Introduction

Most Rust books teach you the language: ownership, traits, the borrow checker, how to structure a crate. This one assumes you've already been through that. It picks up where those books stop and asks a different question: what does `rustc` actually *do* with the code you write?

That question turns out to have a lot of interesting answers. A generic function isn't one function — it's N functions, one per concrete type, each compiled and optimized separately. `Option<&T>` isn't a tagged union with wasted space — the compiler notices that references are never null and reuses the invalid bit pattern, so the `Option` costs nothing. An `async fn` isn't sugar over a callback — it's a state machine, generated by the compiler, with a field for every value that has to survive across an `.await` point. None of this is visible in the source code. All of it explains why the code behaves, compiles, and performs the way it does.

Who this book is for

This book is for developers who already write Rust — comfortably, not necessarily expertly — and have started asking *why*. Maybe you've stared at a confusing borrow checker error and wondered what the compiler is actually tracking. Maybe you've run `cargo bloat` and been surprised by how much code a handful of generic functions produced. Maybe you've wondered why `Box<T>` is one pointer but `Box<dyn Trait>` is two, or why `Rc` doesn't need a mutex but `Arc` does, or what a `Waker` actually wakes.

You don't need to be a compiler engineer to read this book, and you won't come out of it one. But you should already be past the point of learning Rust's syntax and rules by example — the chapters here spend their time on mechanism, not tutorial. If a code sample uses lifetimes, closures, or `unsafe` without explanation, that's because explaining those is a different book's job.

Concretely, you'll get the most out of this if you:

- Have written and shipped real Rust code — a library, a service, a CLI, anything past toy examples
- Understand ownership and borrowing well enough to fix a borrow checker error without guessing
- Are comfortable reading `unsafe` blocks, even if you don't write them often
- Have some curiosity about compilers, systems programming, or performance — this book rewards that curiosity directly

What this book is

Each chapter takes one piece of the compiler's behavior and goes deep: what representation it uses internally, what tool output (`--emit=mir` , `--emit=llvm-ir` , `cargo expand` , `cargo bloat`) actually shows you, and what tradeoffs the compiler is making on your behalf. The book follows the compiler's own pipeline and then radiates out into the corners that pipeline touches:

- **Compilation itself** — parsing, monomorphization, const generics, trait resolution, lang items, and both macro systems (Part I)
- **The borrow checker** — NLL, lifetime inference, variance, and where `unsafe` takes over what the compiler can no longer check (Part II)
- **Memory layout** — `repr` variants, niche optimization, smart pointers, and the allocator (Part III)
- **Dynamic dispatch** — closures as compiler-generated structs, vtables, and trait objects (Part IV)
- **Async/await** — the state machine transform, executors, and wakers (Part V)
- **Ownership's edge cases** — drop order, move semantics, and interior mutability (Part VI)
- **LLVM and codegen** — IR, optimization passes, and SIMD (Part VII)
- **Concurrency** — Send/Sync, atomics and memory ordering, and Rayon (Part VIII)

By the end, the goal isn't just trivia about `rustc` internals. It's a mental model precise enough that borrow checker errors stop feeling arbitrary, performance

regressions stop feeling mysterious, and reaching for `unsafe`, `Pin`, or a `#[repr]` attribute feels like a deliberate choice rather than a leap of faith.

Let's start at the beginning: what actually happens between typing `cargo build` and getting a binary.

Chapter 1: From Source to Binary — The Stages of rustc

Every time you run `cargo build`, a staggering amount of machinery fires up. Your Rust source code passes through at least seven distinct representations before it becomes the machine code your CPU actually executes. Most languages have a similarly layered compilation pipeline, but Rust's is unusual in how much *work* happens at each stage: name resolution, type inference, borrow checking, monomorphization, MIR optimization, LLVM optimization, and finally code generation. Understanding these stages doesn't just satisfy curiosity — it changes how you write code, how you debug compilation errors, and how you reason about the performance of both your builds and your binaries.

Let's trace a piece of Rust code through the entire pipeline.

The Pipeline at a Glance

Here's the full sequence, start to finish:

```
Source Code → Tokens → AST → HIR → THIR → MIR → LLVM IR → Machine Code
```

Each arrow represents a transformation that discards some information and adds other information. By the time LLVM IR reaches the code generator, almost nothing recognizable from your original source remains. Let's take a simple function and follow it through.

```rust
fn sum_positive(values: &[i32]) -> i32 {
    values.iter()
        .filter(|&&x| x > 0)
        .sum()
}
```

This is a small function, but it exercises several interesting parts of the pipeline: closures, trait method resolution, generics, iterator adaptors, and the borrow checker. We'll see how each stage handles it.

Stage 1: Lexing — Source Code to Tokens

The lexer (also called the tokenizer) is the simplest stage. It reads raw bytes and produces a stream of tokens: keywords, identifiers, literals, punctuation, and whitespace (which is mostly discarded). Rust's lexer lives in `rustc_lexer` and is deliberately simple — it doesn't even track nesting or handle string interpolation at this stage.

For our function, the token stream looks roughly like:

```
FN, IDENT("sum_positive"), LPAREN, IDENT("values"), COLON,
AMP, LBRACKET, IDENT("i32"), RBRACKET, RPAREN, RARROW,
IDENT("i32"), LBRACE, IDENT("values"), DOT, IDENT("iter"),
LPAREN, RPAREN, DOT, IDENT("filter"), LPAREN, PIPE, ...
```

There's not much to say about lexing — it's fast, it's straightforward, and it fails only on truly malformed input like unclosed string literals or invalid Unicode. The interesting bit is that Rust's lexer operates at the crate level and processes the entire file in one pass. Comments become tokens too (for doc comments that need to survive into later stages), but regular comments are stripped here.

One detail worth knowing: Rust's tokenization is context-free. Unlike C++, where the meaning of `>>` depends on whether you're inside a template parameter list, Rust's lexer always produces the same tokens regardless of context. This is a deliberate design choice that keeps the lexer simple and fast.

Stage 2: Parsing — Tokens to AST

The parser transforms the token stream into an Abstract Syntax Tree (AST). Rust uses a hand-written recursive descent parser (not a generated one), which lives in `rustc_parse`. The AST closely mirrors the surface syntax of Rust — it preserves things like parenthesization, expression nesting, and syntactic sugar.

At the AST level, our function is represented as a `FnDecl` node with:

- A name: `sum_positive`
- A parameter list containing one parameter with a reference-to-slice type
- A return type annotation
- A body expression: a method call chain

The AST still contains some things that will be resolved later. The `.iter()` call, for instance, is just an unresolved method name at this point. The parser doesn't know whether `iter` is a method on slices, a trait method, or a free function — that's name resolution's job.

Macros are expanded during this phase. When the parser encounters a macro invocation, it expands it and re-parses the result. This is why macro errors sometimes produce strange spans — the source location machinery has to track both the original invocation site and the expansion site. Procedural macros run as separate processes at this stage, receiving `TokenStream` and returning `TokenStream`.

You can see the AST using the unstable flag:

```
cargo rustc -- -Zunpretty=ast
```

This outputs something close to the original source but with all macros expanded and some sugar removed.

Stage 3: HIR — The High-level Intermediate Representation

The AST is lowered to HIR ("High-level IR"), which is where Rust starts to diverge significantly from the source syntax. HIR lives in `rustc_hir` and is the representation used for type checking, trait resolution, and borrow checking preparation.

The lowering from AST to HIR does several things:

- **Desugars syntax**: `for` loops become `loop` + `match` on `Iterator::next()`. `if let` becomes `match`. `?` becomes a `match` with early return. `async` blocks become generator state machines.
- **Assigns IDs**: Every expression, pattern, and definition gets a `HirId` that's stable across incremental compilations.
- **Resolves names**: Imports are resolved, paths are fully qualified. That bare `iter()` call now knows it refers to `<[i32]>::iter`.
- **Lifetime elision**: Elided lifetimes are inserted. Our `&[i32]` gets an anonymous lifetime parameter.

You can inspect HIR with:

```
cargo rustc -- -Zunpretty=hir
```

Our function in HIR will have the `for`-like iterator chain desugared into explicit method calls with fully resolved paths. The closure `|&&x| x > 0` still exists as a closure expression, but its capture mode has been analyzed.

An important detail: HIR is where **type inference** runs. The `rustc_typeck` (now `rustc_hir_typeck`) crate walks the HIR, infers types, checks trait bounds, and resolves method calls. When you see a type error, it's almost always generated from this stage. The error messages reference source spans, but the analysis happens on HIR.

Stage 4: THIR — The Typed HIR

Before MIR, there's a less-discussed intermediate step: THIR (Typed High-level IR). This is the HIR annotated with full type information — every expression has a concrete type, every method call is resolved to a specific impl. THIR exists mainly as a stepping stone to MIR; it simplifies the lowering by separating the concerns of "attach types to everything" from "restructure into control flow graphs."

THIR is also where exhaustiveness checking for `match` expressions happens. The pattern analysis algorithm runs on THIR because it needs full type information to know which patterns are possible.

You can view THIR (on nightly) with:

```
cargo rustc -- -Zunpretty=thir-tree
```

Stage 5: MIR — Where Rust Gets Serious

MIR (Mid-level IR) is arguably the most important representation in rustc. It's where the borrow checker runs, where most Rust-specific optimizations happen, and where the code gets restructured into a form suitable for code generation. MIR is a control-flow graph (CFG) of basic blocks, similar in spirit to SSA form but using "places" (lvalues) instead of SSA values.

Let's look at what happens to our function. You can emit MIR with:

```
cargo rustc -- --emit=mir
```

Or for a more readable format:

```
cargo rustc -- -Zunpretty=mir
```

The MIR for `sum_positive` will look something like this (simplified):

```
fn sum_positive(_1: &[i32]) -> i32 {
    let mut _0: i32;
    let mut _2: std::slice::Iter<'_, i32>;
    let mut _3: std::iter::Filter<std::slice::Iter<'_, i32>,
[closure]>;

    bb0: {
        _2 = <[i32]>::iter(move _1);
        _3 = <std::slice::Iter<i32> as Iterator>::filter(move _2,
const closure);
        _0 = <Filter<..> as Iterator>::sum(move _3);
        return;
    }
}
```

Several things have happened here:

1. **Method calls become function calls**: `values.iter()` becomes `<[i32]>::iter(move _1)`. The receiver is just the first argument.
2. **Closures become constants**: The closure `|&&x| x > 0` is promoted to a separate MIR function and referenced as a constant. Its captures (none, in this case) are explicit.
3. **Generics are not yet monomorphized**: The MIR still references generic types like `Filter<Iter<'_, i32>, [closure]>`. Monomorphization happens later.
4. **Control flow is explicit**: There are no expressions with hidden control flow. Everything is broken into basic blocks (`bb0`, `bb1`, ...) with explicit jumps and returns.
5. **Temporaries are named**: Every intermediate value gets a numbered local (`_0`, `_1`, `_2`, ...).

What the Borrow Checker Sees

The borrow checker operates on MIR, not on the source code or HIR. It constructs a control-flow graph, computes liveness information for every borrow, and checks that no borrow outlives its referent and that no mutable borrow coexists with another borrow.

This is why Non-Lexical Lifetimes (NLL) work: the borrow checker looks at where borrows are actually *used* in the CFG, not at lexical scopes. A borrow that's last used on line 5 can coexist with a mutable borrow starting on line 6, even within the same scope.

MIR Optimizations

Before MIR is handed off to code generation, rustc runs its own optimization passes. These include:

- **Constant propagation**: If a value is known at compile time, substitute it.
- **Dead code elimination**: Remove unreachable basic blocks.
- **Simplify branches**: Replace `if true { ... }` with unconditional jumps.
- **Inline**: MIR-level inlining, controlled by `-Zmir-opt-level`.
- **Copy propagation**: Eliminate redundant moves.

The MIR optimization level can be controlled with `-Zmir-opt-level=0|1|2|3|4`. The default depends on the build profile. These optimizations matter because they can reduce the amount of work LLVM has to do, which speeds up compilation.

Stage 6: LLVM IR — Entering the Backend

After MIR optimization, rustc lowers MIR to LLVM IR. This is where monomorphization happens — every generic function is instantiated for each concrete type it's called with. Our `sum_positive` function doesn't have its own type parameters, but the iterator adaptors it uses do: `Filter` is generic over the iterator type and the closure type, and `sum` is generic over the output type. Each of these gets a concrete instantiation.

LLVM IR is a typed, SSA-based representation that looks like this:

```
cargo rustc -- --emit=llvm-ir
# Output goes to target/debug/deps/your_crate-hash.ll
```

For a simpler example, let's see what a basic function looks like in LLVM IR:

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

```
define i32 @_ZN10your_crate3add17h8f2a3b4c5d6e7f8aE(i32 %a, i32 %b)
unnamed_addr #0 {
start:
  %result = add i32 %a, %b
  ret i32 %result
}
```

A few things to notice:

- **Name mangling**: The function name includes the crate name, module path, function name, and a hash. Use `rustfilt` to demangle: `echo '_ZN10your_crate3add17h...' | rustfilt`.
- **SSA form**: Every value is assigned exactly once. No mutable variables — new values get new names.
- **Type annotations everywhere**: LLVM IR is explicitly typed. `i32` appears on every value.
- **Rust-specific attributes**: References produce `noalias` (for `&mut T`), `nonnull`, and `dereferenceable` attributes that LLVM uses for optimization.

For our `sum_positive` function, the LLVM IR will be substantially longer because monomorphization has inlined the iterator machinery. After LLVM's optimization passes, though, the entire iterator chain typically collapses into a simple loop — this is the "zero-cost abstractions" promise in action.

You can see the LLVM IR at different optimization levels:

```
# Unoptimized — verbose, close to MIR structure
cargo rustc -- --emit=llvm-ir

# Optimized — after LLVM's passes
cargo rustc --release -- --emit=llvm-ir
```

Stage 7: Machine Code — The Final Output

LLVM takes the IR, runs its optimization pipeline (which includes dozens of passes: inlining, loop unrolling, vectorization, dead code elimination, register allocation, instruction scheduling), and produces machine code.

You can inspect the assembly:

```
cargo rustc --release -- --emit=asm
# Output goes to target/release/deps/your_crate-hash.s
```

For our `add` function on x86-64:

```
your_crate::add:
    lea eax, [rdi + rsi]
    ret
```

That's it. One `lea` instruction (used here as a two-operand add) and a `ret`. For `sum_positive`, the optimized assembly will be a loop with a conditional add — the entire iterator adapter chain, the closure, the filter, the sum — all compiled down to a tight loop that a C programmer would recognize.

This is worth emphasizing: the assembly output of idiomatic Rust iterator chains is typically identical to hand-written imperative loops. You can verify this yourself on Compiler Explorer (godbolt.org) by comparing:

```rust
// Iterator style
fn sum_positive_iter(values: &[i32]) -> i32 {
    values.iter().filter(|&&x| x > 0).sum()
}

// Imperative style
fn sum_positive_loop(values: &[i32]) -> i32 {
    let mut total = 0;
    for &v in values {
        if v > 0 {
            total += v;
        }
    }
    total
}
```

At `-C opt-level=3`, these produce identical machine code.

Incremental Compilation: The Query System

Compiling from scratch every time you change a line would be unbearable for large projects. Rustc's incremental compilation system avoids redundant work by caching the results of individual computations and recomputing only what's necessary.

The Query System

Internally, rustc is structured as a database of **queries**. Almost every computation is expressed as a query: "What is the type of this expression?", "What are the trait impls in scope here?", "What is the MIR for this function?". Queries take keys (like a `DefId` identifying a function) and produce values (like a MIR body).

Queries have several crucial properties:

- **Deterministic**: Given the same input, a query always produces the same output.

- **Memoized**: Once computed, results are cached for the duration of the compilation.
- **Tracked**: The system records which queries depend on which other queries, forming a dependency graph.

This is not a traditional Makefile-style dependency graph based on files. It's much more fine-grained. Changing the body of function `foo` might invalidate the MIR for `foo` and anything that inlines `foo`, but it won't invalidate the type-checking results for function `bar` that doesn't call `foo`.

The Red-Green Algorithm

When you modify a source file and rebuild, rustc uses the **red-green algorithm** to determine what to recompute:

1. **Load the previous session's dependency graph** from the incremental compilation cache (stored in `target/debug/incremental/`).
2. **Mark changed inputs as red.** If a source file changed, every query that directly depends on its content is marked red (dirty).
3. **Try to mark nodes green.** For each node that depends on a red node, rustc doesn't immediately recompute it. Instead, it checks: did the *result* of the red dependency actually change? If the result is the same (same hash), the dependent node can be marked green (clean) without recomputation.
4. **Recompute only what's truly invalidated.** Only nodes that can't be proven green are recomputed.

This is extremely effective in practice. Consider changing a comment inside a function body. The lexer/parser output changes (the AST includes doc comments), but after lowering to HIR, the HIR may be identical — the comment was just a regular comment with no semantic effect. The fingerprint of the HIR won't change, so nothing downstream of HIR needs recompilation.

Dependency Graphs in Practice

You can visualize the dependency graph with:

```
cargo rustc -- -Zdump-dep-graph
# Produces a .dot file you can render with graphviz
```

For a real project, these graphs are enormous. But they reveal the structure of the compiler's computation: type checking depends on name resolution, MIR construction depends on type checking, LLVM IR generation depends on MIR, and so on.

The Fingerprinting System

The red-green algorithm relies on **fingerprints** — 128-bit hashes of query results. When rustc computes a query, it hashes the result and stores the fingerprint alongside the cached value. On the next compilation, when it needs to check if a dependency "really changed," it recomputes the query and compares fingerprints.

This is surprisingly subtle. Consider:

```
fn foo() -> i32 {
    // Changed from: 1 + 1
    2
}
```

You changed the implementation from `1 + 1` to `2`. The AST changed. The HIR changed (different expressions). But after constant folding, the MIR might be identical — both produce `_0 = const 2_i32; return;`. If the MIR fingerprint is the same, nothing that depends on the MIR needs recompilation.

Fingerprints use `SipHash-128`, chosen for a good balance of speed and collision resistance. A collision would cause a miscompilation (the compiler would skip recomputing something that actually changed), so the hash needs

to be robust. In practice, 128-bit SipHash has no known collisions for this use case.

Fingerprints are computed incrementally, too. The hash of a large struct is built by hashing each field; if most fields haven't changed, the hash computation can short-circuit in some cases.

You can see fingerprinting in action with:

```
RUSTC_LOG=rustc_incremental=debug cargo build 2>&1 | head -100
```

This will show you which queries were computed, which were loaded from cache, and which had matching fingerprints.

Parallel Compilation

Historically, rustc was single-threaded within a single crate. (Cargo parallelizes across crates, but each `rustc` invocation was sequential.) This has changed. Modern rustc uses **rayon** for intra-crate parallelism, and this capability is maturing rapidly.

What Gets Parallelized

The main opportunity for parallelism is **codegen**. After MIR is complete, the crate is split into **codegen units** (CGUs), and each CGU can be lowered to LLVM IR and optimized independently. This is the `-C codegen-units=N` flag — the default is 16 in debug mode and 1 in release mode (with LTO).

With `-Zthreads=N` (nightly, but increasingly available on stable via codegen unit parallelism), rustc can:

- **Run LLVM optimization passes in parallel** across codegen units.
- **Perform some type checking in parallel** (work-in-progress, using rayon's work-stealing scheduler).

- **Run macro expansion in parallel** (limited, due to ordering constraints).

What *can't* easily be parallelized:

- **Name resolution**: Depends on the order of imports and `use` declarations.
- **Borrow checking**: Each function is independent, so this is embarrassingly parallel in principle, but the implementation is still being parallelized.
- **Trait resolution**: Global coherence checking inherently requires a global view.

Codegen Units in Practice

The `-C codegen-units` flag has profound effects:

```
# Fewer codegen units = better optimization (LLVM sees more code at
once)
cargo rustc --release -- -C codegen-units=1

# More codegen units = faster compilation (more parallelism)
cargo rustc --release -- -C codegen-units=16
```

With `codegen-units=1`, LLVM can inline across the entire crate and eliminate more dead code. With `codegen-units=16`, compilation is faster but the output may be slightly less optimized because LLVM can't see across CGU boundaries (unless LTO is enabled).

The default in release mode changed to `codegen-units=1` because the optimization benefit usually outweighs the compilation speed cost for release builds. Debug builds default to 16 because optimization matters less and compile speed matters more.

Build Profiles: What the Flags Actually Do

Cargo's `[profile.dev]` and `[profile.release]` sections control a set of flags that affect different stages of the pipeline. Let's look at what each flag actually does.

`opt-level`

```
[profile.release]
opt-level = 3  # 0, 1, 2, 3, "s", "z"
```

This maps to LLVM's optimization pipeline:

- `0` : No LLVM optimization. MIR is lowered to LLVM IR and compiled directly. The output is faithful to the MIR structure — no inlining, no dead code elimination, no constant folding (beyond what MIR optimization already did). Compile time is fast. Runtime is slow.
- `1` : Basic optimizations. LLVM runs a reduced set of passes: some inlining, simple constant propagation, basic dead code elimination. A good middle ground for development builds where you need reasonable performance.
- `2` : Full optimization. The standard LLVM optimization pipeline runs: aggressive inlining, loop transformations (unrolling, vectorization, interchange), interprocedural optimization within the codegen unit. This is the default for release builds.
- `3` : Aggressive optimization. Same as `2` but with more aggressive inlining thresholds and loop unrolling. Can sometimes produce slower code due to increased instruction cache pressure — profile before assuming `3` is better than `2` .
- `"s"` : Optimize for size. Uses the `2` -level pipeline but with inlining thresholds biased toward smaller code.
- `"z"` : Optimize aggressively for size. Even more aggressive size reduction, at the cost of runtime performance.

One crucial detail: `opt-level` only controls LLVM. MIR optimizations run at their own level (controlled by `-Zmir-opt-level` on nightly). Even at `opt-`

`level=0` , rustc still runs some MIR optimizations because they're needed for correctness (like removing dead storage for moved values).

lto

```
[profile.release]
lto = true  # false, true, "thin", "fat"
```

Link-Time Optimization allows LLVM to optimize across codegen unit and crate boundaries. Without LTO, each codegen unit (and each crate) is optimized in isolation, and cross-boundary function calls can't be inlined.

- `false` (default for dev): No cross-unit optimization. Each CGU is compiled independently. Fast compilation.
- `"thin"` : ThinLTO. Each CGU is compiled with summary information about its exported/imported symbols. A lightweight cross-module optimization pass runs at link time, importing and inlining hot cross-boundary calls. Good tradeoff between optimization and compile time.
- `true` or `"fat"` : Full LTO. All CGUs (and optionally all dependency crates) are merged into a single LLVM module and optimized together. This gives LLVM maximum visibility for optimization but is significantly slower to compile. It also defeats the parallelism benefit of multiple codegen units — everything must be processed sequentially in the final pass.

The compile-time cost of LTO is substantial. For a medium-sized project, full LTO can increase release build time by 2-5x. ThinLTO typically adds 20-50% overhead. The runtime benefit varies: for CPU-bound applications with hot cross-crate calls, LTO can yield 5-20% improvements. For I/O-bound applications, the difference is negligible.

codegen-units

```
[profile.release]
codegen-units = 1  # default: 1 for release, 16 for dev
```

This controls how rustc partitions MIR into separate LLVM modules. Each codegen unit becomes an independent LLVM compilation job. The partitioning algorithm tries to produce units of roughly equal size, keeping closely related functions together.

The interaction between `codegen-units` and `lto` is important:

codegen-units	lto	Effect
1	false	Single unit, full optimization visibility, no parallelism
16	false	Parallel compilation, limited cross-unit optimization
16	thin	Parallel compilation, ThinLTO recovers most cross-unit optimization
16	fat	Parallel initial compilation, but final LTO pass merges everything

For maximum binary quality: `codegen-units = 1, lto = true`. For maximum compile speed: `codegen-units = 16, lto = false`.

debug

```
[profile.release]
debug = 2  # 0, 1, 2, true, false, "line-tables-only"
```

This controls DWARF debug info generation:

- `0` / `false`: No debug info. Smaller binaries, slightly faster compilation.
- `1` / `"line-tables-only"`: Line tables only. You get file/line information in backtraces and profilers, but no variable inspection in debuggers.
- `2` / `true`: Full debug info. Variable names, types, scopes — everything a debugger needs.

Debug info is appended to the binary (or stored in separate `.dSYM` / `.dwp` files). It doesn't affect runtime performance — it only increases binary size and

slightly increases compile time.

A common trick for production profiling:

```
[profile.release]
debug = 1  # Line tables for perf/profiler, minimal size overhead
```

strip

```
[profile.release]
strip = "symbols"  # "none", "debuginfo", "symbols"
```

This controls post-compilation stripping:

- `"none"` : Keep everything.
- `"debuginfo"` : Strip debug info but keep symbol names (useful for stack traces).
- `"symbols"` : Strip everything. Smallest binary, but backtraces show only addresses.

panic

```
[profile.release]
panic = "abort"  # "unwind", "abort"
```

This affects code generation at the MIR level:

- `"unwind"` (default): Generates landing pads for stack unwinding. Each function call that might panic gets associated cleanup code. This enables `catch_unwind` and ensures destructors run during panics.
- `"abort"` : No unwinding infrastructure. A panic immediately aborts the process. This produces smaller binaries (no landing pads, no unwinding tables) and slightly faster code (the optimizer can assume functions don't unwind). The downside: destructors don't run on panic, and you can't catch panics.

The size difference can be significant. For a medium Rust binary, `panic = "abort"` can reduce binary size by 10-20% and reduce compile time slightly (less code to generate and optimize).

Putting It All Together: Inspecting the Pipeline

Here's a practical workflow for understanding what the compiler does with your code:

```
# Step 1: Look at the HIR (desugared source)
cargo rustc -- -Zunpretty=hir

# Step 2: Look at the MIR (control flow graph, after borrow
checking)
cargo rustc -- -Zunpretty=mir

# Step 3: Look at the LLVM IR (after monomorphization, before LLVM
optimizes)
cargo rustc -- --emit=llvm-ir

# Step 4: Look at the optimized LLVM IR
cargo rustc --release -- --emit=llvm-ir

# Step 5: Look at the final assembly
cargo rustc --release -- --emit=asm

# Step 6: See how long each stage takes
cargo rustc --release -- -Ztime-passes 2>&1 | head -50
```

The `-Ztime-passes` flag is particularly illuminating. It shows you how much time is spent in each compilation phase. For a typical project, you'll see something like:

```
time:   0.012; rss:   50MB  parsing
time:   0.003; rss:   55MB  name resolution
time:   0.025; rss:   70MB  type checking
time:   0.008; rss:   75MB  borrow checking
time:   0.015; rss:   80MB  MIR optimization
time:   0.180; rss:  120MB  LLVM passes
time:   0.045; rss:  130MB  linking
```

Notice that LLVM passes dominate. This is typical — Rust's frontend (parsing through MIR optimization) is usually much faster than the backend (LLVM optimization and code generation). This is why flags like `opt-level`, `lto`, and `codegen-units` have such a large impact on compile time: they all affect the LLVM stage.

For day-to-day development, this suggests a strategy: use `opt-level=0` or `opt-level=1` with multiple codegen units and no LTO. Reserve `opt-level=2` or `3` with LTO for release builds and benchmarking. And use incremental compilation (on by default for dev builds) to avoid rerunning even the frontend for unchanged code.

The Cranelift backend (`-Zcodegen-backend=cranelift`) is an experimental alternative that replaces LLVM with a faster code generator. It produces worse-optimized code but compiles significantly faster — ideal for debug builds where you don't care about runtime performance.

Summary

The rustc pipeline is deep: seven representations, each serving a specific purpose. The AST captures syntax. HIR enables name resolution and type checking. MIR enables borrow checking and Rust-specific optimization. LLVM IR enables platform-independent optimization. Machine code runs on your CPU.

Understanding this pipeline pays off in concrete ways:

- When a borrow checker error confuses you, remember it operates on MIR's control-flow graph, not on your source code's lexical structure.

- When compilation is slow, check `-Ztime-passes` and tune `codegen-units` and `opt-level` for the bottleneck stage.
- When your release binary is too large, consider `lto`, `strip`, and `panic = "abort"` — each affects a different part of the pipeline.
- When iterator chains seem "too abstract to be fast," look at `--emit=asm` and verify that LLVM collapses them into the same tight loop as a hand-written version.

The compiler is doing an enormous amount of work on your behalf. Now you know where to look when you want to see it.

Chapter 2: Monomorphization — The Cost of Generics

Rust's generics promise zero-cost abstractions: you write `fn sort<T: Ord>(slice: &mut [T])` once, and it runs as fast as if you had hand-written a sort function for every concrete type. That promise is real. But "zero-cost" refers to *runtime* cost. At compile time and in your final binary, generics carry a very real price tag. The mechanism that pays that price is called **monomorphization**, and understanding it is essential to understanding why Rust binaries are the size they are, why compile times are what they are, and when you should reach for `dyn Trait` instead.

From Generic to Concrete: How Monomorphization Works

Consider this function:

```
use std::fmt::Display;

fn log_value<T: Display>(label: &str, value: T) {
    println!("[LOG] {}: {}", label, value);
}
```

When you call this function with different types:

```
log_value("count", 42u32);
log_value("name", "Alice");
log_value("ratio", 3.14f64);
```

The compiler does not generate a single function that operates on "some `Display` type" at runtime. Instead, it generates three completely independent functions, each with the type parameter erased and replaced with the concrete type:

```rust
// What the compiler effectively produces:
fn log_value_u32(label: &str, value: u32) {
    println!("[LOG] {}: {}", label, value);
}

fn log_value_str(label: &str, value: &str) {
    println!("[LOG] {}: {}", label, value);
}

fn log_value_f64(label: &str, value: f64) {
    println!("[LOG] {}: {}", label, value);
}
```

Each call site is rewritten to call the appropriate concrete version. The trait bound `T: Display` is gone — it was checked at type-checking time and has no runtime representation. The `Display::fmt` call inside each monomorphized copy is a direct, static function call, not a virtual dispatch through a vtable.

When It Happens

Monomorphization does not happen during type checking or even during MIR optimization. It happens at **codegen time**, when rustc hands MIR to the LLVM backend (or to Cranelift, if you are using it). The process works roughly like this:

1. **Type checking** verifies that `T: Display` is satisfied for each call site. The generic function is checked exactly once, against the trait bounds.
2. **MIR generation** produces a single, generic MIR body for `log_value`. This MIR still has `T` as a parameter.
3. **Monomorphization collection** walks the call graph starting from `main`, discovers every concrete substitution of every generic function that is reachable, and builds a list of "mono items" — concrete `(function, substitution)` pairs that need codegen.
4. **Codegen** takes each mono item and lowers its MIR to LLVM IR with all type parameters replaced by concrete types. LLVM then optimizes and emits machine code for each one independently.

You can see the mono items yourself. Pass `--emit=llvm-ir` and look at the mangled function names:

```
; Function Attrs: nonlazybind uwtable
define internal void @_ZN7example9log_value17h3a8f2e1b4c5d6e7fE(
    ptr %label.0, i64 %label.1, i32 %value) {
    ; ... body for log_value::<u32> ...
}
```

That `h3a8f2e1b4c5d6e7f` suffix is a hash of the concrete substitution. Each monomorphized instance gets its own symbol.

The Full Picture: What Gets Duplicated

It is not just your function body. Every function that your generic function calls, if it is also generic over `T`, also gets monomorphized. Consider:

```
fn process<T: Display + Clone>(items: &[T]) {
    for item in items {
        let copy = item.clone();      // Clone::clone::<T>
        log_value("item", &copy);     // log_value::<&T>
        println!("{}", copy);         // <T as Display>::fmt, plus
fmt machinery
    }
}
```

Calling `process::<u32>` does not just produce one monomorphized function. It triggers a cascade: `process::<u32>`, `Clone::clone::<u32>`, `log_value::<&u32>`, and all the formatting machinery for `u32`. Calling `process::<String>` produces an entirely parallel cascade. The duplication is transitive and multiplicative.

Monomorphization Bloat: Measuring the Damage

The term “monomorphization bloat” refers to the binary size increase caused by duplicating code for every type substitution. It is not a theoretical concern. Real-world Rust projects routinely see it.

The `Vec` Explosion

Consider the standard library's `Vec<T>`. The methods `push`, `pop`, `len`, `get`, `iter`, `sort`, `dedup`, `retain`, `extend`, `drain` — and dozens more — are all generic over `T`. If your program uses `Vec<u8>`, `Vec<u16>`, `Vec<u32>`, `Vec<u64>`, `Vec<String>`, and `Vec<PathBuf>`, each of those ~100 methods is potentially monomorphized six times. That is up to 600 function bodies in your binary, many of which are doing essentially the same thing (push an element, grow the buffer, copy some bytes) but with different element sizes and alignments.

Some of these are truly different — `Vec<String>::drop` must drop each `String` element, while `Vec<u32>::drop` does not need to drop elements at all. But many operations, like `Vec::reserve` or `Vec::set_len`, do not actually depend on `T` in a meaningful way — they only care about the size and alignment. Yet every instantiation gets its own copy.

Measuring with `cargo-bloat`

Install `cargo-bloat` and point it at your binary:

```
$ cargo bloat --release -n 30

 File  .text    Size      Crate Name
 2.7%   8.5%  27.1KiB   hashbrown
hashbrown::raw::RawTable<T,A>::resize_inner
 1.9%   6.1%  19.4KiB   std       std::io::Write::write_fmt
 1.3%   4.2%  13.2KiB   regex
regex_automata::nfa::thompson::compiler::...
 0.8%   2.5%   8.0KiB   serde     serde::de::MapAccess::next_entry
 0.7%   2.2%   7.0KiB   myapp     myapp::process::<u32>
 0.7%   2.1%   6.8KiB   myapp     myapp::process::<String>
 0.7%   2.1%   6.7KiB   myapp     myapp::process::<PathBuf>
 ...
```

You will see the same function name appearing multiple times with different type suffixes. When a function like `HashMap::insert` shows up ten times, each for a different key-value type combination, you are looking at monomorphization bloat.

Measuring with `cargo-llvm-lines`

While `cargo-bloat` shows you the final binary, `cargo-llvm-lines` shows you how much LLVM IR the compiler generates *before* optimization — a better proxy for the monomorphization load on the compiler:

```
$ cargo llvm-lines --release | head -20

  Lines          Copies        Function name
  -----          ------        -------------
  30740 (100%)   1120 (100%)   (TOTAL)
   1395 (4.5%)     83 (7.4%)   core::ptr::drop_in_place
    998 (3.2%)     15 (1.3%)
alloc::raw_vec::RawVec<T,A>::grow_amortized
    876 (2.9%)     12 (1.1%)   hashbrown::raw::RawTable<T,A>::find
    684 (2.2%)     32 (2.9%)   core::fmt::Formatter::pad
```

The "Copies" column is the key metric. If `drop_in_place` has 83 copies, that means 83 different types in your program required their own drop glue. Each copy is a separate function body that LLVM must optimize independently.

Reading Demangled Symbols

To see exactly which monomorphizations exist in your binary, dump the symbol table and demangle:

```
$ nm -C target/release/myapp | grep 'log_value'
00000000000a1234 t myapp::log_value::<u32>
00000000000a1300 t myapp::log_value::<&str>
00000000000a13cc t myapp::log_value::<f64>
00000000000a1498 t myapp::log_value::<alloc::string::String>
```

Each line is a concrete monomorphized instance sitting in your binary's `.text` section, consuming space.

Polymorphization: rustc's Attempt to Fight Back

The Rust compiler team recognized the monomorphization bloat problem and introduced **polymorphization**, available behind the `-Zpolymorphize` flag (still unstable as of early 2026). The idea is straightforward: if a type parameter is not actually *used* in a way that affects code generation, do not monomorphize for it.

Consider this function:

```
fn first_element<T, U>(slice: &[T], _unused: U) -> Option<&T> {
    slice.first()
}
```

The type parameter `U` appears only in the function signature, never in the body. `first_element::<i32, String>` and `first_element::<i32, Vec<u8>>` would generate identical machine code. Polymorphization detects this and reuses a single code generation for all `U` substitutions when `T` is the same.

How It Works

During monomorphization collection, the compiler performs a "used parameter" analysis on each generic function's MIR body. For each type parameter, it asks: does this parameter appear in any type that affects code layout or behavior? Specifically, does it appear in:

- A local variable's type (affecting stack layout)
- A function call's type arguments (triggering further monomorphization)
- An `impl` resolution (selecting which trait method to call)
- A `size_of` or `align_of` call

If a parameter fails all of these checks, it is marked as "unused" and replaced with a placeholder during codegen. Multiple substitutions that differ only in unused parameters collapse into a single mono item.

Current Limitations

Polymorphization has been on nightly behind a flag for several years, and its stabilization keeps getting deferred. There are several reasons:

False positives in "used" analysis. The analysis is conservative in some places and overly aggressive in others. Determining whether a type parameter "matters" is subtle — a parameter might not appear directly in the function body but could affect drop glue, alignment requirements, or trait method resolution in non-obvious ways. Getting this wrong leads to miscompilations.

Interactions with codegen units. Rustc splits code generation into multiple parallel codegen units. Polymorphization interacts with the mono item collection phase in ways that complicate partitioning, because two mono items that were previously distinct might now be identified as the same.

Limited practical benefit so far. The easy wins — truly unused parameters — are relatively rare in practice. The more common case is a function where the parameter *is* used, but the generated code happens to be identical (e.g., two types with the same size and alignment). Detecting that level of equivalence would require comparing generated LLVM IR, which polymorphization does not currently do.

The feature is worth understanding conceptually because it represents the principled fix for monomorphization bloat, but in practice, developers who need to control binary size today reach for a different tool: trait objects.

Trait Objects: The Dynamic Dispatch Escape Hatch

When monomorphization bloat is a problem, you can avoid it entirely by switching from static dispatch to dynamic dispatch with trait objects:

```rust
// Static dispatch: monomorphized for each T
fn log_static<T: Display>(value: &T) {
    println!("{}", value);
}

// Dynamic dispatch: one function body, one copy in the binary
fn log_dynamic(value: &dyn Display) {
    println!("{}", value);
}
```

`log_dynamic` generates exactly one function body, regardless of how many types it is called with. The price is an indirect call through the vtable when invoking `Display::fmt`, which means:

1. The CPU cannot inline the method call.
2. The branch predictor must predict the target of the indirect call.
3. Each call through the vtable follows a pointer to the vtable, reads the function pointer, then calls through it — two pointer chases instead of zero.

Code Size vs Speed: When Dynamic Dispatch Wins

The conventional wisdom is "static dispatch is faster." And in microbenchmarks, it usually is — the overhead of an indirect call is measurable on tight loops. But in real applications, the picture is more nuanced.

Instruction cache pressure. Every monomorphized copy of a function takes space in the instruction cache (L1i on x86 is typically 32 KiB per core). If you have 20 monomorphized copies of a complex function, you are burning 20x the icache space. A single `dyn Trait` version uses 1x the space, leaving more room for other hot code. On workloads with diverse call patterns — like a compiler, a web server handling varied request types, or a game engine with many entity types — icache pressure from monomorphization bloat can make the "fast" static dispatch version *slower* overall.

iTLB misses. Closely related to icache: the instruction TLB maps virtual pages to physical pages for code. More code means more pages means more iTLB misses. This is a second-order effect but measurable on large binaries.

The crossover point. As a rough heuristic: if a generic function is large (many instructions), called with many types, and not in a tight inner loop, `dyn Trait` may perform as well or better. If a function is small, called in a hot loop, and performance-critical, static dispatch wins. The `std` library itself uses this heuristic — `std::io::Write` has an inner implementation that works on `dyn Write` to avoid monomorphizing the formatting and buffering logic for every concrete writer type.

`impl Trait` vs `dyn Trait` in Function Signatures

These two forms look similar but have fundamentally different compilation models:

```
// Caller chooses T, function is monomorphized per T
fn process(input: impl Read) { /* ... */ }

// Function receives a fat pointer, one compiled body
fn process(input: &mut dyn Read) { /* ... */ }
```

`impl Trait` in argument position is syntactic sugar for a generic parameter. `impl Trait` in return position is an opaque type — the compiler knows the concrete type, the caller does not, but it is still monomorphized. Neither form uses dynamic dispatch.

`dyn Trait` always means dynamic dispatch, always means a vtable pointer, always means one function body.

A Practical Pattern: Thin Generic Wrapper, Dynamic Inner Function

The standard library and many well-optimized crates use a pattern where the generic function is a thin wrapper that delegates to a non-generic inner function:

```rust
pub fn serialize<T: Serialize>(value: &T, writer: &mut dyn Write) ->
Result<()> {
    // This outer function is monomorphized per T, but it's tiny
    let serialized = value.serialize()?;  // trait method call,
monomorphized

    // The heavy lifting happens in a non-generic function
    write_bytes_impl(writer, &serialized)
}

// This function exists exactly once in the binary
fn write_bytes_impl(writer: &mut dyn Write, bytes: &[u8]) ->
Result<()> {
    writer.write_all(bytes)?;
    writer.flush()?;
    Ok(())
}
```

The generic part handles type-specific logic (serialization), while the non-generic part handles type-agnostic logic (writing bytes). This pattern minimizes the amount of code that gets duplicated while preserving static dispatch where it matters.

Cross-Crate Monomorphization

Here is a question that seems simple but has deep implications: if I call `Vec::push` in my crate, and `Vec` is defined in `alloc`, where does the monomorphized `Vec::<MyType>::push` get compiled?

The answer: **in your crate**, not in `alloc`. And this has consequences for the entire compilation model.

Why Generic Functions Cannot Be Pre-Compiled

A non-generic function can be compiled to machine code once, placed in a library file (`.rlib`), and linked into any binary that calls it. A generic function cannot, because the compiler does not know what types it will be instantiated

with. `Vec::push` might need to be compiled for `Vec<u8>`, `Vec<String>`, `Vec<MyCustomType>` — types that did not exist when the standard library was compiled.

The solution: when rustc compiles a crate, it serializes the **MIR** (Mid-level Intermediate Representation) of every generic function into the crate's metadata (stored in the `.rlib` file). When a downstream crate calls that generic function with a concrete type, the compiler deserializes the MIR, substitutes the concrete type, and performs codegen in the downstream crate.

This is why generic-heavy crates like `serde`, `itertools`, and `rayon` contribute more to your compile time than their line count would suggest. Their code is not compiled once — it is compiled in every crate that uses them, once for each unique type substitution.

`#[inline]` and Its Real Purpose

You might think `#[inline]` is about telling LLVM to inline a function. That is only part of the story. Its more important effect is controlling **cross-crate availability of function bodies**.

For non-generic functions, the compiled machine code lives in the crate that defined them. A downstream crate calls them via a normal linker symbol. LLVM in the downstream crate cannot inline them because it does not have the function body — unless the function is marked `#[inline]`.

When you write `#[inline]` on a non-generic function, rustc serializes that function's MIR into the crate metadata, just like it does for generic functions. This makes the function body available for LLVM to inline in downstream crates.

For generic functions, `#[inline]` is less important because their MIR is already serialized into metadata (it has to be — they cannot be compiled without concrete type parameters). But `#[inline]` on a generic function still serves as a *hint* to LLVM's inliner, suggesting that the function is a good inlining candidate.

Here is the hierarchy:

Annotation	MIR in metadata?	LLVM inlining hint?
(none, non-generic)	No	No
`#[inline]` (non-generic)	Yes	Yes
(none, generic)	Yes (required)	No
`#[inline]` (generic)	Yes (required)	Yes
`#[inline(always)]`	Yes	Strong yes
`#[inline(never)]`	Depends	No

The Codegen Units Wrinkle

Rustc splits a crate's codegen work into multiple **codegen units** (CGUs) for parallel compilation. By default, a release build uses 16 CGUs. Each CGU is compiled to LLVM IR independently, and LLVM optimizes each one independently.

This means LLVM cannot inline a function call if the caller and callee are in different CGUs — unless the callee is marked `#[inline]` or is available as a generic instantiation. With `codegen-units = 1`, LLVM sees everything and can inline aggressively, which is why `codegen-units = 1` often produces faster (and smaller) binaries at the cost of slower compilation.

Thin LTO partially solves this by allowing cross-CGU optimization, but it adds its own compile-time cost.

Comparison with C++ and Java

Rust's monomorphization approach is not unique. It sits at one end of a spectrum, with Java's type erasure at the other end. Understanding where each approach falls clarifies the tradeoffs.

C++ Templates: Same Strategy, Fewer Guardrails

C++ templates and Rust generics use essentially the same monomorphization strategy. `template<typename T> void foo(T x)` in C++ produces a separate function body for each type it is called with, just like Rust.

The key differences are in error checking and code organization:

Error checking. In Rust, a generic function is type-checked against its trait bounds *once*, when the generic function is compiled. If you write `fn foo<T: Display>(x: T)` and try to call `x.nonexistent_method()`, the compiler rejects it immediately, even if no caller has instantiated `foo` yet. In C++, templates are checked only when instantiated. A template can contain arbitrary code that is syntactically valid but semantically meaningless — the error surfaces only at the call site, often producing multi-page error messages that trace through layers of template instantiation.

C++20's concepts partially address this by adding Rust-style constraints, but they are opt-in and the language still permits unconstrained templates.

Code organization. In C++, template definitions traditionally had to appear in header files because the calling translation unit needs the full definition to instantiate the template. C++20 modules change this story somewhat, but the header-inclusion model is still dominant. In Rust, generic function bodies are serialized as MIR in `.rlib` metadata — the calling crate gets the function body automatically without any manual "header" inclusion.

Bloat mitigation. C++ linkers perform identical-COMDAT-folding (ICF), which merges functions with identical machine code. If `vector<int>::push_back` and `vector<unsigned>::push_back` produce identical code (which they often do, since both are 4-byte types), the linker can detect this and keep only one copy. Rust's linker can also do ICF (pass `-C link-arg=-Wl,--icf=all` on Linux with lld, or use the default behavior of mold), but it is not enabled by default and is not as commonly relied upon.

Java's Type Erasure: The Opposite Approach

Java generics use **type erasure**: `List<String>` and `List<Integer>` compile to the same bytecode, operating on `Object` references. The type parameter exists only at compile time for type checking; at runtime, there is one `List` class, and all elements are accessed through `Object` references with runtime casts.

The tradeoffs are almost perfectly inverted from Rust's approach:

Aspect	Rust (Monomorphization)	Java (Type Erasure)
Runtime performance	No dispatch overhead, LLVM can inline and optimize per-type	Indirect access through Object references, runtime casts, no per-type optimization
Binary/bytecode size	Grows with each type instantiation	Constant regardless of type instantiations
Compile time	Grows with instantiations	Constant
Primitive types	`Vec<u32>` stores u32 values directly, no boxing	`List<Integer>` requires boxing int to Integer, heap allocation per element
Specialization	Every instantiation is inherently specialized	No specialization (though Valhalla/Project Leyden aim to change this)
Reflection	Type parameter info erased at compile time (Rust has no runtime reflection anyway)	Type parameter info erased at runtime, leading to `List<String>` and `List<Integer>`

Aspect	Rust (Monomorphization)	Java (Type Erasure)
		being the same `Class`

Java's approach is viable because of two factors that do not apply to Rust. First, Java's garbage collector amortizes the cost of boxing primitives. Second, the JIT compiler can speculatively devirtualize and inline at runtime based on observed types, partially recovering the performance that monomorphization provides statically.

Project Valhalla is Java's ongoing effort to get the best of both worlds: value types that can be stored inline in collections (like Rust's `Vec<u32>` stores `u32` directly) while maintaining the single-compiled-class model where possible. It is, in essence, adding selective monomorphization to a language that was designed around erasure.

The Middle Ground: Shared Generics in Other Languages

Some languages attempt a middle path. Swift, for example, uses "witness tables" (similar to vtables) for generic code by default but can specialize (monomorphize) when the concrete type is visible. This gives smaller binaries in the common case and fast code when the optimizer can see through the abstraction.

.NET's CLR performs monomorphization for value types (so `List<int>` gets its own code) but shares a single implementation for all reference types (so `List<string>` and `List<object>` share code, since all references are the same size). This is a pragmatic hybrid that works well for a managed runtime.

Rust's choice to always monomorphize is consistent with its "zero-cost abstractions" principle: you should never pay a runtime cost for using generics. The compile-time and binary-size costs are considered acceptable tradeoffs, especially since developers can opt into dynamic dispatch with `dyn Trait` when they disagree with that default.

Practical Strategies for Controlling Bloat

Understanding monomorphization is not just academic. Here are concrete techniques used in production Rust codebases to manage the tradeoff:

Extract non-generic inner functions. As shown earlier, move type-agnostic logic out of generic functions. The `hashbrown` crate (Rust's `HashMap` implementation) does this extensively — the probing and table management logic works on raw bytes and metadata, with a thin generic layer on top.

Use `dyn Trait` for large, cold code paths. Error handling, logging, serialization output, and configuration parsing are rarely in hot loops. Using `dyn Write`, `dyn Error`, or `dyn Serialize` in these paths trades negligible runtime performance for significant binary size savings.

Reduce the number of distinct type instantiations. If you have `process::<MyTypeA>` and `process::<MyTypeB>` but the two types are structurally identical (same size, same alignment, same behavior), consider using a single type with a runtime discriminant instead.

Enable LTO and linker ICF. Thin LTO (`lto = "thin"` in `Cargo.toml`) allows LLVM to optimize across codegen units. Linker ICF merges identical function bodies. Together, they can recover a significant fraction of the bloat without any source code changes. With `lld` or `mold`:

```
# Cargo.toml
[profile.release]
lto = "thin"

# .cargo/config.toml (for lld with ICF on Linux)
[target.x86_64-unknown-linux-gnu]
rustflags = ["-C", "link-arg=-Wl,--icf=all"]
```

Profile first. Do not blindly replace `impl Trait` with `dyn Trait` everywhere. Use `cargo bloat --release` to find the actual largest monomorphized families, and target those. Often, a handful of heavily-generic functions account for the majority of the bloat.

Summary

Monomorphization is the engine behind Rust's zero-cost generics. It produces fast code by eliminating all abstraction overhead at the cost of duplicating function bodies for every type instantiation. The resulting binary size increase is measurable and sometimes significant, especially in generic-heavy codebases.

The compiler's polymorphization effort aims to reduce unnecessary duplication but remains unstable and limited. In practice, the developer's main tools for managing bloat are architectural: extracting non-generic inner functions, using `dyn Trait` for cold paths, and relying on LTO and linker optimizations to clean up the rest.

Cross-crate monomorphization adds another dimension — generic function MIR must be serialized into crate metadata, and the actual codegen happens in the calling crate, which is why generic-heavy dependencies disproportionately affect compile times.

Compared to C++ (similar strategy, weaker type-checking at definition time) and Java (opposite strategy via type erasure, with runtime costs), Rust's approach is a deliberate choice that prioritizes runtime performance over binary size and compile time. It is the right default for a systems language, and `dyn Trait` provides the escape hatch for when it is not.