# Comprehensive Rust Learning Documentation

I've created five detailed, standalone documents for learning Rust programming from the ground up, focusing on DEEP understanding for experienced developers. Each document provides comprehensive technical details, practical examples, and best practices.

# DOCUMENT 1: Rust Toolchain Deep Dive

## Understanding Rust's development infrastructure from the ground up

Rust's toolchain represents a carefully architected system where **rustup manages versions, rustc compiles code, and cargo orchestrates builds**—all working together to provide deterministic, reproducible development. The toolchain's query-based compiler architecture and sophisticated build system distinguish Rust from traditional compiled languages, enabling both safety guarantees and modern developer ergonomics through incremental compilation and intelligent dependency management.

rustup serves as the meta-tool that installs and manages everything else, acting as a proxy that redirects tool invocations to the appropriate toolchain version. When you type `rustc` or `cargo`, you're actually invoking rustup, which determines the correct toolchain based on a priority hierarchy and forwards the command. This architecture allows seamless switching between stable, beta, and nightly toolchains, or even custom compiler builds for development work.

The compilation process transforms Rust source through multiple intermediate representations—each optimized for different analyses—before producing machine code via LLVM. This multi-stage pipeline enables Rust's unique combination of high-level safety guarantees and low-level performance, with the borrow checker operating at the MIR level where code is simple enough for dataflow analysis but still generic enough to avoid code duplication.

## The rustup architecture that powers version management

rustup operates through a surprisingly elegant system of toolchain specifications, symbolic links, and environment-based overrides. Each toolchain follows the format `<channel>[-<date>][-`

`<host>]`, where the channel can be stable, beta, nightly, or a specific version like 1.75.0. The host triple identifies the platform, and the optional date pins nightly builds to specific days.

Internally, rustup represents toolchains through a hierarchy of structures, starting with `PartialToolchainDesc` for user input and resolving to a fully specified `ToolchainDesc`. The **Manifestation struct** maintains installations by handling component addition and removal, implementing transactional operations with automatic rollback on failure to maintain system consistency. This means if component installation fails halfway through, rustup automatically reverts changes rather than leaving your system in a broken state.

Components are distributed as separate packages that can be mixed and matched. When you install a toolchain, you're actually installing a collection of components: the compiler (rustc), standard library, cargo, documentation, and optional tools like rustfmt and clippy. For nightly toolchains, rustup implements automatic fallback—if the latest nightly is missing a required component, it tries earlier nightly builds until finding one with all requested components.

Toolchain resolution follows a strict priority order that determines which compiler version runs for any given command. **Command-line overrides** take highest priority, allowing `cargo +nightly build` to use nightly regardless of other settings. The `RUSTUP_TOOLCHAIN` environment variable comes next, useful for CI/CD scripts that need specific versions. Directory overrides set with `rustup override set nightly` apply to specific project directories and all subdirectories. The `rust-toolchain.toml` file in your project root (searched up the directory tree) specifies per-project requirements. Finally, the default toolchain applies if nothing else matches.

Toolchains install to `~/.rustup/toolchains/` with separate directories for each version and platform, while compiled binaries from cargo install land in `~/.cargo/bin/`. The separation allows multiple toolchain versions to coexist without conflict, and rustup's proxy binaries in `~/.cargo/bin/` dynamically select the appropriate toolchain for each invocation.

For compiler development, rustup supports linking custom local builds through `rustup toolchain link my-toolchain path/to/sysroot`, allowing you to test compiler changes without installing through the normal channels. This proves essential for contributing to rustc itself or experimenting with compiler modifications.

## How rustc transforms source code into executable binaries

The Rust compiler employs a **query-based architecture** rather than traditional sequential passes, fundamentally changing how compilation works. All major operations are organized as queries that call each other, with results cached on disk to enable incremental compilation. The central `TyCtxt` (Typing Context) struct manages this query system, storing the in-memory cache and coordinating compilation phases.

This design means the compiler doesn't simply process files in order from front to back. Instead, code generation might query for optimized MIR, which queries for borrow-checked MIR, which queries for type-checked HIR, and so on—pulling information through the compilation pipeline on demand. Each query is a pure function: same inputs always produce identical results, ensuring deterministic compilation critical for reproducible builds.

The compiler uses multiple intermediate representations, each optimized for specific purposes. The **Abstract Syntax Tree (AST)** directly represents source structure as returned by the parser, used for syntactic validation and macro expansion. **High-level IR (HIR)** is desugared AST, closer to semantic meaning with implicit elements like elided lifetimes made explicit. HIR remains amenable to type checking and trait resolution while still relatively close to user-written code.

**Typed HIR (THIR)** is fully typed and even more desugared, with method calls and implicit dereferences made completely explicit. This intermediate step eases the transition to MIR by handling the remaining desugaring that's easier to express on a typed representation. **Mid-level IR (MIR)** represents code as a control-flow graph with basic blocks containing simple typed statements. This is where the magic happens—borrow checking, dataflow analysis for uninitialized variables, Rust-specific optimizations, and constant evaluation all operate on MIR.

MIR's power comes from being simple enough for analysis but still generic (not yet monomorphized). Basic blocks consist of statements followed by a terminator that can branch to multiple successors. Locals represent variables (including compiler temporaries), places identify memory locations with projections for field access and dereferencing, and rvalues represent computed values being assigned to places. This flattened representation eliminates nested expressions, making dataflow analysis tractable.

After MIR comes **monomorphization**, where generic code gets instantiated with concrete types. The compiler analyzes which concrete type combinations are actually used and generates specialized versions for each. This produces the zero-cost generics Rust is famous for—generic functions compile to the same code as hand-written versions for each type, with no runtime overhead or virtual dispatch. The downside is increased binary size from code duplication.

Finally, MIR transforms to **LLVM IR**, a typed assembly language with rich annotations. LLVM then applies hundreds of optimization passes, performs architecture-specific code generation, and outputs object files. The linker combines these with dependencies and the runtime to produce the final executable or library.

The compiler itself is written in Rust and compiled with an older version of itself through a **three-stage bootstrap process**. Stage 0 uses a pre-built compiler to build stage 1, stage 1 builds stage 2, and stage 2 builds stage 3. Comparing stage 2 and stage 3 ensures the compiler can correctly compile itself—a crucial self-hosting property.

## cargo's build orchestration and dependency resolution

Cargo combines a build system, package manager, and workspace manager into a cohesive tool that makes Rust projects manageable. Under the hood, cargo implements sophisticated dependency resolution, fingerprinting-based incremental builds, and parallel compilation while maintaining deterministic, reproducible results.

When you run `cargo build`, a complex dance begins. Cargo first parses `Cargo.toml` to understand dependencies, then fetches dependency metadata from registries like crates.io. The **resolver** determines which versions satisfy all constraints, generating or updating `Cargo.lock` with exact versions. Sources download to `~/.cargo/registry/` for caching across all projects on your system.

The build process relies on **fingerprinting**—cargo computes fingerprints for each crate based on source file contents, dependencies, build configurations, compiler settings, and optimization levels. These fingerprints store in `target/debug/.fingerprint/`, and cargo only recompiles crates whose fingerprints changed. This fingerprinting happens at crate granularity, not file granularity, explaining why changing one file in a crate triggers recompilation of the entire crate but not its dependents.

Cargo organizes build artifacts carefully: `target/debug/incremental/` holds incremental compilation data, `target/debug/deps/` contains compiled dependencies, and `target/debug/` stores final binaries. The debug vs release distinction determines which profile settings apply, with release builds in `target/release/` using aggressive optimization at the cost of compilation time.

Different cargo commands optimize for different use cases. `cargo build` produces artifacts, `cargo run` builds and executes the binary, while `cargo check` performs type checking without code generation—running roughly twice as fast as a full build and perfect for development workflows focused on fixing compiler errors. `cargo clippy` adds lint checks beyond the compiler's built-in warnings, and `cargo test` builds and runs all tests.

**Pipelined compilation** (available in nightly) improves parallelism by allowing rustc to start compiling dependent crates before dependencies fully complete. Normally, crate B must wait for crate A to finish entirely. With pipelining, B can start as soon as A's metadata generation completes, overlapping compilation phases for better CPU utilization.

Build scripts (`build.rs`) run before crate compilation, enabling code generation, C dependency compilation, platform feature detection, and compiler configuration. The build script communicates with cargo through special `cargo:` prefixed println statements, controlling rerun conditions, linking parameters, and rustc configuration.

## Cross-compilation and target triples explained

Rust treats cross-compilation as a first-class feature—**every rustc is inherently a cross-compiler** capable of targeting any supported platform. This fundamentally differs from toolchains like GCC where you need separate compilers for each target.

Target triples follow the format `<architecture><sub>-<vendor>-<system>-<abi>`. Common examples include `x86_64-unknown-linux-gnu` for 64-bit Linux with glibc, `x86_64-pc-windows-msvc` for 64-bit Windows with MSVC toolchain, `aarch64-unknown-linux-gnu` for 64-bit ARM Linux, and `wasm32-unknown-unknown` for WebAssembly. The "unknown" vendor indicates no specific vendor, while the ABI specifies C library and calling conventions.

Cross-compiling requires three components: the Rust standard library for the target (installed via `rustup target add <triple>`), a linker for the target (usually from a C cross-compiler toolchain), and system libraries for the target if linking against C dependencies. You configure the linker in `~/.cargo/config.toml` under `[target.<triple>]` sections, specifying which linker binary and archiver to use.

The **cross** tool simplifies cross-compilation by providing Docker containers with complete toolchains for numerous targets. It's a drop-in cargo replacement: `cross build --target=armv7-unknown-linux-gnueabihf` handles toolchain setup automatically, eliminating manual configuration and missing library issues.

## Debugging with rust-gdb and rust-lldb

Rust provides debugger wrappers that enhance GDB and LLDB with Rust-aware features, translating the debugger experience to understand Rust's type system, enums, and ownership model. These tools parse DWARF or PDB debug information generated by rustc, extended with Rust-specific metadata.

**rust-gdb** wraps GNU Debugger, automatically loading Rust pretty-printers that understand Rust's type system. It includes a custom expression parser supporting a subset of Rust expressions, allowing you to evaluate Rust-like expressions in the debugger. Common commands include `break main` to set breakpoints, `run arg1 arg2` to execute with arguments, `next` to step over, `step` to step into, `print variable` to display values, and `backtrace` to show the call stack.

**rust-lldb** serves as the default debugger on macOS due to better system integration. It loads Python-based formatters, implementing Rust type display through Python scripts. Commands

mirror GDB's semantics: `breakpoint set -n main`, `run args`, `next`, `step`, `print variable`, and `bt` for backtraces.

For debugging sessions, compile with debug information using `cargo build` (debug builds include debug info by default). Environment variables enhance debugging: `RUST_BACKTRACE=1` enables backtraces on panic, `RUST_BACKTRACE=full` includes all stack frames, and `RUST_LOG=debug` enables logging.

## Development environment setup for 2024-2025

Install Rust through rustup exclusively—never use system package managers. Platform-specific requirements vary: Windows needs Visual Studio 2022 with C++ workload, Linux needs build-essential for GCC, and macOS requires Xcode Command Line Tools.

**VS Code** dominates with the **rust-analyzer** extension providing language server support—type checking, code completion, inline errors, and refactoring. Add CodeLLDB for debugging and Even Better TOML for Cargo.toml syntax. Configure with `"rust-analyzer.checkOnSave.command": "clippy"` to run clippy automatically.

**RustRover** (JetBrains' dedicated Rust IDE, 2024) provides polished commercial experience with built-in debugging, excellent type inference, and powerful refactoring.

Essential tools include **rustfmt** for formatting (`cargo fmt`), **clippy** for linting (`cargo clippy`), and **cargo check** for fast type checking. Alternative linkers like **mold** (Linux) dramatically reduce linking time for large projects.

**Best practices** for 2024-2025 include using rust-analyzer for instant feedback, cargo check for quick verification, clippy for code quality, and rustfmt for consistent style. Combined with an alternative linker and optimized profiles, compilation becomes almost imperceptible for small changes.

---

# DOCUMENT 2: Rust Compiler & Compilation Model

## Inside rustc: the query-driven architecture

The Rust compiler implements a fundamentally different architecture than traditional compilers. Instead of sequential passes, **rustc uses a query-driven system where compilation phases pull information on demand, caching results for incremental rebuilds**. This enables Rust's sophisticated static analysis with reasonable compilation speeds.

The **TyCtxt** (Typing Context) struct manages the query system, storing the in-memory cache and coordinating all compilation phases. Each query is a pure function—identical inputs always produce identical outputs—ensuring deterministic compilation critical for reproducible builds.

Query modifiers control caching: `eval_always` forces re-execution, `cache_on_disk_if` conditionally persists results, `no_hash` skips fingerprinting, and `anon` creates anonymous dependency nodes. This architecture directly enables incremental compilation through the red-green algorithm.

## Complete compilation pipeline: seven transformation stages

**Stage 1: Lexing** converts character streams to tokens. The low-level lexer operates on raw bytes, supporting Unicode throughout. The high-level lexer performs **string interning**—storing unique strings once in an arena allocator for cheap equality comparisons and reduced memory usage.

**Stage 2: Parsing** constructs an Abstract Syntax Tree using recursive descent. The parser implements error recovery, attempting to parse a superset of Rust's grammar and generating errors rather than stopping at the first problem. This produces better diagnostics for code with multiple errors.

**Stage 3: HIR Lowering** transforms AST to High-level IR through desugaring. For loops become loops with iterators, `if let` becomes `match` statements, and `async fn` transforms to state machines. HIR makes implicit elements explicit while remaining relatively close to source structure.

**Stage 4: Type Checking** performs type inference using Hindley-Milner with extensions, trait resolution to pair implementations with references, and type checking to verify operations match types. All types are interned in arena allocators for memory efficiency.

**Stage 5: MIR Lowering** generates Mid-level IR, a control-flow graph where code organizes into basic blocks. MIR flattens all expressions into sequences of assignments and explicit control flow. **Borrow checking** executes on MIR through dataflow analysis, tracking loans, paths, and facts to enforce borrowing rules.

**MIR optimization** applies Rust-specific passes: dead code elimination, constant propagation, inlining, and pattern-specific optimizations. These work on generic MIR, benefiting all instantiations.

**Stage 6: Monomorphization** instantiates generic code with concrete types. The compiler generates specialized versions for each combination: `Vec<i32>::push` and `Vec<String>::push`

produce separate functions. This enables zero-cost abstraction—no runtime overhead or virtual dispatch.

**Stage 7: Code Generation** translates MIR to LLVM IR, applies hundreds of LLVM optimization passes, generates architecture-specific machine code, and links everything into the final executable.

## Incremental compilation: red-green algorithm

The **red-green algorithm** determines which queries must re-execute (red) versus which can reuse previous results (green). On first compilation, rustc executes queries, builds the dependency DAG, stores results to disk in `target/debug/incremental/`, and hashes all results using stable hashing.

On subsequent compilations, the **try-mark-green algorithm** recursively colors queries. If all dependencies are green and result hash unchanged, the query is green (reuse cached result). If any dependency is red or result changed, the query is red (must re-execute). Dependencies are visited in original execution order because control flow can change.

**Stable hashing** ensures fingerprints remain valid across compilations using stable identifiers like `DefPath` rather than internal IDs. This balances cache hit rates against overhead, achieving 10x speedups for small changes while maintaining correctness.

## Optimization levels: performance tradeoffs

`-C opt-level=0` (debug default) applies no optimizations, prioritizing compilation speed. Produces large, slow binaries with full debuggability.

`-C opt-level=1` enables basic optimizations without significantly slowing compilation. Good balance for development.

`-C opt-level=2` performs aggressive speed optimizations: inlining, loop unrolling, vectorization. Standard release level for clang. Sometimes faster than `-O3` due to better instruction cache utilization.

`-C opt-level=3` (cargo release default) maximizes speed with more aggressive transformations. Longest compilation, occasionally slower than `-O2` due to code bloat.

`-C opt-level="s"` and `"z"` optimize for size, disabling optimizations that significantly increase code size. Suits embedded systems and WebAssembly.

**Link-time optimization (LTO)** enables whole-program optimization. **Thin LTO** provides best tradeoff—80-90% of Fat LTO's benefits with parallelizable compilation. **Fat LTO** maximizes performance with single-threaded whole-program optimization.

**Codegen units** control parallelism. More units = faster compilation but fewer optimization opportunities. `codegen-units = 1` enables maximum optimization but forces single-threaded code generation.

## Reading assembly output

Generate assembly with `rustc --emit asm file.rs` or use **cargo-show-asm** for better experience: `cargo asm --lib function_name` displays specific functions, `cargo asm --lib --rust function_name` interleaves source code.

**Intel syntax** (`mov rax, 42`) is more readable than AT&T syntax (`movq $42, %rax`). **x86-64 conventions**: rax for return values, rdi/rsi/rdx/rcx/r8/r9 for first six arguments, rsp for stack pointer.

**Common patterns** reveal optimization: function prologues/epilogues show stack management, loop unrolling replicates loop bodies, SIMD vectorization processes multiple data elements simultaneously, and inlining eliminates call overhead.

## Compiler flags and configuration

**Codegen flags** (`-C`): `opt-level`, `debuginfo`, `target-cpu=native`, `target-feature=+avx2`, `lto`, `codegen-units`, `panic=abort|unwind`, `overflow-checks`.

**Emission flags** (`--emit`): `asm`, `llvm-ir`, `mir`, `obj`, `link`. Multiple outputs combine with commas.

**Unstable flags** (`-Z`): `time-passes` shows compilation timing, `print-type-sizes` reports memory layout, `mir-opt-level` controls MIR optimization separately.

**Attributes** control codegen: `#[inline]`, `#[inline(always)]`, `#[inline(never)]`, `#[cold]`, `#[target_feature(enable = "avx2")]`, `#[must_use]`, `#[deprecated]`.

**Cargo profiles** provide the most maintainable configuration, centralizing settings in Cargo.toml for consistency across team members and CI builds.

# DOCUMENT 3: Memory Safety & Ownership System

## Rust's revolutionary memory management

Rust achieves memory safety without garbage collection through an **ownership system that enforces safety rules at compile time**. Every value has exactly one owner, borrowing rules prevent simultaneous mutable and immutable access, and lifetimes ensure references never outlive referents—all checked statically with zero runtime cost.

This represents a third option beyond manual memory management (C/C++) and garbage collection (Java, Python). **Compile-time verification** provides C-level performance with high-level safety guarantees, eliminating use-after-free, double-free, and data races.

## Ownership: three fundamental rules

1. **Every value has a single owner** responsible for deallocation
2. **Only one owner at a time** (no shared ownership by default)
3. **When owner goes out of scope, value is dropped** automatically

These rules eliminate entire bug classes. Ownership is zero-cost—all enforcement happens during compilation through the borrow checker operating on MIR.

At the memory level, `String` consists of three stack words: pointer to heap, length, capacity. When `s2 = s1` executes, only stack words are copied, but **Rust marks s1 invalid**. This prevents double-free—only one deallocation occurs.

Transfer of ownership is a **move**. Most types are move-only: String, Vec, Box, structs with non-Copy fields. When passing to functions or assigning, ownership transfers and original binding becomes unusable.

**Copy trait** enables duplication for cheap-to-copy types fitting in a machine word or two. Includes primitives, tuples of Copy types, fixed arrays. Cannot implement Copy for types with Drop or non-Copy fields.

## Borrowing: temporary access without ownership

**Borrowing rules**: at any time, either **one mutable reference OR any number of immutable references** (not both), and **references must always be valid**. These prevent data races at compile time.

**Immutable borrow** (`&T`) allows reading without modification. Multiple immutable borrows can coexist. When function returns, reference goes out of scope but owned data remains.

**Mutable borrow** (`&mut T`) allows modification but **only one mutable reference exists at any time**. Cannot have mutable borrow while immutable borrows exist.

**Non-Lexical Lifetimes (NLL)** determine borrow scope by last use rather than lexical scope, accepting more safe programs:

```
let mut x = 5;
let y = &x;
println!("{}", y); // Last use of y

let z = &mut x; // OK! After y's last use
*z += 1;
```

The **borrow checker** uses dataflow analysis on MIR, tracking loans, paths, and liveness, computing which borrows are active at each program point and verifying no conflicts.

## Lifetimes: tracking reference validity

**Lifetimes** are named regions in the control-flow graph where references are valid. The compiler verifies references never outlive their data.

Lifetime annotations make relationships explicit:

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

The `'a` indicates returned reference will be valid as long as both inputs are valid. Annotations describe existing relationships rather than changing lifetimes.

**Lifetime elision rules** infer lifetimes in simple cases: each elided parameter lifetime gets a distinct lifetime, single input lifetime assigns to all outputs, `&self` lifetime assigns to all outputs.

**Structs with references** require lifetime parameters:

```
struct ImportantExcerpt<'a> {
    part: &'a str, // Cannot outlive struct
}
```

`'static` **lifetime** indicates data living for entire program duration: string literals and global variables.

## Move vs Copy semantics

**Move semantics** (default): ownership transfers, source becomes invalid. "Move" mechanically copies bits but compiler marks source invalid, preventing double-free.

**Copy semantics** (opt-in): types implementing Copy are duplicated on assignment. All primitives, booleans, char, tuples of Copy types, fixed arrays of Copy types. Cannot implement Copy with Drop or non-Copy fields.

**Clone trait** enables explicit deep copying for non-Copy types: `let v2 = v1.clone();`

## Stack vs Heap allocation

**Stack**: LIFO structure, fixed size known at compile time, extremely fast (pointer bump), automatically cleaned up, limited size (~8MB main thread).

**Heap**: dynamic size, slower allocation, larger capacity, explicit allocation via Box/Vec/String, freed when owner dropped (RAII).

**When to use**: Stack for fixed-size short-lived data, heap for dynamic collections, data outliving function scope, large structures, recursive data.

## Drop trait and RAII

**Drop trait** provides destructor called automatically when owner goes out of scope:

```
impl Drop for FileGuard {
    fn drop(&mut self) {
        println!("Closing file");
        // Cleanup code
    }
}
```

Fields drop in declaration order. Cannot call `drop()` explicitly, use `std::mem::drop(x)` to transfer ownership to drop function. `std::mem::forget(x)` prevents drop, leaking memory.

## Interior mutability: Cell and RefCell

**Problem**: Rust's default inherited mutability requires `&mut` for mutation. Interior mutability allows mutation through `&` references.

**Cell** for Copy types only:

```
let count = Cell::new(0);
count.set(count.get() + 1); // Mutate through &self
```

`get()` returns copy, `set()` replaces value. No borrowing, zero runtime cost, no panic risk.

**RefCell** for any type:

```
let data = RefCell::new(vec![1, 2]);
data.borrow_mut().push(3); // Mutate through &self
```

`borrow()` returns `Ref<T>` (shared reference), `borrow_mut()` returns `RefMut<T>` (mutable reference). **Runtime checking**—panics if borrowing rules violated.

**When to use**: Cell for simple Copy types with zero cost, RefCell for non-Copy types needing references, testing/mocking scenarios.

**Common pattern**: `Rc<RefCell<T>>` for shared mutable state in single-threaded code.

## Smart pointers: Box, Rc, Arc, Weak

**Box** provides single ownership with heap allocation. Use for recursive types, large data avoiding stack overflow, trait objects:

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

**Rc** provides reference counting for multiple ownership (single-threaded):

```
let a = Rc::new(5);
let b = Rc::clone(&a); // Increment counter
```

`clone()` only increments counter (cheap). Not thread-safe. Cannot have cycles without Weak.

**Arc** provides atomic reference counting (thread-safe):

```
let data = Arc::new(vec![1, 2, 3]);
let data_clone = Arc::clone(&data);
thread::spawn(move || println!("{:?}", data_clone));
```

Uses atomic operations, slightly slower than Rc. Implements Send + Sync.

**Weak** provides non-owning references for breaking cycles:

```
let weak = Rc::downgrade(&rc);
if let Some(rc) = weak.upgrade() {
    // Use rc
}
```

Doesn't prevent deallocation. Useful for parent-child relationships, caches.

**Combining with mutability**: - Single-threaded: `Rc<RefCell<T>>` - Multi-threaded: `Arc<Mutex<T>>` or `Arc<RwLock<T>>`

## How Rust achieves memory safety without GC

**Compile-time guarantees**: Ownership prevents use-after-free, borrow checker prevents data races, lifetimes prevent dangling pointers, type system prevents invalid operations.

**Zero-cost abstractions**: All checks at compile time, no runtime overhead, no GC pause times, predictable performance.

**RAII pattern**: Deterministic destruction, resources freed when owner goes out of scope, no manual management.

**What Rust prevents**: Use-after-free, double-free, data races, dangling pointers, iterator invalidation—all at compile time.

**Performance**: No GC pauses, predictable latency, better cache locality, lower memory overhead than GC languages. Similar performance to C/C++ while eliminating bug classes.

# DOCUMENT 4: Core Rust Syntax & Language Constructs

## Variables, mutability, and shadowing

**Immutability by default**:

```
let x = 5;  // Immutable
// x = 6;   // ERROR
```

**Mutable variables**:

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

**Shadowing** creates new binding, allows type changes:

```
let x = 5;
let x = x + 1;  // New binding
let x = x * 2;  // Another new binding
let spaces = "   ";
let spaces = spaces.len();  // Type change OK
```

**mut vs shadowing**: mut modifies existing (type unchanged), shadowing creates new (type can change).

## Data types: scalar and compound

**Integers**: i8-i128, isize (signed) / u8-u128, usize (unsigned). Literals support underscores: `98_222`, hex `0xff`, octal `0o77`, binary `0b1111_0000`, byte `b'A'`.

**Overflow behavior**: Debug panics, release wraps. Explicit methods: `wrapping_*`, `checked_*`, `overflowing_*`, `saturating_*`.

**Floating-point**: f32, f64 (default). IEEE 754 standard.

**Boolean**: true/false (1 byte).

**Character**: 4-byte Unicode Scalar Value. Supports emoji: `let emoji = '😻';`

**Tuples**: Fixed-length, heterogeneous:

```
let tup: (i32, f64, u8) = (500, 6.4, 1);
let (x, y, z) = tup;  // Destructuring
let five_hundred = tup.0;  // Index access
```

**Arrays**: Fixed-length, homogeneous, stack-allocated:

```
let a = [1, 2, 3, 4, 5];
let a: [i32; 5] = [1, 2, 3, 4, 5];
let a = [3; 5];  // [3, 3, 3, 3, 3]
```

## Functions and methods

**Function syntax**:

```
fn add(x: i32, y: i32) -> i32 {
    x + y  // No semicolon = return
}
```

**Associated functions** (static methods):

```
impl Point {
    fn new(x: f64, y: f64) -> Point {
        Point { x, y }
    }
}
let p = Point::new(1.0, 2.0);
```

**Methods**:

```
impl Point {
    fn distance(&self) -> f64 { }       // Immutable
    fn translate(&mut self, dx: f64) { }  // Mutable
    fn consume(self) -> (f64, f64) { }    // Ownership
}
```

## Control flow

**if expressions**:

```
let number = if condition { 5 } else { 6 };
```

**Loops**:

```
// loop with return value
let result = loop {
    counter += 1;
    if counter == 10 {
        break counter * 2;
    }
};

// Loop labels
'outer: loop {
    loop {
        break 'outer;
    }
}

// for with iterators
for element in array.iter() { }
for i in 0..10 { }
```

## Pattern matching in depth

**match expression** (exhaustive):

```
match value {
    1 => println!("one"),
    2 | 3 => println!("two or three"),
    4..=9 => println!("range"),
    _ => println!("anything"),
}
```

**Destructuring patterns**:

```
// Structs
match point {
    Point { x, y: 0 } => println!("On x axis"),
    Point { x: 0, y } => println!("On y axis"),
    Point { x, y } => println!("({}, {})", x, y),
}

// Enums with data
match msg {
    Message::Quit => { },
    Message::Move { x, y } => { },
    Message::Write(text) => { },
}

// Tuples
match tuple {
    (0, y, z) => { },
    (1, ..) => { },  // Ignore rest
}
```

**Guards**:

```
match num {
    Some(x) if x < 5 => println!("less than five"),
    Some(x) => println!("{}", x),
    None => (),
}
```

**@ bindings**:

```
match msg {
    Message::Hello { id: id_var @ 3..=7 } => {
        println!("Found id: {}", id_var)
    },
}
```

**if let / while let**:

```
if let Some(value) = optional {
    println!("{}", value);
}

while let Some(top) = stack.pop() {
    println!("{}", top);
}
```

## Enums and algebraic data types

**Enum definition**:

```
enum Message {
    Quit,                       // Unit
    Move { x: i32, y: i32 },    // Struct-like
    Write(String),              // Tuple
    ChangeColor(i32, i32, i32),
}
```

**Option** eliminates null pointer errors:

```
enum Option<T> {
    Some(T),
    None,
}
// Forces explicit handling
```

**Result** for error handling:

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

## Structs

**Named structs**:

```
struct User {
    username: String,
    email: String,
}

// Field init shorthand
User { username, email }

// Struct update syntax
User { email: new_email, ..user1 }
```

**Tuple structs**:

```
struct Color(i32, i32, i32);
let black = Color(0, 0, 0);
```

**Unit structs** (no data):

```
struct AlwaysEqual;
```

## Traits and trait bounds

**Trait definition**:

```
pub trait Summary {
    fn summarize(&self) -> String;

    // Default implementation
    fn default_summary(&self) -> String {
        String::from("(Read more...)")
    }
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}: {}", self.headline, self.content)
    }
}
```

**Trait bounds**:

```
// impl Trait syntax
fn notify(item: &impl Summary) { }

// Generic with bound
fn notify<T: Summary>(item: &T) { }

// Multiple bounds
fn notify<T: Summary + Display>(item: &T) { }

// where clause
fn some_function<T, U>(t: &T, u: &U)
where
    T: Display + Clone,
    U: Clone + Debug,
{ }
```

**Associated types**:

```rust
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

**Orphan rule**: Can only implement trait if trait OR type is local.

## Generics and monomorphization

**Generic functions**:

```rust
fn largest<T: PartialOrd>(list: &[T]) -> &T {
    let mut largest = &list[0];
    for item in list {
        if item > largest {
            largest = item;
        }
    }
    largest
}
```

**Generic structs**:

```rust
struct Point<T, U> {
    x: T,
    y: U,
}

impl<T> Point<T, T> {
    fn x(&self) -> &T { &self.x }
}

// Concrete implementation
impl Point<f32, f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```

**Monomorphization** generates specialized versions for each concrete type at compile time:

```
// You write:
let integer = Some(5);
let float = Some(5.0);

// Compiler generates (conceptually):
enum Option_i32 { Some(i32), None }
enum Option_f64 { Some(f64), None }
```

**Zero-cost abstraction**: Static dispatch, no runtime overhead, aggressive optimizations per type, but increased binary size.

## Error handling

**Result**:

```
fn divide(a: f64, b: f64) -> Result<f64, String> {
    if b == 0.0 {
        Err("Cannot divide by zero".to_string())
    } else {
        Ok(a / b)
    }
}
```

**? operator** for early return on error:

```
fn read_username() -> Result<String, io::Error> {
    let mut file = File::open("username.txt")?;
    let mut username = String::new();
    file.read_to_string(&mut username)?;
    Ok(username)
}

// Works with Option too
fn last_char(text: &str) -> Option<char> {
    text.lines().next()?.chars().last()
}
```

**Combinators**:

```
some_option.map(|x| x * 2)
Some(5).and_then(|x| Some(x * 2))
value.unwrap_or(default)
value.unwrap_or_else(|| compute_default())
Some(5).ok_or("error message")
```

## Closures

**Capture modes**:

```
// Immutable borrow
let list = vec![1, 2, 3];
let borrows = || println!("{:?}", list);

// Mutable borrow
let mut list = vec![1, 2, 3];
let mut borrows_mut = || list.push(7);

// Move ownership
let list = vec![1, 2, 3];
let consumes = move || println!("{:?}", list);
```

**Fn traits hierarchy**: - **FnOnce**: Takes ownership, single call - **FnMut**: Mutable borrow, multiple calls - **Fn**: Immutable borrow, multiple calls

**Function pointers**:

```
fn add_one(x: i32) -> i32 { x + 1 }
let f: fn(i32) -> i32 = add_one;
// Function pointers implement all Fn traits
```

## Iterators

**Iterator trait**:

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

**Three iterator methods**:

```
v.iter()        // &T      - immutable borrow
v.iter_mut()    // &mut T  - mutable borrow
v.into_iter()   // T       - takes ownership
```

**Lazy evaluation**: Iterators don't execute until consumed:

```rust
let iter = v.iter().map(|x| x + 1); // Nothing executed yet
let result: Vec<_> = iter.collect(); // Now executes
```

**Iterator adapters** (lazy):

```rust
.map(|x| x * 2)
.filter(|&x| x > 5)
.filter_map(|x| if x > 0 { Some(x) } else { None })
.chain(other_iter)
.zip(other_iter)
.enumerate()  // Add index
.take(n)
.skip(n)
.flat_map(|x| x.iter())
.flatten()
```

**Consumers** (terminal):

```rust
.collect()
.sum()
.count()
.any(|x| x > 5)
.all(|x| x > 0)
.find(|&x| x == 2)
.fold(0, |acc, x| acc + x)
.for_each(|x| println!("{}", x))
```

**Performance**: Zero-cost abstraction—compiles to same code as hand-written loops. Lazy evaluation prevents intermediate allocations.

## Best practices summary

- Prefer immutability, use `mut` sparingly
- Result for recoverable errors, panic! for unrecoverable
- Use ? operator for error propagation
- Generics provide zero runtime cost
- Prefer iterators over manual loops
- Match is exhaustive—compiler ensures all cases
- Traits enable polymorphism without runtime overhead

# DOCUMENT 5: Project Structure & Build System

## Cargo.toml: complete manifest reference

**[package] section** defines fundamental metadata:

```toml
[package]
name = "my_project"
version = "0.1.0"
edition = "2021"
rust-version = "1.75.0"  # MSRV
description = "Plain text description"
documentation = "https://docs.rs/my-crate"
repository = "https://github.com/user/repo"
license = "MIT OR Apache-2.0"
keywords = ["key1", "key2", "key3"]  # Max 5
categories = ["category1"]  # Max 5
```

**Target definitions**:

```toml
[[bin]]
name = "my-binary"
path = "src/bin/my-binary.rs"
required-features = ["feature1"]

[lib]
name = "my_lib"
path = "src/lib.rs"
crate-type = ["lib", "rlib", "cdylib"]
```

**Dependencies**:

```toml
[dependencies]
serde = "1.0"
tokio = { version = "1.0", features = ["full"] }
my-local = { path = "../my-local" }
my-git = { git = "https://github.com/user/repo", branch = "main" }

[dev-dependencies]  # Tests only
[build-dependencies]  # Build scripts only

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

**Features**:

```toml
[features]
default = ["std"]
std = []
serde-support = ["dep:serde"]
full = ["std", "serde", "advanced"]
```

**Profiles**:

```toml
[profile.dev]
opt-level = 0
debug = true

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

[profile.release-small]
inherits = "release"
opt-level = "z"
lto = true
```

**Workspaces**:

```toml
[workspace]
members = ["crate1", "crate2", "crates/*"]
exclude = ["crates/old"]
resolver = "2"

[workspace.package]
version = "1.0.0"
edition = "2021"

[workspace.dependencies]
shared-dep = "1.0"
```

## Module system: organization and visibility

**Module declaration**:

```rust
// Inline module
mod my_module {
    pub fn public_function() {}
    fn private_function() {}
}

// File-based: mod foo; looks for src/foo.rs or src/foo/mod.rs
```

**Module tree structure**:

```
src/
├── lib.rs (or main.rs)
├── config.rs          // mod config;
├── network/
│   ├── mod.rs        // mod network;
│   ├── server.rs     // In network/mod.rs: mod server;
│   └── client.rs
└── utils.rs
```

**Visibility rules**:

```rust
pub              // Public to all
pub(crate)       // Public within crate only
pub(super)       // Public to parent module
pub(in path)     // Public to specific ancestor

mod outer {
    pub mod inner {
        pub fn public_fn() {}           // Everywhere
        pub(crate) fn crate_fn() {}     // Within crate
        pub(super) fn parent_fn() {}    // In 'outer'
        fn private_fn() {}              // Only 'inner'
    }
}
```

**Path resolution**:

```rust
// Absolute paths
use crate::config::Settings;
use std::collections::HashMap;

// Relative paths
use super::parent_module;
use self::sibling_module;

// Re-exports flatten API
pub use self::deeply::nested::module::ImportantType;
```

## Crates: binary vs library

**Library crates**: - Single library per package (src/lib.rs) - Provides reusable functionality - Default types: `lib`, `rlib` - Other types: `dylib`, `staticlib`, `cdylib`

**Binary crates**: - Create executables - src/main.rs is default binary - Multiple binaries via src/bin/*.rs - Must have `fn main()`

**Common pattern**: Both lib and bin:

```
src/
├── lib.rs        // Library with main logic
└── main.rs       // Thin binary wrapper
```

Allows other crates to use your library, easy testing, binary as CLI wrapper.

## Workspaces: multi-crate projects

**Workspace structure**:

```toml
# Root Cargo.toml
[workspace]
members = ["app", "utils", "shared"]
resolver = "2"

[workspace.package]
version = "1.0.0"
edition = "2021"

[workspace.dependencies]
tokio = "1.0"
```

**Benefits**: Single Cargo.lock, shared target/ directory, consistent dependency versions, single command for all crates.

**Commands**:

```
cargo build --workspace     # Build all
cargo test --workspace      # Test all
cargo build -p crate-name   # Build one
```

**Dependency inheritance**:

```toml
[package]
version.workspace = true
edition.workspace = true

[dependencies]
tokio.workspace = true
```

## Dependency management and versioning

**SemVer specifications**: - Caret: `"^1.2.3"` = `>=1.2.3, <2.0.0` (default) - Tilde: `"~1.2.3"` = `>=1.2.3, <1.3.0` - Wildcard: `"1.*"` = `>=1.0.0, <2.0.0` - Exact: `"=1.2.3"` - Range: `">=1.2, <1.5"`

**Cargo.lock** records exact versions for reproducibility. Generated automatically, updated when dependencies change. Commit for binaries, not for libraries.

**Dependency types**: - Path: `{ path = "../my-crate" }` - Git: `{ git = "...", branch = "main" }` - Registry: `"1.0"` - Renamed: `gtk = { package = "gtk4", version = "0.5" }`

## Features and conditional compilation

**Define features**:

```toml
[features]
default = ["std"]
std = []
serde = ["dep:serde"]
full = ["std", "serde", "advanced"]
```

**Use in code**:

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

#[cfg(feature = "serde")]
impl Serialize for MyType { }

#[cfg(all(feature = "std", target_os = "linux"))]
fn linux_std_only() {}
```

**Common cfg conditions**: - `target_os = "windows"`, `"linux"`, `"macos"` - `target_arch = "x86_64"`, `"aarch64"` - `target_pointer_width = "32"`, `"64"` - `unix`, `windows` - `debug_assertions` - `test`

**Enable features**:

```
cargo build --features "feature1,feature2"
cargo build --no-default-features
cargo build --all-features
```

**Best practices**: Features should be additive, avoid mutually exclusive features, document in README, test different combinations.

## Testing: comprehensive system

**Unit tests** (in src/):

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

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_add() {
        assert_eq!(add(2, 2), 4);
    }

    #[test]
    #[should_panic]
    fn test_panic() {
        panic!("Expected");
    }

    #[test]
    #[ignore]
    fn expensive_test() {
        // cargo test -- --ignored
    }
}
```

**Integration tests** (in tests/):

```
tests/
├── integration_test.rs
├── another_test.rs
└── common/
    └── mod.rs  // Shared utilities
```

Each file is separate crate testing public API only.

**Documentation tests**:

```
/// Adds two numbers.
///
/// # Examples
///
/// ```
/// use my_crate::add;
/// assert_eq!(add(2, 2), 4);
/// ```
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

**Doc test attributes**:

```
/// ```no_run
/// // Compiles but doesn't run
/// ```

/// ```ignore
/// // Completely ignored
/// ```

/// ```should_panic
/// // Should panic
/// ```

/// ```compile_fail
/// // Should fail to compile
/// ```
```

**Running tests**:

```
cargo test                    # All tests
cargo test test_name          # Specific test
cargo test --lib              # Unit tests only
cargo test --doc              # Doc tests only
cargo test -- --nocapture     # Show println!
cargo test -- --test-threads=1  # Single-threaded
```

## Documentation with rustdoc

**Doc comments**:

```rust
/// Outer doc comment for following item.
/// Supports **Markdown**.
pub fn documented() {}

//! Inner doc comment for containing item.
```

**Standard sections**:

```rust
/// Brief summary.
///
/// Detailed description.
///
/// # Examples
///
/// ```
/// let result = function(42);
/// ```
///
/// # Panics
///
/// Panics if input is negative.
///
/// # Errors
///
/// Returns `Err` if operation fails.
///
/// # Safety
///
/// Unsafe because it dereferences raw pointer.
pub fn function(x: i32) -> i32 { x }
```

**Intra-doc links**:

```rust
/// See [`OtherType`] for more.
/// Check [`other_module::function`].
/// Or use shorthand: [other_function]
```

**Generate docs**:

```
cargo doc                  # Generate
cargo doc --open           # Generate and open
cargo doc --no-deps        # Skip dependencies
```

**Attributes**:

```rust
#[doc(hidden)]          // Hide from docs
#[doc(inline)]          // Inline re-exported
#[doc(alias = "alt")]   // Search alias
```

**Best practices**: Document all public items, include runnable examples, use standard sections, link related items.

## Build scripts (build.rs)

**When to use**: Compile C libraries, generate code, detect features, find system libraries, set configuration.

**Basic structure**:

```rust
// build.rs
fn main() {
    println!("cargo::rerun-if-changed=build.rs");
}
```

**Key instructions**:

```rust
// Change detection
println!("cargo::rerun-if-changed=src/template.txt");

// Linking
println!("cargo::rustc-link-lib=static=mylib");
println!("cargo::rustc-link-search=native=/usr/local/lib");

// Compilation flags
println!("cargo::rustc-cfg=feature_x");
println!("cargo::rustc-env=VERSION={}", env!("CARGO_PKG_VERSION"));
```

**Environment variables**: `OUT_DIR`, `TARGET`, `HOST`, `CARGO_MANIFEST_DIR`, `CARGO_PKG_VERSION`, `PROFILE`.

**Code generation example**:

```rust
use std::env;
use std::fs;
use std::path::Path;

fn main() {
    let out_dir = env::var_os("OUT_DIR").unwrap();
    let dest = Path::new(&out_dir).join("generated.rs");

    fs::write(&dest, "pub const GEN: &str = \"Hello!\";").unwrap();

    println!("cargo::rerun-if-changed=build.rs");
}

// In src/lib.rs:
include!(concat!(env!("OUT_DIR"), "/generated.rs"));
```

**Build dependencies**:

```toml
[build-dependencies]
cc = "1.0"
bindgen = "0.65"
```

## Publishing to crates.io

**Prerequisites**: 1. Create account at crates.io (GitHub login) 2. Generate API token 3. Login: `cargo login <token>`

**Required metadata**:

```toml
[package]
name = "unique-crate-name"
version = "0.1.0"
edition = "2021"
description = "Short description"
license = "MIT OR Apache-2.0"
documentation = "https://docs.rs/my-crate"
repository = "https://github.com/user/my-crate"
readme = "README.md"
keywords = ["key1", "key2"]  # Max 5
categories = ["category1"]   # Max 5
```

**Publishing process**:

```bash
cargo publish --dry-run    # Test
cargo package --list       # See files
cargo publish              # Publish
```

**Version management**:

```
# Update version in Cargo.toml
cargo publish              # Publish new version
```

**Yanking** (prevent new dependencies):

```
cargo yank --version 1.0.1
cargo yank --version 1.0.1 --undo
```

**Managing owners**:

```
cargo owner --add github-handle
cargo owner --remove github-handle
cargo owner --list
```

**Best practices**: - Comprehensive documentation - Meaningful README - Examples in docs - Test with --all-features and --no-default-features - Update CHANGELOG - Create git tag for version - Verify docs build on docs.rs

**Files to include/exclude**:

```
[package]
include = ["src/**/*", "Cargo.toml", "LICENSE*", "README*"]
exclude = ["target", ".github", "*.large-file"]
```

## REFERENCES FOR FURTHER LEARNING

All information sourced from authoritative Rust documentation:

- **The Rust Programming Language (The Book)**: doc.rust-lang.org/book/
- **Rust by Example**: doc.rust-lang.org/rust-by-example/
- **The Rust Reference**: doc.rust-lang.org/reference/
- **The Rustonomicon**: doc.rust-lang.org/nomicon/ (unsafe Rust)
- **Rust Compiler Development Guide**: rustc-dev-guide.rust-lang.org
- **The Cargo Book**: doc.rust-lang.org/cargo/
- **Standard Library Documentation**: doc.rust-lang.org/std/
- **Rust RFC Book**: rust-lang.github.io/rfcs/

- **This Week in Rust**: Weekly newsletter at this-week-in-rust.org
- **Official Rust Blog**: blog.rust-lang.org

These five comprehensive documents provide deep technical understanding of Rust from toolchain architecture through language constructs, focusing on how things work under the hood rather than just surface-level usage. Each section includes practical examples, best practices, and the reasoning behind design decisions.