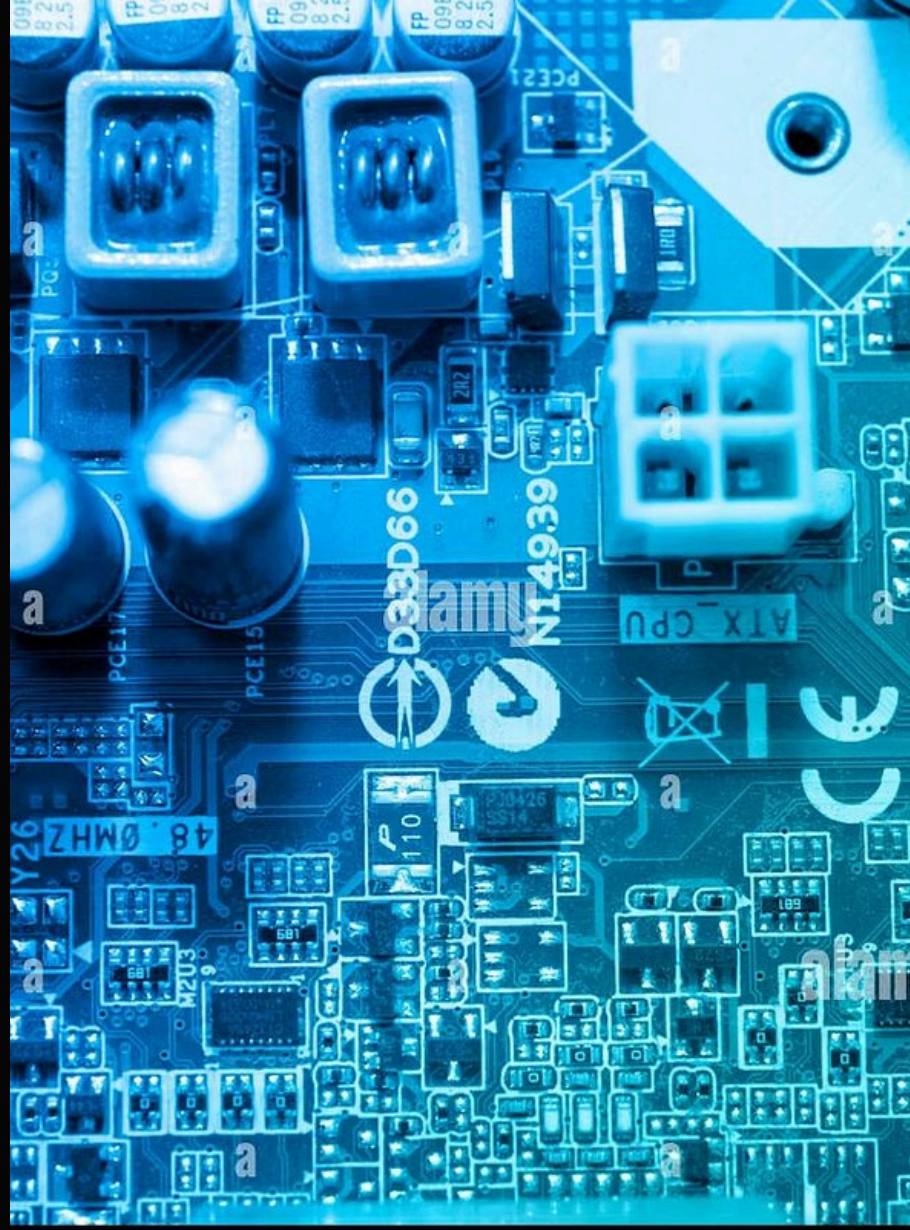
- 0.x.y — initial/unstable development
- 1.0.0 — stable, committed API
- PATCH / MINOR / MAJOR increments correct

Quality Gates

- `cargo test` — all tests green
- `cargo fmt` — consistent formatting
- `cargo clippy` — no lint warnings
- CI passing (GitHub Actions)

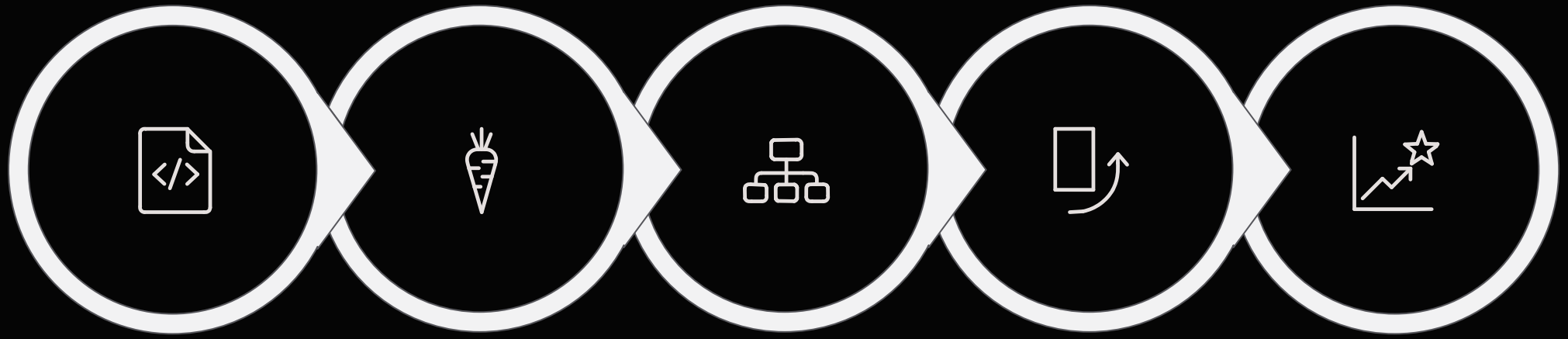
PART 5.3

Rust Compiler Internals: A Guided Tour



The Rust Compiler Pipeline

Every Rust file travels through a carefully ordered pipeline. Each stage performs distinct analysis and transformation, and safety guarantees are enforced across multiple stages — not just one.



Frontend

Parsing → AST → HIR → Type Checking

Middle End

MIR → Borrow Checker → Optimisation Passes

Backend

LLVM IR → Platform Optimisations → Machine Code

Stage 1: Parsing and the AST

The compiler begins by tokenising source text and building an **Abstract Syntax Tree**. The AST is a faithful structural representation of what you wrote — no desugaring yet. You can inspect it directly using nightly flags.

Source Code

```
fn add(x: i32, y: i32) -> i32 {
    x + y
}
```

Simplified AST

```
Function {
  name: "add",
  params: [
    Param { name: "x", type: "i32" },
    Param { name: "y", type: "i32" }
  ],
  return_type: "i32",
  body: Block {
    expr: BinaryOp {
      op: Add,
      left:  Variable("x"),
      right: Variable("y")
    }
  }
}
```

Inspecting the AST

```
# View the raw AST
$ cargo rustc -- -Z unpretty=ast-tree

# View HIR (next stage)
$ cargo rustc -- -Z unpretty=hir

# View HIR with inferred types
$ cargo rustc -- -Z unpretty=hir,typed
```

These flags require a nightly compiler. Install with `rustup toolchain install nightly` and prefix commands with `cargo +nightly rustc ...`.

Stage 2: HIR and Type Checking

The **High-Level IR** is produced by desugaring the AST — eliminating syntactic sugar to produce a simpler, canonical form. Type inference and trait resolution happen at this stage.

Desugaring: if let → match

```
// Source with syntactic sugar
if let Some(x) = option {
    println!("{}", x);
}

// HIR equivalent (desugared)
match option {
    Some(x) => println!("{}", x),
    _       => {}
}
```

Type Checking at HIR Level

- → Infers concrete types for every expression (`i32`, `f64`, etc.)
- → Validates that trait bounds are satisfied at each call site
- → Resolves method calls and operator overloads to concrete implementations
- → Ensures no null or untyped values can pass through

```
let x = 5; // inferred: i32
let y = 5.0; // inferred: f64
let z = x + y; // ERROR: cannot add i32 and f64
```

Stage 3: MIR — Mid-Level IR

MIR is a **control-flow graph** representation — explicit basic blocks, explicit borrows, and no syntactic sugar at all. The borrow checker operates on MIR, making its analysis precise and auditable.

Rust Source

```
fn main() {
    let mut x = 5;
    let y = &mut x;
    *y += 1;
    println!("{}", x);
}
```

Simplified MIR

```
fn main() -> () {
 let mut _1: i32; // x
 let _2: &mut i32; // y
 let mut _3: i32;

 bb0: {
 _1 = const 5_i32;
 _2 = &mut _1;
 _3 = (*_2);
 (*_2) = Add(_3, const 1_i32);
 return;
 }
}

// Inspect MIR:
// $ cargo rustc -- --emit mir
// $ rustc +nightly -Z unpretty=mir src/main.rs
```

Borrow Checking on MIR

The borrow checker analyses **lifetime regions** in the MIR control-flow graph. It detects overlapping mutable and immutable borrows with precision — pointing you directly to the conflicting lines.

Problematic Source

```
fn main() {
    let mut data = vec![1, 2, 3];
    let x = &data[0];   // borrow starts (R1)
    data.push(4);       // mutable borrow (R2) ← ERROR
    println!("{}", x);  // R1 still live
}
```

Compiler Error

```
error[E0502]: cannot borrow `data` as mutable
because it is also borrowed as immutable
  |
3 | let x = &data[0];
  |         ---- immutable borrow occurs here
4 | data.push(4);
  | ^^^^^^^^^^^^ mutable borrow occurs here
5 | println!("{}", x);
  |                - immutable borrow later used here
```

Region Analysis

```
let mut x  = 5;
let r1 = &x;    // region 'a starts
let r2 = &x;    // region 'b starts

println!("{}", r1); // 'a ends
let r3 = &mut x;   // 'c starts – OK, 'a ended

println!("{}", r2); // 'b ends
*r3 += 1;          // 'c ends
```

Multiple immutable borrows (`'a`, `'b`) may coexist. A mutable borrow (`'c`) must not overlap with any live immutable borrow. The borrow checker verifies this across all code paths in the MIR.

Stage 4: LLVM IR and Optimisation

After borrow checking passes, rustc lowers MIR to **LLVM IR** — a platform-independent assembly-like language. LLVM then applies powerful optimisations such as inlining, constant folding, and SIMD vectorisation.

Rust Source → LLVM IR

```
pub fn add_one(x: i32) -> i32 { x + 1 }

// LLVM IR (before optimisation)
define i32 @add_one(i32 %x) {
  %1 = add i32 %x, 1
  ret i32 %1
}

// Called with a constant → folded to 6 at compile time
// add_one(5)  ≡  6  (zero runtime cost)
```

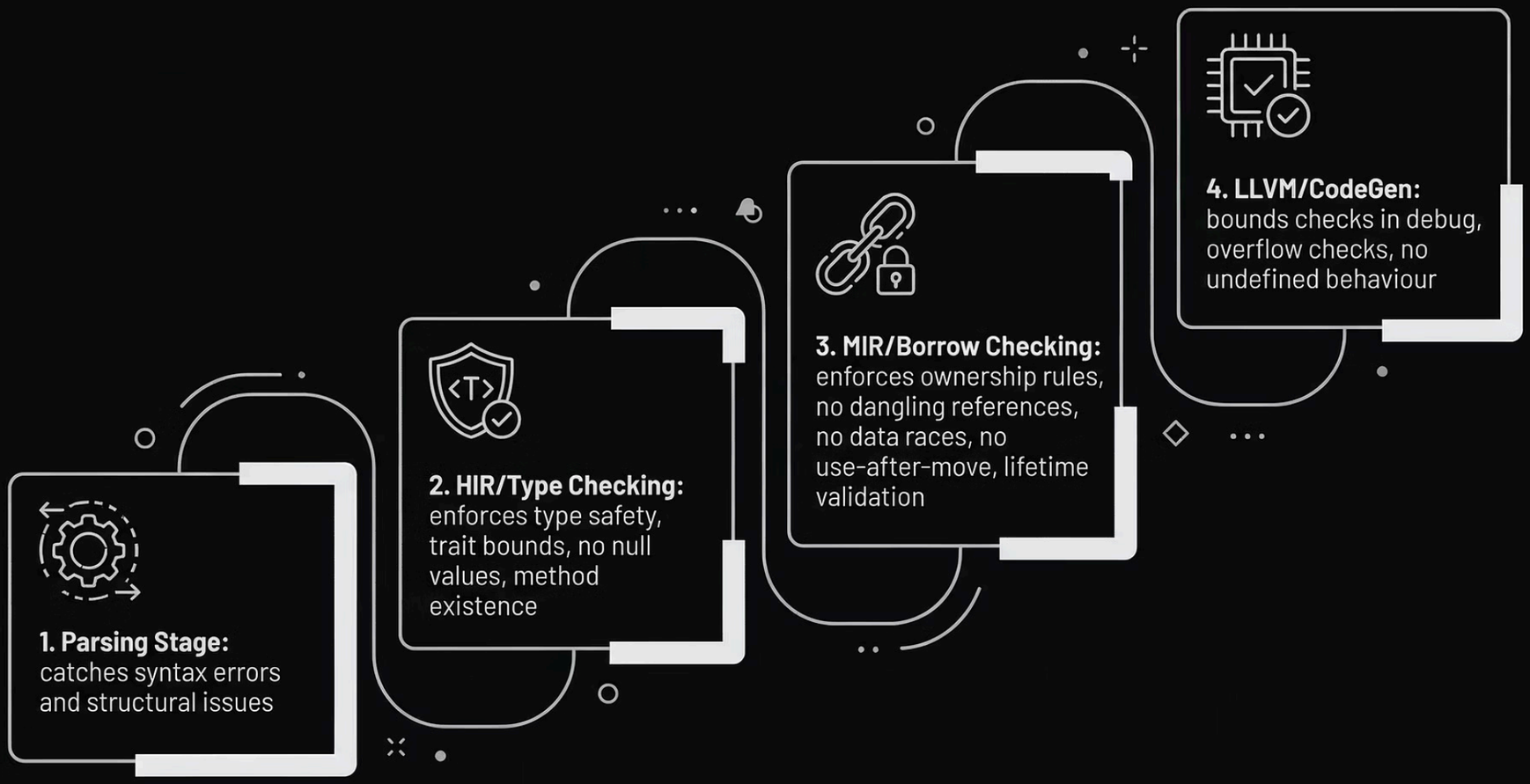
View the IR

```
# Unoptimised
$ rustc --emit llvm-ir src/main.rs

# With full optimisation
$ rustc -C opt-level=3 --emit llvm-ir src/main.rs

# View assembly output
$ rustc --emit asm src/main.rs
$ cargo rustc -- --emit asm
```

Comparing the `-O0` and `-O3` LLVM IR for the same function is one of the best ways to understand what the compiler does on your behalf.

Safety Guaranteed at Every Stage

Rust's safety model is not a single runtime check — it is a cascade of compile-time analyses, each layer catching a different class of error. No safety-critical check is deferred to runtime in safe code.



The compiler **proves** safety before generating any machine code. If it compiles, it is free from the classes of bugs Rust targets — no sanitiser or valgrind run needed.

Zero-Cost Abstractions in Action

Rust's promise: high-level abstractions cost nothing at runtime compared to equivalent hand-written low-level code. Iterator chains, closures, and generics are all **compiled away** entirely.

High-Level Rust

```
fn process_data(data: &[i32]) -> i32 {
    data.iter()
        .filter(|&&x| x > 0)
        .map(|&x| x * 2)
        .sum()
}
```

What the Compiler Generates

```
fn process_data(data: &[i32]) -> i32 {
    let mut sum = 0;
    let mut i   = 0;
    while i < data.len() {
        let x = data[i];
        if x > 0 { sum += x * 2; }
        i += 1;
    }
    sum
}
// Further: SIMD vectorisation, loop unrolling...
```

No Overhead

Abstractions produce identical machine code to hand-written loops

Static Dispatch

Generics are monomorphised — no virtual dispatch cost

Compile-Time Checks

Safety is verified statically; no runtime bookkeeping required

Reading Compiler Diagnostics

Rust's compiler error messages are deliberately rich. They show not just *what* went wrong, but *where* each conflicting borrow begins and ends, and often suggest a remedy. Learning to read them fluently is a core Rust skill.

Source with Borrow Conflict

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

Full Compiler Output

```
error[E0502]: cannot borrow `v` as mutable
because it is also borrowed as immutable
 --> src/main.rs:6:5
  |
4 | let first = &v[0];
  | - immutable borrow occurs here
5 |
6 | v.push(4);
  | ^^^^^^^^^ mutable borrow occurs here
7 |
8 | println!("{}", first);
  | ----- immutable borrow
  | later used here
  |
  = help: consider using a different method
  that doesn't require reallocation, or
  reorder operations to keep the borrow
  contained
```

Exploring Compiler Output

The compiler can emit a wealth of intermediate representations. Use these flags to understand what is happening at each pipeline stage — invaluable for performance work and curiosity alike.

Timing & Verbosity

```
$ cargo build --timings
$ cargo build -vv
$ cargo build --verbose
```

Intermediate Representations

```
$ cargo rustc -- -Z
unpretty=expanded # macros
$ cargo rustc -- --emit mir
$ cargo rustc -- --emit
llvm-ir
$ cargo rustc -- --emit asm
```

Compilation Profiling

```
$ cargo build -Z self-
profile
$ cargo install cargo-
profiler
$ cargo profiler callgrind
```

Build Plan

```
$ cargo build --build-plan
\
 > build.json
# Inspect compilation
order
# and flags for every crate
```

PART 5.4

Lab Session

Hands-on exercises to reinforce Module 5 concepts — from workspace refactoring to exploring the compiler pipeline directly.

Lab Overview: Monolith to Workspace

You will take a single-crate project with several modules and split it into a proper Cargo workspace. This mirrors real-world refactoring work on growing Rust projects.

Starting Point

```
my_app/
├── Cargo.toml
└── src/
    ├── main.rs
    ├── database.rs
    ├── models.rs
    ├── api/
    │   ├── mod.rs
    │   ├── auth.rs
    │   └── routes.rs
    └── utils/
        ├── mod.rs
        ├── strings.rs
        └── validation.rs
```

Target Workspace

```
my_workspace/
├── Cargo.toml      ← workspace root
├── crates/
│   ├── core/       ← models, database
│   ├── api/        ← auth, routes
│   └── utils/      ← strings, validation
└── app/            ← main binary
```

Each crate in the workspace has its own `Cargo.toml` but shares a single `Cargo.lock` and `target/` directory at the workspace root.

Lab Step 1: Create the Workspace Structure

Set up the directory tree and workspace-level `Cargo.toml` before moving any code. Verify with `cargo check --workspace` at the end of each step.

Shell Commands

```
$ mkdir my_workspace && cd my_workspace

# Write workspace Cargo.toml
$ cat > Cargo.toml << EOF
[workspace]
members = [
    "crates/core",
    "crates/api",
    "crates/utils",
    "app",
]
resolver = "2"
EOF

# Create directory structure
$ mkdir -p crates/core/src
$ mkdir -p crates/api/src
$ mkdir -p crates/utils/src
$ mkdir -p app/src
```

Initialise Each Crate

```
$ cd crates/core  && cargo init --lib && cd ../..
$ cd crates/api   && cargo init --lib && cd ../..
$ cd crates/utils && cargo init --lib && cd ../..
$ cd app          && cargo init       && cd ..
```

Verify

```
$ cargo check --workspace
# Should compile with stub lib.rs files
```

Lab Step 2: Move Code to Crates

Populate each crate's `lib.rs` by moving the relevant modules from the monolith. Use `pub use` to re-export the public API at the crate root, keeping downstream callers simple.

crates/core/src/lib.rs

```
//! Core data models and database operations
mod models;
mod database;

pub use models::{User, Product, Order};
pub use database::{Database, Connection};
```

crates/api/src/lib.rs

```
//! API routes and authentication
mod auth;
mod routes;

pub use auth::{Auth, authenticate, login};
pub use routes::{Router, Route, handler};
```

crates/utils/src/lib.rs

```
//! Utility functions
mod strings;
mod validation;

pub use strings::{reverse, truncate, capitalize};
pub use validation::{validate_email, validate_phone};
```

Re-exporting with `pub use` at the crate root means consumers write `use core::User` rather than `use core::models::User` — a cleaner public API boundary.

Lab Step 3: Set Up Dependencies

Wire the crates together with path dependencies and pull in external crates from crates.io. Notice that `serde` could be declared once at the workspace level to avoid version drift.

crates/core/Cargo.toml

```
[package]
name    = "core"
version = "0.1.0"
edition = "2021"

[dependencies]
serde     = { version = "1.0", features = ["derive"] }
thiserror = "1.0"
```

crates/api/Cargo.toml

```
[dependencies]
core  = { path = "../core" }
utils = { path = "../utils" }
serde = "1.0"
axum  = "0.6"
```

app/Cargo.toml

```
[package]
name    = "app"
version = "0.1.0"
edition = "2021"

[dependencies]
core   = { path = "../crates/core" }
api    = { path = "../crates/api" }
utils  = { path = "../crates/utils" }
anyhow = "1.0"
tokio  = { version = "1.0", features = ["full"] }
```

Build the Whole Workspace

```
$ cargo build --workspace
$ cargo test --workspace
```

Lab Step 4: Explore Compiler Output

With the workspace building cleanly, dig into what the compiler actually produces. Each task below reveals a different layer of the pipeline.

1

Build Timings

```
$ cargo build --timings
$ ls target/cargo-timings/
```

2

View MIR

```
$ cargo rustc -- --emit mir
$ find target -name "*.mir"
```

3

View LLVM IR

```
$ RUSTFLAGS="--emit llvm-ir" \
    cargo build --release
$ find target -name "*.ll"
```

4

View Assembly

```
$ cargo rustc -- --emit asm
$ find target -name "*.s"
```

Lab Step 5: Experiment with Release Profiles

Add custom profiles to the workspace `Cargo.toml` and compare the resulting binary sizes. The size difference between debug, release, and size-optimised builds can be dramatic.

workspace/Cargo.toml Additions

```
[profile.dev]
opt-level = 1
debug     = true

[profile.release]
opt-level     = 3
lto           = true
codegen-units = 1
strip         = "symbols"
panic         = "abort"

[profile.size]
inherits  = "release"
opt-level = "z"
strip     = "symbols"
```

Build and Compare

```
$ cargo build --release
$ cargo build --profile size

# Compare binary sizes
$ ls -lh target/release/app
$ ls -lh target/size/app

# Inspect sections (Linux)
$ readelf -S target/release/app
# macOS
$ otool -l target/release/app
# Windows
$ dumpbin /headers target/release/app.exe
```

Lab Challenge Tasks

Choose the challenge level that matches your current confidence. Each tier builds on the previous, and the red tier will take you deep into the compiler's inner workings.

🟢 Level 1: Workspace Setup

- Create the full workspace structure from scratch
- Move all modules to their appropriate crates
- Verify all cross-crate dependencies resolve
- Build the entire workspace with `cargo build --workspace`

🟡 Level 2: Cross-Crate Testing

- Add unit tests to each crate
- Write integration tests that span crate boundaries
- Set up an `app/tests/` directory
- Run `cargo test --workspace` to green

🔴 Level 3: Compiler Exploration

- Examine MIR for a performance-critical function
- Compare LLVM IR between debug and release builds
- Analyse assembly for an iterator chain
- Profile per-crate compile time and identify hotspots

Module 5 Key Takeaways

A concise summary of everything covered across the four parts of Module 5. Return to this slide as a quick reference during your own projects.

Organisation Hierarchy

Packages contain crates; crates contain modules. The `pub` keyword controls every visibility boundary. Use `pub use` to craft clean public APIs.

Cargo Power Features

Workspaces share a single lockfile and target directory. Build profiles let you tune optimisation, LTO, and binary size independently per use case.

Publishing Standards

Complete metadata, good documentation, doc tests, `cargo fmt`, and `cargo clippy` are all prerequisites. Semantic versioning communicates compatibility intent.

Compiler Pipeline

Source → AST → HIR (type check) → MIR (borrow check) → LLVM IR → Machine code. Safety is enforced at multiple stages, never deferred to runtime.

Zero-Cost Abstractions

Iterators, closures, and generics are fully compiled away. High-level Rust produces the same machine code as equivalent hand-written low-level loops.

Compiler Commands Quick Reference

A consolidated reference of `rustc` and Cargo flags for inspecting and tuning compiler behaviour. Keep this handy during the lab and in your day-to-day workflow.

Output Types

```
rustc --emit=asm       # Assembly
rustc --emit=llvm-ir   # LLVM IR
rustc --emit=mir       # MIR
rustc --emit=obj       # Object file
```

Optimisation

```
rustc -C opt-level=3       # Max speed
rustc -C target-cpu=native  # CPU-specific
rustc -C lto=true          # Link-time opt
```

Diagnostics

```
rustc -v                 # Version
rustc --explain E0308    # Explain error
```

Unstable (nightly)

```
rustc -Z unpretty=hir      # Show HIR
rustc -Z unpretty=mir      # Show MIR
rustc -Z dump-mir=main     # Dump MIR
```

Code Generation

```
rustc -C debuginfo=2       # Full debug
rustc -C overflow-checks=on # Overflow check
rustc -C panic=abort       # Abort on panic
```

Via Cargo

```
cargo rustc -- --emit=asm
cargo rustc -- -C opt-level=3
cargo build --timings
cargo build -Z self-profile
```

Additional Resources

Deepen your understanding with these official docs, community tools, and technical deep-dives. The rustc developer guide in particular is essential reading for anyone wanting to contribute to or reason about the compiler.

📚 Official Documentation

- → **The Rust Book** — Chapter 7 (Modules) and Chapter 14 (Cargo)
- → **The Cargo Book** — **doc.rust-lang.org/cargo**
- → **Rustc Dev Guide** — **rustc-dev-guide.rust-lang.org**

🔍 Deep Dives

- → "Introduction to MIR" — Niko Matsakis (blog.rust-lang.org)
- → "LLVM for Rust Developers" — llvm.org documentation

🛠️ Essential Tools

cargo-expand

Expand macros in place

cargo-tree

Full dependency graph

cargo-bloat

Find binary size hotspots

cargo-asm

View generated assembly per function

cargo-audit

Security vulnerability scan

Questions & Answers

Let's open the floor. Common discussion points:

Module Organisation

When should you reach for a workspace vs a deeply nested module tree?

Workspace vs Multiple Repos

What are the trade-offs for team size, release cadence, and CI complexity?

Publishing Best Practices

Versioning strategies, yanking policy, and managing breaking changes gracefully.

Compiler Internals

How does the borrow checker interact with lifetimes in complex generic code?

Performance Optimisation

When is it worth examining MIR or assembly, and what should you look for?

> "Rust gives you control from high-level modules down to generated assembly."

Congratulations! 🎉

You have completed all five modules of **Rust Programming: Systems Reimagined**. From ownership and borrowing to the compiler pipeline — you now have a thorough foundation in one of the world's most exciting systems programming languages.

✓ Module 1

Rust Foundations & Ownership

✓ Module 2

Flow Control & Pattern Matching

✓ Module 3

Data Structures & Collections

✓ Module 4

Traits & Error Handling

✓ Module 5

Modules, Crates, Packaging & Compiler Internals

> Now go build something amazing. 🚀
> The Rust community is waiting for your crate.