# Rust for Haters

Welcome to **Rust for Haters** - educational Rust documentation generated from YouTube video transcripts.

## Topics

## Deep Dives

### Advanced Data Structures

- Chess Engine: 0x88 Board Representation & Core Types

### Comprehensive Learning Guide

- Comprehensive Rust Learning Documentation

### Error Handling

- The `.unwrap()` Method in Rust

### Javascripts

**Source Files:** - `tts-reader.js` [src] [doc]

### Memory

- Advanced Rust: Ownership, Borrowing, and Encapsulation
- Advanced Rust Programming: Ownership, Borrowing, and Encapsulation
- Advanced Rust Programming: Ownership, Borrowing, and Encapsulation

### Compiler Internals

- Inside the Rust Compiler: From Source Code to Binary

### Standard Library

- Chapter 1: Memory Layout and Alignment
- Chapter 2: The Allocator Traits
- Chapter 3: Box — Owned Heap Allocation
- Chapter 4: Vec — Dynamic Arrays

**Source Files:** - `box_exploration.rs` [src] [doc] - `vec_exploration.rs` [src] [doc]

### Syntax And Patterns

- Rust Syntax, Keywords, and Patterns Reference

### Toolchain

- Rust Terminal Development Cheatsheet

### Tts

- Ideas and Code for TTS for mkdocs

## Lecture Series

- **Ripgrep** — 39 lectures
- **Rust Memory Safety Examples** — 7 lectures

## Contributing

New documentation is generated using the transcript processing tool:

```
python process_transcript.py "https://youtu.be/VIDEO_ID" --push
```

# Chess Engine: 0x88 Board Representation & Core Types

## Overview

This chess engine uses the **0x88 board representation**, a clever trick for efficient board bounds checking. This section establishes the fundamental data structures and utility functions that the rest of the engine builds upon.

## The 0x88 Board Representation

### Core Concept

```
pub type Sq = usize; // Square index type (0..127)
const BOARD_SIZE: usize = 128; // Using 0x88 representation
```

Instead of using a standard 64-square array (8×8), this engine uses a 128-element array. Here's why this is brilliant:

- **Standard approach**: Square indices 0-63 for an 8×8 board
- **0x88 approach**: Square indices spread across 0-127, using only specific indices

### The Magic of 0x88

The key insight is in the binary representation: - Valid squares have indices where `(index & 0x88) == 0` - 0x88 in binary is `10001000` - This means valid squares must have: - Bit 7 = 0 (values < 128) - Bit 3 = 0 (file values 0-7)

### Square Indexing Formula

```
fn sq(rank: i32, file: i32) -> Sq {
    ((rank << 4) | file) as usize
}
```

This creates indices like: - a1 (rank=0, file=0): `(0 << 4) | 0 = 0x00 = 0` - h1 (rank=0, file=7): `(0 << 4) | 7 = 0x07 = 7` - a8 (rank=7, file=0): `(7 << 4) | 0 = 0x70 = 112` - h8 (rank=7, file=7): `(7 << 4) | 7 = 0x77 = 119`

### Why This Matters

The 0x88 trick enables ultra-fast bounds checking:

```
fn on_board(s: Sq) -> bool {
    (s & 0x88) == 0
}
```

When you move a piece by adding offsets (e.g., knight moves +31, +33, etc.), invalid moves naturally produce indices with bits 3 or 7 set, failing the bounds check without explicit range testing.

## Piece Representation

### The Piece Enum

```
#[derive(Copy, Clone, PartialEq, Eq, Debug)]
enum Piece {
    Empty,
    WP, WN, WB, WR, WQ, WK,  // White pieces
    BP, BN, BB, BR, BQ, BK,  // Black pieces
}
```

**Key Design Decisions:** - `Copy` trait: Pieces are simple values that can be copied cheaply - `PartialEq/Eq`: Enables piece comparison (e.g., `piece == Piece::Empty`) - Separate enum values for each color simplifies color checking

### Piece Utility Methods

**Character Conversion:**

```
fn from_char(c: char) -> Piece  // 'P' -> Piece::WP, 'p' -> Piece::BP
fn to_char(self) -> char        // Piece::WP -> 'P', Empty -> '.'
```

This follows standard chess notation where uppercase = white, lowercase = black.

**Color Detection:**

```
fn is_white(self) -> bool  // Pattern matches white pieces
fn is_black(self) -> bool  // Pattern matches black pieces
fn is_empty(self) -> bool  // Checks if square is empty
```

These use Rust's pattern matching for efficient, branch-free checks.

## Coordinate Translation

### Algebraic Notation Conversion

```
fn sq_to_alg(s: Sq) -> String {
    let r = (s >> 4) as i32;  // Extract rank (high nibble)
    let f = (s & 15) as i32;   // Extract file (low nibble)
    // Converts to "e2", "d7", etc.
}

fn alg_to_sq(s: &str) -> Option<Sq> {
    // Parses "e2" -> square index
    // Returns None for invalid input
}
```

**Bit Manipulation Details:** - `s >> 4`: Right shift by 4 bits extracts the rank (divides by 16) - `s & 15`: Masks lower 4 bits to extract file (modulo 16) - Files map to 'a'-'h', ranks to '1'-'8'

## Why This Foundation Matters

This section establishes:

1. **Efficient Board Access**: O(1) bounds checking without comparisons
2. **Clean Abstractions**: Type-safe square indices prevent bugs
3. **Standard Interfaces**: FEN/algebraic notation compatibility
4. **Memory Efficiency**: Despite using 128 slots, only 64 are active
5. **Performance**: Bit operations are faster than arithmetic comparisons

## Practical Usage Patterns

When the engine evaluates moves:

```
// Example: Checking if a knight move is valid
let knight_offsets = [31, 33, 18, -14, -31, -33, -18, 14];
for offset in knight_offsets {
    let target = (current_square as i32 + offset) as Sq;
    if on_board(target) {  // Lightning-fast bounds check!
        // Process valid move
    }
}
```

## Technical Notes for Rust Learners

### Memory Layout

- The board array has 128 elements but only uses indices where `(i & 0x88) == 0`
- This wastes 64 array slots but gains massive performance in bounds checking
- Modern CPUs handle this sparse array well due to caching

### Type Safety

- Using `type Sq = usize` creates a type alias, not a new type
- This provides semantic clarity without runtime overhead
- Consider using a newtype pattern (`struct Sq(usize)`) for stronger type safety in production code

### Performance Characteristics

- Bounds checking: Single AND operation + comparison (2 CPU instructions)
- Traditional approach: 4 comparisons (rank >= 0, rank < 8, file >= 0, file < 8)
- This adds up over millions of move evaluations in search trees

**Next Section Preview:** The Board State section builds on this foundation to manage the complete game state, including castling rights, en passant, and move history.

# 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.

# The `.unwrap()` Method in Rust

## Core Purpose

`.unwrap()` is a method that extracts the value from `Option<T>` or `Result<T, E>` types, but **panics** if the value isn't there. It's essentially saying: "I'm certain this has a value, and if I'm wrong, crash the program."

---

## The Two Implementations

### Option::unwrap()

```rust
// Simplified from std library
impl<T> Option<T> {
    pub fn unwrap(self) -> T {
        match self {
            Some(val) => val,
            None => panic!("called `Option::unwrap()` on a `None` value"),
        }
    }
}
```

**What happens:**

1. Takes ownership of the `Option<T>` (consumes it)
2. Pattern matches on the enum variant
3. If `Some(val)`, returns the inner `T` value by moving it out
4. If `None`, calls `panic!()` macro

### Result::unwrap()

```rust
// Simplified from std library
impl<T, E: fmt::Debug> Result<T, E> {
    pub fn unwrap(self) -> T {
        match self {
            Ok(val) => val,
            Err(err) => panic!("called `Result::unwrap()` on an `Err` value:
{:?}", err),
        }
    }
}
```

**Key difference:** Prints the error value when panicking (requires `E: Debug`).

---

# Memory Layout & Performance

## Zero-Cost Abstraction

```rust
let x: Option<i32> = Some(42);
let value = x.unwrap();
```

After optimization, this compiles to roughly:

```assembly
# Check if Option is Some (discriminant check)
cmp byte ptr [rsp], 1       # Compare tag byte with 1 (Some variant)
jne .panic_label            # Jump if not equal (it's None)
mov eax, dword ptr [rsp+4]  # Move the i32 value to return register
# ... continue execution

.panic_label:
    call panic_function
```

The compiler knows `Option<i32>` is laid out as:

| Offset | Size | Content |
|---|---|---|
| 0 | 1 byte | Discriminant (0/1) |
| 1-3 | 3 bytes | Padding |
| 4-7 | 4 bytes | The i32 value |

### What Panic Actually Does

When `unwrap()` panics:

1. **Unwind preparation**: Compiler inserts unwinding tables at compile time
2. **Stack unwinding**: Walks back up the call stack, running destructors for all in-scope values
3. **Resource cleanup**: Drops all RAII objects (files close, locks release, memory frees)
4. **Thread termination**: By default, kills just the current thread
5. **Main thread special case**: If main thread panics, entire process exits

---

## Design Rationale

### Why Have `.unwrap()` At All?

**Problem**: Rust forces you to handle `Option` and `Result`. Sometimes you need an escape hatch:

```
// Prototyping - you know it works but handling properly is verbose
let config = parse_config().unwrap();

// Provably unreachable None case
let index = vec![1, 2, 3].iter().position(|&x| x == 2).unwrap();
// We KNOW 2 is in the array, so unwrap is safe here
```

**Design philosophy**: Make the dangerous operation explicit and searchable. Every `.unwrap()` in your codebase is easy to grep for during code review.

### The Panic vs Return Question

Rust could have made `unwrap()` return a default value or special sentinel, but that has problems:

```
// If unwrap returned 0 on None:
let value = dangerous_operation().unwrap(); // Returns 0
if value == 0 {
    // Is this a real zero or an error? Can't tell!
}
```

**Panic is better because:**

- Loud failure beats silent corruption
- Forces you to think about error handling
- Can't accidentally ignore the error
- Backtraces show exactly where it failed

---

## When to Use (and Not Use)

### Legitimate Uses

**1. Prototypes and examples:**

```
let file = File::open("config.toml").unwrap();
```

**2. Tests:**

```
#[test]
fn test_parsing() {
    let result = parse("valid input").unwrap();
    assert_eq!(result, expected);
}
```

**3. Provably safe scenarios:**

```
let mut map = HashMap::new();
map.insert("key", 42);
let value = map.get("key").unwrap(); // We JUST inserted it
```

### Alternatives (Better Choices)

**1. Pattern matching:**

```
match file_result {
    Ok(file) => process(file),
    Err(e) => handle_error(e),
}
```

**2. `.expect()` - Better than unwrap:**

```rust
let config = parse_config()
    .expect("Failed to parse config.toml - check syntax");
```

Provides context in panic message. Always prefer this over `.unwrap()`.

**3. The `?` operator:**

```rust
fn read_file() -> Result<String, io::Error> {
    let mut file = File::open("data.txt")?;  // Propagates error
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    Ok(contents)
}
```

**4. `.unwrap_or()` / `.unwrap_or_else()`:**

```rust
let value = maybe_value.unwrap_or(default);
let value = maybe_value.unwrap_or_else(|| expensive_computation());
```

**5. `.unwrap_or_default()`:**

```rust
let count: i32 = user_input.parse().unwrap_or_default(); // 0 on failure
```

## Related Methods

| Method | Behavior |
| --- | --- |
| `.expect("msg")` | Panic with custom message |
| `.unwrap_unchecked()` | No check - undefined behavior if wrong! |
| `.unwrap_or(val)` | Return `val` if None/Err |
| `.unwrap_or_else(f)` | Call closure `f` if None/Err |
| `.unwrap_or_default()` | Return `Default::default()` if None/Err |
| `.ok_or(err)?` | Convert Option to Result, then propagate |

## Performance Characteristics

### Hot Path Concern

```
// In a tight loop - unwrap adds a branch
for item in large_collection {
    let value = item.parse::<i32>().unwrap(); // Branch + possible panic
}
```

### Cold Path Optimization

The panic path is marked as "cold" (unlikely) for branch prediction:

```
// Actual stdlib implementation uses #[cold] and #[inline(never)]
#[cold]
#[inline(never)]
fn unwrap_failed() -> ! {
    panic!("called `Option::unwrap()` on a `None` value")
}
```

This tells the CPU to assume the happy path and not waste cache/pipeline resources on the panic branch.

## The Core Insight

`.unwrap()` is a **controlled explosion**. It's Rust saying:

> "You MUST handle errors. If you refuse, you don't get silent corruption or undefined behavior - you get a loud, traceable crash with cleanup."

It's a safety valve that maintains memory safety while acknowledging that sometimes you really do know better than the type system, or you're just prototyping and proper error handling can come later.

The key is that it's **explicit, searchable, and safe** even when you're wrong - it crashes cleanly instead of corrupting memory.

tts-reader.js

```
// TTS Lecture Reader for MkDocs - VibeVoice Edition
// Uses VibeVoice neural TTS for high-quality speech synthesis

class LectureReader {
  constructor() {
    // VibeVoice WebSocket endpoint
    this.ttsEndpoint = 'wss://rfh.tinymachines.ai/tts/stream';

    // Audio playback
    this.audioCtx = null;
    this.audioQueue = [];
    this.isProcessingAudio = false;
    this.socket = null;

    // State
    this.chunks = [];
    this.currentChunk = 0;
    this.isPlaying = false;
    this.isPaused = false;

    // Settings - actual VibeVoice presets
    this.selectedVoice = 'en-Carter_man';
    this.voices = [
      { name: 'Carter (Male)', value: 'en-Carter_man' },
      { name: 'Davis (Male)', value: 'en-Davis_man' },
      { name: 'Frank (Male)', value: 'en-Frank_man' },
      { name: 'Mike (Male)', value: 'en-Mike_man' },
      { name: 'Emma (Female)', value: 'en-Emma_woman' },
      { name: 'Grace (Female)', value: 'en-Grace_woman' },
      { name: 'Samuel (Indian)', value: 'in-Samuel_man' },
    ];

    // Audio settings
    this.sampleRate = 24000;
    this.bufferSize = 4096;
    this.prebufferSize = 8192; // Prebuffer before starting playback

    this.init();
  }

  init() {
    this.createUI();
    this.bindEvents();
    this.populateVoiceSelector();
  }

  createUI() {
    const widget = document.createElement('div');
    widget.id = 'tts-reader';
    widget.innerHTML = `
      <div class="tts-collapsed" id="tts-toggle">
        <svg viewBox="0 0 24 24" width="24" height="24">
          <path fill="currentColor" d="M3 9v6h4l5 5V4L7 9H3zm13.5 3c0-1.77-1.02-
3.29-2.5-4.03v8.05c1.48-.73 2.5-2.25 2.5-4.02zM14 3.23v2.06c2.89.86 5 3.54 5
```

```
6.71s-2.11 5.85-5 6.71v2.06c4.01-.91 7-4.49 7-8.77s-2.99-7.86-7-8.77z"/>
        </svg>      </div>      <div class="tts-expanded" id="tts-panel">
<div class="tts-header">          <span>Lecture Reader</span>          <button
id="tts-minimize" title="Minimize">-</button>        </div>        <div
class="tts-progress">          <div class="tts-progress-bar" id="tts-progress-
bar"></div>        </div>        <div class="tts-controls">          <button
id="tts-prev" title="Previous section">⏮</button>          <button id="tts-play"
title="Play">▶</button>          <button id="tts-next" title="Next
section">⏭</button>        </div>        <div class="tts-settings">
<label>            Voice:            <select id="tts-voice"></select>
</label>        </div>        <div class="tts-status" id="tts-status">Ready</div>
</div>    `;    document body appendChild(widget);  }  populateVoiceSelector() {
const selector = document getElementById('tts-voice');    if (!selector) return;
selector innerHTML = '';    this voices forEach(voice => {      const option =
document createElement('option');      option value = voice value;
option textContent = voice name;      if (voice value === this selectedVoice) {
option selected = true;      }      selector appendChild(option);    });  }
bindEvents() {    document getElementById('tts-toggle') addEventListener('click',
() => this togglePanel());    document getElementById('tts-
minimize') addEventListener('click', () => this togglePanel());
document getElementById('tts-play') addEventListener('click', () =>
this togglePlayback());    document getElementById('tts-
prev') addEventListener('click', () => this prevChunk());
document getElementById('tts-next') addEventListener('click', () =>
this nextChunk());    document getElementById('tts-
voice') addEventListener('change', (e) => {      this selectedVoice =
e target value;    });    // Keyboard shortcuts
document addEventListener('keydown', (e) => {      if (e target tagName ===
'INPUT' || e target tagName === 'TEXTAREA') return;      if (e key === ' ' &&
e ctrlKey) {        e preventDefault();        this togglePlayback();      } else
if (e key === 'ArrowRight' && e ctrlKey) {        e preventDefault();
this nextChunk();      } else if (e key === 'ArrowLeft' && e ctrlKey) {
e preventDefault();        this prevChunk();      }    });  }  togglePanel() {
const panel = document getElementById('tts-panel');    const toggle =
document getElementById('tts-toggle');    panel classList toggle('visible');
toggle classList toggle('hidden');    if (this chunks length === 0) {
this extractContent();    }  }  extractContent() {    const content =
document querySelector('.md-content__inner') ||
document querySelector('article') ||
document querySelector('main');    if (!content) {      this setStatus('No
content found');      return;    }    const clone = content cloneNode(true);
const removeSelectors = [      'script', 'style', 'nav', '.headerlink', '.toc',
'pre', 'code', '.highlight',      '.admonition.note',      '#tts-reader',
'[data-tts-skip]'    ];    removeSelectors forEach(sel => {
clone querySelectorAll(sel) forEach(el => el remove());    });    let text =
clone textContent || '';    text = text replace(/\s+/g, ' ').replace(/\n+/g,
'\n').trim();    // Chunk into ~500 char segments for streaming    this chunks =
this chunkText(text, 500);    this currentChunk = 0;
this setStatus(`${this chunks length} sections ready`);    this updateProgress();
}  chunkText(text, maxLen = 500) {    const chunks = [];    const sentences =
text match(/[^.!?]+[.!?]+/g) || [text];    let currentChunk = '';    for (const
sentence of sentences) {      if ((currentChunk + sentence).length > maxLen) {
if (currentChunk) chunks push(currentChunk trim());        currentChunk =
sentence;      } else {        currentChunk += sentence;      }    }    if
```

```
(currentChunk.trim()) chunks.push(currentChunk.trim());    return chunks;  }
initAudio() {    if (!this.audioCtx) {      this.audioCtx = new
(window.AudioContext || window.webkitAudioContext)({        sampleRate:
this.sampleRate      });    }    if (this.audioCtx.state === 'suspended') {
this.audioCtx.resume();    }  }  async play() {    if (this.chunks.length === 0)
{      this.extractContent();      if (this.chunks.length === 0) return;    }
this.initAudio();    this.isPlaying = true;    this.isPaused = false;
this.updatePlayButton();    await this.speakChunk(this.currentChunk);  }  async
speakChunk(index) {    if (index >= this.chunks.length) {      this.stop();
this.setStatus('Finished');      return;    }    this.currentChunk = index;
const text = this.chunks[index];    this.setStatus(`Loading section ${index +
1}...`);    this.updateProgress();    try {      await this.streamTTS(text);
// Auto-advance to next chunk      if (this.isPlaying && !this.isPaused) {
this.currentChunk++;        if (this.currentChunk < this.chunks.length) {
await this.speakChunk(this.currentChunk);        } else {          this.stop();
this.setStatus('Finished');        }      }    } catch (err) {
console.error('TTS error:', err);      this.setStatus(`Error: ${err.message}`);
this.stop();    }  }  streamTTS(text) {    return new Promise((resolve, reject)
=> {      const params = new URLSearchParams({        text: text,        voice:
this.selectedVoice,        cfg: '1.5'      });      const wsUrl =
`${this.ttsEndpoint}?${params.toString()}`;      this.socket = new
WebSocket(wsUrl);      this.socket.binaryType = 'arraybuffer';
this.audioQueue = [];      let audioStarted = false;      let totalSamples = 0;
this.socket.onopen = () => {        this.setStatus(`Section ${this.currentChunk +
1} of ${this.chunks.length}`);      };      this.socket.onmessage = (event) => {
if (typeof event.data === 'string') {          // JSON log message          try {
const msg = JSON.parse(event.data);            if (msg.event ===
'generation_error') {              reject(new Error(msg.data?.error ||
'Generation failed'));            }          } catch (e) {            // Ignore
parse errors          }          return;        }        // Binary PCM16 audio
data        const pcm16 = new Int16Array(event.data);        const float32 = new
Float32Array(pcm16.length);        for (let i = 0; i < pcm16.length; i++) {
float32[i] = pcm16[i] / 32768;        }        this.audioQueue.push(float32);
totalSamples += float32.length;        // Start playback after prebuffering
if (!audioStarted && totalSamples >= this.prebufferSize) {          audioStarted
= true;          this.playAudioQueue();        }      };      this.socket.onclose
= () => {        // Play remaining audio        if (this.audioQueue.length > 0 &&
!audioStarted) {          this.playAudioQueue();        }        // Wait for
audio to finish        const checkComplete = () => {          if
(this.audioQueue.length === 0 && !this.isProcessingAudio) {            resolve();
} else if (this.isPlaying) {            setTimeout(checkComplete, 100);
} else {            resolve();          }        };
setTimeout(checkComplete, 100);      };      this.socket.onerror = (err) => {
reject(new Error('WebSocket connection failed'));      };    });  }
playAudioQueue() {    if (this.isProcessingAudio || this.audioQueue.length === 0
|| !this.isPlaying) {      return;    }    this.isProcessingAudio = true;    //
Combine all queued audio    const totalLength = this.audioQueue.reduce((sum, arr)
=> sum + arr.length, 0);    const combined = new Float32Array(totalLength);
let offset = 0;    while (this.audioQueue.length > 0) {      const chunk =
this.audioQueue.shift();      combined.set(chunk, offset);      offset +=
chunk.length;    }    // Create and play audio buffer    const buffer =
this.audioCtx.createBuffer(1, combined.length, this.sampleRate);
buffer.getChannelData(0).set(combined);    const source =
this.audioCtx.createBufferSource();    source.buffer = buffer;
```

```
source.connect(this.audioCtx.destination);    source.onended = () => {
this.isProcessingAudio = false;       // Check for more audio       if
(this.audioQueue.length > 0) {         this.playAudioQueue();       }    };
source.start();  }  pause() {    this.isPaused = true;    this.isPlaying = false;
this.updatePlayButton();    this.setStatus('Paused');    if (this.socket &&
this.socket.readyState === WebSocket.OPEN) {      this.socket.close();    }  }
stop() {    this.isPlaying = false;    this.isPaused = false;
this.updatePlayButton();    this.audioQueue = [];    if (this.socket &&
this.socket.readyState === WebSocket.OPEN) {      this.socket.close();    }  }
togglePlayback() {    if (this.isPlaying) {      this.pause();    } else {
this.play();    }  }  prevChunk() {    this.stop();    this.currentChunk =
Math.max(0, this.currentChunk - 1);    this.updateProgress();
this.setStatus(`Section ${this.currentChunk + 1} of ${this.chunks.length}`);
this.play();  }  nextChunk() {    this.stop();    this.currentChunk =
Math.min(this.chunks.length - 1, this.currentChunk + 1);
this.updateProgress();    this.setStatus(`Section ${this.currentChunk + 1} of
${this.chunks.length}`);    this.play();  }  updatePlayButton() {    const btn =
document.getElementById('tts-play');    btn.textContent = this.isPlaying ? '⏸' :
'▶';    btn.title = this.isPlaying ? 'Pause' : 'Play';  }  updateProgress() {
const bar = document.getElementById('tts-progress-bar');    const percent =
this.chunks.length > 0      ? ((this.currentChunk + 1) / this.chunks.length) *
100      : 0;    bar.style.width = `${percent}%`;  }  setStatus(msg) {
document.getElementById('tts-status').textContent = msg;  }}// Initialize when
DOM is readydocument.addEventListener('DOMContentLoaded', () => {
window.lectureReader = new LectureReader();});
```

# Mastering Rust Through ripgrep: A Deep Dive into Production Systems Programming

## Course Introduction

Welcome to an in-depth exploration of one of the most respected Rust codebases in the ecosystem. ripgrep isn't just a fast grep replacement—it's a masterclass in systems programming, demonstrating how to build performant, maintainable, and user-friendly command-line tools in Rust.

Created by Andrew Gallick (BurntSushi), ripgrep has become a reference implementation for many Rust patterns. When Rust developers want to understand how to properly implement the builder pattern, design trait hierarchies, or structure a multi-crate workspace, they often turn to ripgrep. By studying this codebase, you're not just learning one tool—you're learning idioms and patterns that will serve you across your entire Rust journey.

What makes ripgrep particularly valuable for learning? It sits at the intersection of several challenging domains: high-performance text processing, cross-platform compatibility, parallel execution, and complex configuration management. Yet the code remains remarkably readable, with clear abstractions and thoughtful organization. This is production Rust at its finest.

Throughout this course, you'll discover how ripgrep achieves grep-like functionality while consistently outperforming traditional tools. You'll see how careful API design makes complex functionality accessible, how the type system prevents entire categories of bugs, and how Rust's ownership model enables both safety and performance.

---

## Prerequisites

This course assumes you have a solid foundation in programming and basic familiarity with Rust. Specifically, you should be comfortable with:

### Rust Fundamentals

- **Ownership and borrowing**: You understand why Rust has these concepts and can work with references without fighting the borrow checker constantly

- **Structs, enums, and pattern matching**: These are Rust's bread and butter for data modeling
- **Traits**: You know how to define and implement traits, even if advanced patterns like trait objects are still fuzzy
- **Error handling**: You're familiar with `Result`, `Option`, and the `?` operator
- **Modules and crates**: You understand how Rust organizes code into modules and how dependencies work

### General Programming Knowledge

- **Regular expressions**: You should understand basic regex syntax and concepts
- **Command-line tools**: Familiarity with grep, find, or similar tools helps contextualize the problem domain
- **Concurrent programming concepts**: Understanding threads and parallelism at a conceptual level is helpful, though we'll explain ripgrep's specific approaches

### Nice to Have (But Not Required)

- Experience with the builder pattern in any language
- Familiarity with memory-mapped files
- Understanding of Unicode text processing challenges

If you're still learning Rust fundamentals, consider working through the official Rust Book first. This course will be here when you're ready to see how these concepts come together in production code.

## Learning Objectives

By completing this course, you will:

### Understand Production Rust Architecture

- Recognize how to structure a multi-crate Rust workspace where each crate has a focused responsibility
- See how the facade pattern creates clean public APIs that hide internal complexity
- Learn when and why to split functionality across multiple crates

### Master the Builder Pattern Ecosystem

- Implement builders that validate configuration at construction time
- Understand the trade-offs between eager and lazy validation
- See how builders compose with other builders to configure complex systems

### Develop Trait-Based Abstraction Skills

- Design traits that define clear contracts while allowing diverse implementations
- Understand how ripgrep's `Matcher` and `Sink` traits enable extensibility
- Learn when trait objects are appropriate versus generic parameters

### Gain Performance Engineering Intuition

- Recognize patterns for memory-efficient processing of large inputs
- Understand when to use memory mapping versus buffered reading
- See how literal optimization extracts fast paths from complex patterns
- Learn how parallel iteration is implemented with work-stealing

### Appreciate Systems Programming Patterns

- Handle platform differences gracefully with conditional compilation
- Manage external processes with proper timeout and cleanup behavior
- Work with terminal capabilities and ANSI color codes
- Process compressed files transparently

### Navigate Real-World Complexity

- See how configuration flows from command-line flags through to execution
- Understand the layers involved in a seemingly simple "search for pattern in files" operation
- Appreciate the edge cases that production tools must handle

---

# How to Use This Course

This course is structured for deep, engaged learning. Each lesson consists of two companion documents that work together:

## Lecture Documents

These are the documents you're reading now—prose explanations written in a lecture style. They: - Introduce concepts before you see the code - Explain the "why" behind design decisions - Connect implementation details to broader Rust principles - Highlight patterns you'll encounter repeatedly - Discuss trade-offs and alternatives that were considered

Lecture documents work well for reading or listening (if converted to audio). They provide the conceptual framework that makes the code meaningful.

## Code Companion Documents

Each lecture has an accompanying code companion that contains: - Annotated source code excerpts - Inline comments explaining specific lines - Cross-references to related code in other lessons - Type signatures and their implications

The code companions are reference documents—you'll flip to them when the lecture mentions specific implementations.

## Suggested Approach

**For each lesson:** 1. Read the lecture document first, building mental models 2. Open the code companion alongside the actual source code 3. Trace through the code, matching it to the lecture's explanations 4. Experiment: modify the code, run tests, see what breaks

**Pacing recommendations:** - **Intensive study**: 2-3 lessons per day, with hands-on coding - **Moderate pace**: 1 lesson per day, with time for exploration - **Deep dive**: 2-3 lessons per week, implementing similar patterns yourself

**Working with the actual codebase:**

```
git clone https://github.com/BurntSushi/ripgrep
cd ripgrep
cargo build --release
```

We strongly recommend having the codebase open as you study. Reading code in your editor, with full IDE support for jumping to definitions, is far more effective than reading excerpts alone.

---

## Learning Threads Overview

Several conceptual threads weave through multiple lessons. Recognizing these threads helps you see how patterns recur and evolve throughout the codebase.

### The Builder Pattern Ecosystem

*Appears in: Lessons 7, 11, 14, 20, 25, 28*

ripgrep uses the builder pattern extensively—but not in a rote, mechanical way. You'll see builders that: - Validate configuration during building (fail-fast approach) - Allow incremental configuration with sensible defaults - Compose with other builders to configure nested systems - Use generic parameters to track configuration state at compile time

By following this thread, you'll develop intuition for when and how to apply builder patterns in your own code.

### Trait-Based Extensibility

*Appears in: Lessons 3, 5, 8, 17*

The `Matcher` and `Sink` traits are the architectural backbone of ripgrep's library ecosystem. This thread explores: - How traits define contracts between components - When to use trait objects versus generic parameters - How associated types carry additional type information - The relationship between trait design and testability

### Memory-Efficient Stream Processing

*Appears in: Lessons 12, 15, 16*

Processing potentially gigabyte-sized files with bounded memory is a fundamental challenge. This thread covers: - Line buffer design for incremental processing - When memory mapping provides benefits (and when it doesn't) - How context lines complicate streaming (and how to handle it) - Balancing memory usage against performance

### Configuration Flow

*Appears in: Lessons 33, 34, 35, 36*

Command-line tools must transform user input into program behavior. This thread follows configuration from: - Raw command-line arguments - Through low-level parsing - Into validated high-level structures - Finally becoming configured components

### Cross-Platform Considerations

*Appears in: Lessons 29, 30, 31*

Real-world tools must work across operating systems. This thread examines: - Conditional compilation for platform-specific code - Abstracting over platform differences - Handling external processes portably - Terminal capability detection

---

## Course Outline

### Part 1: The Library Ecosystem (Lessons 1-2)

**Lesson 1: The Facade Pattern** — `crates/grep/src/lib.rs` How ripgrep organizes its functionality across multiple crates while presenting a unified API. You'll see how the `grep` crate serves as a facade, re-exporting types from specialized crates without exposing internal organization.

**Lesson 2: Minimal Working Example** — `crates/grep/examples/simplegrep.rs` A complete, working grep implementation in approximately 50 lines. This lesson shows the payoff of ripgrep's API design—how all the complexity we'll study enables remarkably simple client code.

### Part 2: The Matcher Abstraction (Lessons 3-5)

**Lesson 3: The Matcher Trait** — `crates/matcher/src/lib.rs` The central abstraction that defines what it means to "match" text. You'll learn how this trait enables ripgrep to support different regex engines while maintaining a consistent interface.

**Lesson 4: Capture Group Interpolation** — `crates/matcher/src/interpolate.rs` How `$1`, `$name`, and other replacement patterns work at the byte level. This lesson demonstrates careful parsing and Rust's approach to string manipulation.

**Lesson 5: The Sink Trait** — `crates/searcher/src/sink.rs` The consumer pattern for search results. You'll see how this trait handles context lines, binary detection, and result filtering through a callback-based design.

## Part 3: Regex Integration (Lessons 6-10)

**Lesson 6: Regex Crate Entry** — `crates/regex/src/lib.rs` A brief orientation to how the regex matcher crate is organized, setting the stage for deeper exploration.

**Lesson 7: Regex Configuration** — `crates/regex/src/config.rs` Builder pattern for regex options including case sensitivity, word boundaries, and multi-line mode. See how configuration is validated and applied.

**Lesson 8: Implementing Matcher** — `crates/regex/src/matcher.rs` The concrete implementation of the `Matcher` trait for Rust's regex crate. Learn about thread-safe regex caching and how abstract interfaces meet concrete implementations.

**Lesson 9: Literal Optimization** — `crates/regex/src/literal.rs` How ripgrep extracts literal string prefixes from regex patterns to enable fast-path searching. This is a key performance optimization that makes ripgrep fast.

**Lesson 10: AST Manipulation** — `crates/regex/src/ast.rs` Parsing and transforming regex syntax trees. You'll see how ripgrep modifies regex patterns (for case-insensitive matching, for example) at the AST level.

## Part 4: The Search Engine (Lessons 11-16)

**Lesson 11: Searcher Overview** — `crates/searcher/src/lib.rs` The public API for searching. Configuration options for encoding, binary detection, line terminators, and context lines.

**Lesson 12: Efficient Line Reading** — `crates/searcher/src/line_buffer.rs` Memory-efficient buffering for line-oriented processing. You'll understand how ripgrep processes files incrementally without loading them entirely into memory.

**Lesson 13: Line Handling Utilities** — `crates/searcher/src/lines.rs` Finding line boundaries, counting lines, and handling different line terminators. The detailed work that underlies "line-oriented" searching.

**Lesson 14: Searcher Core** — `crates/searcher/src/searcher/mod.rs` The main `Searcher` and `SearcherBuilder`. How configuration options translate into search behavior.

**Lesson 15: Search Algorithm** — `crates/searcher/src/searcher/core.rs` The actual search loop. Context line handling, match iteration, and coordinating with the `Sink` trait for output.

**Lesson 16: Memory Mapping** — `crates/searcher/src/searcher/mmap.rs` When and how to use memory-mapped files. You'll learn the heuristics ripgrep uses to decide between mmap and buffered reading.

## Part 5: Output Formatting (Lessons 17-22)

**Lesson 17: Printer Architecture** — `crates/printer/src/lib.rs` Overview of the three output modes: Standard (human-readable), JSON (machine-readable), and Summary (counts only).

**Lesson 18: Terminal Colors** — `crates/printer/src/color.rs` ANSI color handling with the termcolor crate. User-configurable color schemes and terminal capability detection.

**Lesson 19: Printer Utilities** — `crates/printer/src/util.rs` Shared formatting utilities: line numbers, byte offsets, path formatting, and separator handling.

**Lesson 20: Standard Output** — `crates/printer/src/standard.rs` The main human-readable output format. Context lines, column numbers, and matching fragment highlighting.

**Lesson 21: JSON Output** — `crates/printer/src/json.rs` Machine-readable JSON output using streaming JSON generation. See how ripgrep provides structured data for tooling integration.

**Lesson 22: Summary Output** — `crates/printer/src/summary.rs` Count-only and path-only modes. Aggregation strategies for different summary styles.

## Part 6: File Discovery (Lessons 23-28)

**Lesson 23: Glob Pattern Matching** — `crates/globset/src/lib.rs` High-level glob API. `GlobSet` for efficiently matching against many patterns simultaneously.

**Lesson 24: Glob Implementation** — `crates/globset/src/glob.rs` Glob-to-regex compilation. Special cases for performance (`*.rs` is faster than `**.rs`) and correctness.

**Lesson 25: Ignore Crate Overview** — `crates/ignore/src/lib.rs` The file filtering ecosystem. How `DirEntry`, `WalkBuilder`, and ignore patterns work together.

**Lesson 26: Gitignore Parsing** — `crates/ignore/src/gitignore.rs` Full `.gitignore` implementation including negation patterns, directory-only patterns, and nested gitignore files.

**Lesson 27: File Type Definitions** — `crates/ignore/src/types.rs` Mapping file extensions and names to types. How `--type rust` knows which files to include.

**Lesson 28: Parallel Directory Walking** — `crates/ignore/src/walk.rs` The parallel walker with work-stealing. Skip logic, cycle detection, and coordination between walker threads.

### Part 7: CLI Infrastructure (Lessons 29-31)

**Lesson 29: CLI Utilities Overview** — `crates/cli/src/lib.rs` Shared CLI utilities used across ripgrep. Hostname detection, tty handling, and standard stream management.

**Lesson 30: Decompression Support** — `crates/cli/src/decompress.rs` Searching compressed files transparently. Process spawning for external decompression tools.

**Lesson 31: Process Management** — `crates/cli/src/process.rs` Running external commands with timeouts. Graceful shutdown, signal handling, and resource cleanup.

### Part 8: The Application (Lessons 32-37)

**Lesson 32: Application Entry Point** — `crates/core/main.rs` The `rg` binary. Error handling strategies, logging setup, and the structure of the main function.

**Lesson 33: Flag System Architecture** — `crates/core/flags/mod.rs` How flags are defined, documented, parsed, and converted. The metadata system that generates help text.

**Lesson 34: Low-Level Arguments** — `crates/core/flags/lowargs.rs` Direct CLI parsing with the lexopt crate. Argument normalization and early validation.

**Lesson 35: High-Level Arguments** — `crates/core/flags/hiargs.rs` Semantic validation and conversion. Creating fully-configured search components from user input.

**Lesson 36: Search Orchestration** — `crates/core/search.rs` Tying everything together. Parallel versus sequential search, result aggregation, and exit code handling.

**Lesson 37: Haystack Discovery** — `crates/core/haystack.rs` Finding files to search. Stdin handling, explicit path arguments, and integration with the walker.

---

## Next Steps

After completing this course, you'll have a deep understanding of ripgrep's architecture and the patterns that make it successful. Here are paths for continued learning:

### Extend ripgrep

- Implement a new output format (perhaps SARIF for static analysis tools)
- Add a new matcher backend for a different regex engine
- Create a new file type definition for languages you use

### Apply These Patterns

- Build your own CLI tool using ripgrep's architectural patterns
- Implement a builder-heavy API for a library you're creating
- Design trait hierarchies for extensible systems

### Explore Related Codebases

- **BurntSushi's other projects**: The `regex`, `memchr`, and `aho-corasick` crates demonstrate low-level optimization techniques
- **fd**: A fast `find` replacement that shares some patterns with ripgrep
- **tokio**: For async Rust patterns at a similar level of sophistication

### Contribute

The best way to solidify understanding is to contribute. ripgrep's issue tracker has discussions about potential improvements, and even documentation contributions help.

---

## Ready to Begin?

Turn to **Lesson 1: The Facade Pattern** to start your journey. You'll see how ripgrep presents a unified interface to its users while maintaining clean internal organization. It's an elegant introduction to the architectural thinking that pervades this codebase.

Remember: take your time, experiment with the code, and don't hesitate to re-read sections. Production codebases reward patient study. Every lesson you complete builds understanding that compounds in later lessons.

Let's dive in.

# ripgrep crates/grep/src/lib.rs: The Facade Pattern

## What This File Does

This tiny file is the architectural keystone of ripgrep's library ecosystem. In just twenty lines, it establishes how ripgrep's modular crates present themselves as a unified library to external consumers. Rather than forcing users to understand and depend on six separate crates with their own versioning and compatibility concerns, this facade crate re-exports everything through a single entry point.

The file demonstrates a powerful software engineering pattern: building complex systems from independent, testable components while hiding that complexity from users. It's a masterclass in API design economy—the less code here, the more valuable it becomes, because every line represents a deliberate architectural decision about what to expose and how.

See: Companion Code Section 1

## Section 1: The Module Documentation Block

The file opens with a documentation comment using Rust's `/*!` syntax, which attaches documentation to the enclosing item—in this case, the crate itself. This becomes the landing page when users browse the library on docs.rs or generate local documentation. Notice the honesty in the documentation: it explicitly states that no high-level guide exists yet, and that examples are sparse. This kind of forthright documentation builds trust with users and sets appropriate expectations.

The phrasing "high level facade" is significant. In software architecture, a **facade** is a structural pattern that provides a simplified interface to a complex subsystem. The documentation is telling you exactly what role this crate plays: it's not where the work happens, it's where the work gets organized and presented. This distinction matters when you're navigating a large codebase—you now know to look elsewhere for implementation details.

See: Companion Code Section 2

## Section 2: Public Re-exports as Architecture

The core of this file is a series of `pub extern crate` declarations. This syntax might look unfamiliar if you've learned Rust recently, because modern Rust often uses `pub use` for re-exports. The `extern crate` syntax is older but still valid, and here it serves a specific purpose: it makes entire crates available as submodules of the facade crate. When someone depends on `grep`, they can access `grep::searcher`, `grep::printer`, and so on without adding those crates to their own `Cargo.toml`.

The renaming pattern is elegant: `grep_cli as cli`, `grep_matcher as matcher`, and so forth. The actual crate names follow Rust's ecosystem convention of prefixing with the project name to avoid collisions on crates.io. But users of the facade don't need to care about that—they get clean, intuitive module names. This is API ergonomics in action.

See: Companion Code Section 3

---

## Section 3: Conditional Compilation with Feature Flags

One line stands out from the others: the PCRE2 re-export is guarded by `#[cfg(feature = "pcre2")]`. This is **conditional compilation**, one of Rust's most powerful features for managing optional dependencies. PCRE2 is a powerful regular expression library written in C, offering features that Rust's native regex engine doesn't support—particularly look-around assertions and backreferences.

But PCRE2 requires a C compiler and the PCRE2 library to be installed on the build system. By making it optional, ripgrep can be built on systems without these dependencies, and users who don't need PCRE2's advanced features don't pay the compilation or binary size cost. This is a perfect example of Rust's "you don't pay for what you don't use" philosophy extending to library design.

See: Companion Code Section 4

## Section 4: The Crate Dependency Graph

Understanding what each re-exported crate does illuminates ripgrep's architecture. The `matcher` crate defines the core trait that all regex implementations must satisfy. The `regex` crate

implements that trait using Rust's native regex engine. The `pcre2` crate implements it using the PCRE2 C library. The `searcher` crate handles the mechanics of reading files and applying matchers. The `printer` crate formats search results for output. And the `cli` crate provides utilities for command-line interface concerns.

This separation isn't arbitrary—it reflects ripgrep's design philosophy of composable, replaceable components. Want to use a different regex engine? Implement the `Matcher` trait. Want different output formatting? Write a new printer. The facade crate lets you pick and choose which components you need while still benefiting from the integration work that's been done.

See: Companion Code Section 5

---

## Section 5: The Builder Pattern Connection

This facade crate connects directly to the Builder Pattern ecosystem that runs throughout ripgrep. Each of the re-exported crates uses builders extensively: `SearcherBuilder` in the searcher crate, `StandardBuilder` in the printer crate, configuration builders in the regex crate, and `WalkBuilder` in the ignore crate. The facade doesn't re-export the ignore crate directly—that's a separate concern—but the pattern is consistent across the ecosystem.

The builders in these crates follow a common structure: they accumulate configuration through method chains, validate settings, and produce the configured object. By gathering all these crates under one umbrella, the facade makes it easier to see and use these related builders together. A typical usage might chain configuration across multiple builders before executing a search.

See: Companion Code Section 6

---

## Section 6: Why Not Just One Big Crate?

You might wonder why ripgrep is split into so many crates at all. Why not just have one library with everything in it? The answer involves compile times, modularity, and the ability to use pieces independently.

Rust's compilation unit is the crate, and crates can be compiled in parallel. By splitting functionality into separate crates with clear dependencies, ripgrep achieves faster incremental builds—changing the printer code doesn't require recompiling the searcher. Additionally, external users might only need part of the functionality. A tool that just needs fast regex matching might depend only on `grep_regex` and `grep_matcher`, skipping the printer and CLI utilities entirely.

The facade crate gives you the best of both worlds: the granularity exists for those who need it, but the unified interface exists for those who don't want to think about it.

See: Companion Code Section 7

---

## Key Takeaways

First, the facade pattern in Rust uses `pub extern crate` or `pub use` to present multiple crates as a unified API, shielding users from internal modular complexity while preserving the benefits of that modularity.

Second, conditional compilation with `#[cfg(feature = "...")]` enables optional dependencies that users can opt into, keeping the default build simple while supporting advanced use cases.

Third, crate naming conventions like `grep_searcher` for the package name and `searcher` for the re-export balance ecosystem collision avoidance with user-friendly APIs.

Fourth, the smallest files in a codebase are often the most architecturally significant—this twenty-line file defines how all of ripgrep's library code is organized and accessed.

Fifth, documentation honesty builds trust; explicitly stating limitations and planned improvements helps users make informed decisions about using your library.

---

## What to Read Next

How does the searcher actually coordinate file reading and pattern matching? Read `crates/searcher/src/searcher/mod.rs` to see the `SearcherBuilder` and understand how searches are configured and executed.

What's this `Matcher` trait that all regex implementations must satisfy? Read `crates/matcher/src/lib.rs` to understand the abstraction that enables regex engine interchangeability.

How do the printers format and output search results? Read `crates/printer/src/standard.rs` to see `StandardBuilder` and the machinery behind ripgrep's output formatting.

# ripgrep crates/grep/src/lib.rs: The Umbrella Crate

## What This File Does

This tiny file serves as the public gateway to ripgrep's searching functionality. Despite being only about twenty lines of actual code, it represents one of the most important architectural decisions in the project: the separation of ripgrep's core searching logic into a reusable library distinct from its command-line interface.

The `grep` crate acts as an umbrella crate—a single dependency that re-exports multiple specialized crates under one unified namespace. When other Rust projects want to embed ripgrep-style search capabilities, they don't need to figure out which of the six underlying crates to import or how they relate to each other. They simply depend on `grep` and get organized access to everything: matchers, searchers, printers, and both regex engines.

See: Companion Code Section 1

## The Umbrella Crate Pattern

Rust's module system encourages splitting large projects into multiple crates for several reasons: faster compilation, clearer boundaries, and independent versioning. But this splitting creates a discoverability problem for downstream users. If ripgrep's search functionality lived across six separate crates with names like `grep-matcher`, `grep-regex`, and `grep-searcher`, users would need to understand the entire architecture just to write their first search.

The umbrella crate pattern solves this by providing a curated entry point. The `grep` crate doesn't contain any searching logic itself—it's purely organizational. Each `pub extern crate` declaration takes an internal crate and re-exports it under a cleaner name. The `grep_cli` crate becomes accessible as `grep::cli`, and `grep_matcher` becomes `grep::matcher`. This transforms a flat collection of crates into a hierarchical namespace that communicates structure.

This pattern appears throughout the Rust ecosystem. The `tokio` runtime uses it to bundle I/O primitives, networking, and task scheduling. The `serde` ecosystem uses a variation where the main crate provides the traits while separate crates provide format implementations. What makes ripgrep's approach notable is how cleanly it separates the library from the application—the `rg` binary is just one possible consumer of the `grep` library.

See: Companion Code Section 2

---

## Module Documentation and the Honesty of Incomplete APIs

The doc comment at the top of this file contains something refreshing: honesty about limitations. It explicitly states that "there is no high level documentation available yet" and that "examples are sparse." A cookbook and guide are planned but not present.

This matters because documentation comments in Rust aren't just comments—they're compiled into the crate's documentation and become part of the public API's user experience. By acknowledging what's missing, the author sets appropriate expectations for anyone who encounters this library. They won't waste time searching for a guide that doesn't exist.

The comment also reveals the intended audience: developers who want "a high level facade to the crates that make up ripgrep's core searching routines." This tells us the library is meant to be usable without understanding the internal architecture, even though the documentation to achieve that goal remains aspirational.

See: Companion Code Section 3

---

## The Six Pillars of ripgrep Search

Let's examine what gets re-exported and why each piece exists. Understanding this taxonomy is essential before diving into any individual crate, because each one handles a specific responsibility in the search pipeline.

The `cli` module provides utilities for building command-line tools around search functionality. It handles things like detecting whether output is going to a terminal, setting up platform-specific behaviors, and parsing arguments. While named "cli," it's really about bridging the gap between search logic and the environment it runs in.

The `matcher` module defines the core abstraction for what it means to match text. This is deliberately separate from any specific regex implementation. It provides the `Matcher` trait that any pattern-matching backend must implement, enabling ripgrep to support multiple regex engines through a single interface.

See: Companion Code Section 4

## Regex Engines: Default and Optional

The `regex` module wraps Rust's standard `regex` crate, adapting it to implement the `Matcher` trait. This is the default engine that ripgrep uses for most searches. It's fast, safe, and handles the vast majority of patterns users throw at it.

The `pcre2` module provides an alternative engine based on the PCRE2 library. Notice the `#[cfg(feature = "pcre2")]` attribute—this is conditional compilation. The PCRE2 engine only gets included if the build explicitly enables the `pcre2` feature. This matters because PCRE2 is a C library requiring external dependencies, while the default regex engine is pure Rust.

Why support two engines? The regex crate prioritizes guaranteed performance—it won't compile patterns that could cause exponential blowup. PCRE2 supports more features like backreferences and look-around assertions, but without the same performance guarantees. By keeping both behind the same trait, ripgrep lets users choose their trade-off without changing the rest of the search pipeline.

See: Companion Code Section 5

---

## The Searcher and Printer: Orchestrating Results

The `searcher` module contains the actual search execution logic. Given a matcher and something to search (files, buffers, readers), it coordinates reading content, applying the matcher, and reporting results. This is where line-by-line iteration happens, where context lines before and after matches get tracked, and where memory mapping decisions get made.

The `printer` module handles output formatting. It knows about different output styles—standard grep output, JSON, summary statistics—and can colorize matches for terminal display. By separating printing from searching, ripgrep allows embedded uses to capture structured results rather than formatted text.

This separation illustrates a key design principle: each crate handles one concern. The matcher doesn't know about files. The searcher doesn't know about formatting. The printer doesn't know about regex syntax. Changes to output formatting can't break search logic, and new regex engines can't break printers.

See: Companion Code Section 6

---

## The pub extern crate Pattern

The syntax `pub extern crate grep_matcher as matcher` deserves closer attention because it's doing several things at once. In modern Rust (2018 edition and later), `extern crate` declarations are rarely needed—the compiler automatically finds dependencies listed in `Cargo.toml`. But when you want to re-export an entire crate under a new name as part of your public API, this syntax becomes essential.

The `pub` makes the re-export visible to downstream users. The `as matcher` renames `grep_matcher` to just `matcher` within the `grep` namespace. Without the `as` clause, users would need to write `grep::grep_matcher`, which defeats the purpose of providing a cleaner interface.

This pattern creates a stable public API that can evolve independently of internal crate names. If the maintainers decided to rename `grep_matcher` to `grep_pattern` internally, they could keep the re-export as `matcher` and downstream code would continue working.

See: Companion Code Section 7

---

## Conditional Compilation and Optional Dependencies

The `#[cfg(feature = "pcre2")]` attribute demonstrates Rust's compile-time feature system. Features are optional capabilities defined in `Cargo.toml` that downstream crates can enable or disable. When disabled, the code behind the `#[cfg]` attribute doesn't exist in the compiled binary—it's not just turned off, it's absent.

For PCRE2 support, this is crucial. PCRE2 requires native libraries that may not be available on all systems. By making it a feature, ripgrep can build and work perfectly on systems without PCRE2 development libraries installed. Users who need PCRE2's advanced regex features opt in explicitly and accept the additional build dependencies.

This pattern extends throughout professional Rust development. It balances capability with complexity, letting libraries provide advanced features without burdening users who don't need them.

See: Companion Code Section 8

## The Learning Path Through ripgrep's Crates

This umbrella crate reveals the learning path through ripgrep's architecture. The abstraction flows from abstract to concrete, from interface to implementation.

Start with `matcher` to understand what ripgrep considers "matching" to mean. The `Matcher` trait defines the contract that any pattern engine must fulfill. Then examine how `regex` and `pcre2` implement that contract differently—same interface, different capabilities and trade-offs.

Move to `searcher` to see how matching gets applied to real content. The `Sink` trait there defines how search results flow outward—another abstraction point that separates searching from result handling. Finally, `printer` shows one implementation of that sink, formatting results for human consumption.

This progression from abstraction to implementation, from interface to concrete type, represents the core architectural philosophy. Understanding it prepares you to read any part of the codebase with appropriate context.

See: Companion Code Section 9

---

## Key Takeaways

1. **First**, umbrella crates provide curated entry points to complex multi-crate projects, transforming flat dependency lists into hierarchical namespaces that communicate architecture.
2. **Second**, the `pub extern crate ... as` pattern re-exports dependencies under cleaner names, creating stable public APIs that can evolve independently of internal organization.
3. **Third**, conditional compilation with `#[cfg(feature = "...")]` enables optional functionality that only exists in binaries when explicitly requested, crucial for dependencies with external requirements.
4. **Fourth**, ripgrep's architecture separates concerns across crates: matching (pattern engines), searching (content traversal), and printing (output formatting) each have clear boundaries.
5. **Fifth**, supporting multiple regex engines behind a common trait lets users choose their trade-off between features and performance guarantees without changing application code.
6. **Sixth**, honest documentation about limitations builds trust and sets appropriate expectations—better to acknowledge missing guides than pretend they exist.

## What to Read Next

How does ripgrep define what "matching" means across different regex engines? Read `matcher/lib.rs` to see the `Matcher` trait that all pattern engines must implement.

How does the default regex engine adapt to this interface? Read `regex/matcher.rs` to see how Rust's regex crate gets wrapped to fulfill the `Matcher` contract.

What makes PCRE2 different enough to warrant a separate engine? Read `pcre2/matcher.rs` to understand the capability and performance trade-offs of the alternative backend.

How do search results flow from the searcher to consumers? Read `searcher/sink.rs` to understand the `Sink` trait that decouples searching from result handling.

# ripgrep crates/core/main.rs: Application Entry Point

## What This File Does

This file is the heart of ripgrep's executable—the place where all the architectural decisions manifest as actual program behavior. While the umbrella `grep` crate you studied previously defines the searching abstractions, this file orchestrates them into a complete command-line application. It handles everything from conditional memory allocator selection to parallel search coordination to graceful error handling.

What makes this file particularly instructive is how it demonstrates Rust's approach to building robust CLI applications. You'll see conditional compilation for platform-specific optimizations, the modern `ExitCode` return type for proper process termination, and a clear separation between argument parsing, mode selection, and execution. The file also showcases how ripgrep achieves its impressive performance through careful parallelization while maintaining correct output ordering and graceful degradation.

---

## Section 1: The Allocator Override

The file begins with a fascinating piece of conditional compilation that addresses a real-world performance problem. Rust applications use the system's default memory allocator unless you specify otherwise. On most Linux systems, that's glibc's allocator, which performs well for general workloads. But when ripgrep is compiled against musl—a minimal C library often used for static Linux binaries—the situation changes dramatically.

Musl's allocator prioritizes small code size and correctness over raw speed. This design choice makes perfect sense for musl's goals, but it can noticeably slow down ripgrep. The solution is to conditionally use jemalloc, a high-performance allocator, but only when targeting musl on 64-bit systems. The `#[cfg]` attributes create compile-time conditions that determine whether this code exists in the final binary at all.

This pattern demonstrates **conditional compilation at the deepest level**. The resulting binary literally has different allocation code depending on the build target. The `#[global_allocator]` attribute tells Rust to use this allocator for all heap allocations throughout the entire program. It's a

great example of how Rust lets you make sweeping architectural decisions that affect the entire runtime without cluttering your application logic.

See: Companion Code Section 1

---

## Section 2: The Main Function and Exit Code Philosophy

The `main` function returns `ExitCode`, which represents Rust's modern approach to process termination since version 1.61. Unlike returning `Result` from main (which prints debug output on error) or calling `std::process::exit` (which doesn't run destructors), `ExitCode` provides clean, explicit control over the exit status while allowing normal cleanup.

The main function's structure reveals ripgrep's error handling philosophy. It calls `run` with parsed arguments and matches on the result. Success propagates the exit code from the search logic. Errors get special handling—particularly broken pipe errors, which deserve graceful termination rather than error messages.

The broken pipe handling is worth studying closely. In Unix tradition, when you pipe output to a program that closes early (like `rg pattern | head -n 5`), the writing process receives a `SIGPIPE` signal. Traditional C programs die from this unhandled signal—ungraceful but silent. Rust doesn't register for `SIGPIPE` by default, so instead you get an I/O error. Ripgrep explicitly checks for this error kind and exits with success, matching Unix conventions while being more explicit about what's happening.

See: Companion Code Section 2

---

## Section 3: The Run Function and Mode Dispatch

The `run` function is ripgrep's central dispatcher, translating parsed arguments into appropriate behavior. It demonstrates a common Rust pattern: using enums to represent mutually exclusive modes of operation, then matching on those modes to select the right code path.

The `ParseResult` enum has three variants: errors that should be reported, special modes that bypass normal initialization, and successfully parsed arguments. This three-way split is intentional—special modes like `--help` and `--version` need to work even when other initialization might fail. If ripgrep can't access the current working directory, you still want `rg --help` to work.

The mode matching shows thoughtful optimization. Before dispatching to search functions, ripgrep checks `matches_possible()`. If the pattern can't possibly match anything (perhaps it's a

literal pattern that's longer than the maximum line length), there's no point searching. Then the thread count determines whether to use single-threaded or parallel search. This isn't just an optimization—sorting behavior depends on it. Sorted output disables parallelism because maintaining sort order across parallel workers would defeat the performance benefits.

See: Companion Code Section 3

---

## Section 4: Single-Threaded Search Architecture

The single-threaded `search` function reveals ripgrep's core search loop architecture. It's structured as a pipeline: build haystacks from directory traversal results, optionally sort them, then search each one sequentially. The term "haystack" is ripgrep's abstraction for "something that can be searched"—typically a file, but the abstraction allows for other sources.

The search worker encapsulates three components working together: a **matcher** (the pattern matching engine), a **searcher** (the line-oriented search logic from the `grep-searcher` crate), and a **printer** (the output formatter). This composition reflects the trait-based abstractions you'll study in the matcher learning thread. The matcher could be the standard regex engine or PCRE2; the searcher doesn't care because it works through the `Matcher` trait.

Error handling within the loop demonstrates pragmatic resilience. A broken pipe breaks the loop entirely—that's graceful termination. Other errors get logged but don't stop the search. If one file is unreadable, ripgrep continues with the rest. The `err_message!` macro handles thread-safe error output. Statistics accumulate across files if requested, and the `quit_after_match` flag enables early termination for `-q` (quiet) mode.

See: Companion Code Section 4

---

## Section 5: Parallel Search and Thread Coordination

The parallel search function showcases Rust's fearless concurrency in action. The `ignore` crate's parallel walker handles the threading complexity—ripgrep just needs to provide a closure that processes each discovered file. But coordinating output and statistics across threads requires careful synchronization.

The synchronization strategy uses different primitives for different needs. The `matched` and `searched` flags use `AtomicBool` with `SeqCst` ordering—the strongest memory ordering guarantee. This ensures that flag updates are immediately visible across all threads. Statistics use

a `Mutex` because they involve more complex updates than simple boolean flags. Output uses a buffered writer approach where each thread writes to its own buffer, then the completed buffer is printed atomically.

The closure structure deserves attention. Each call to `run` creates a fresh closure that captures references to the shared state. The closure itself is cloneable (notice `searcher.clone()`) because each thread needs its own search worker. The `Box::new(move |result| ...)` pattern creates a trait object that the parallel walker can invoke. The `WalkState` return type tells the walker whether to continue, skip the current directory, or quit entirely. A broken pipe during printing returns `WalkState::Quit`, stopping all threads gracefully.

See: Companion Code Section 5

---

## Section 6: The File Listing Modes

The `files` and `files_parallel` functions implement the `--files` flag, which lists files that would be searched without actually searching them. These functions mirror the structure of the search functions but are simpler since they don't need matchers or searchers—just path printing.

The parallel file listing introduces a channel-based architecture that differs from the search parallel approach. Instead of each thread handling its own output, all discovered paths are sent through a channel to a dedicated printing thread. The comment explains the rationale: this is "probably faster than using a mutex" though it's "never been seriously litigated." This kind of honest uncertainty in comments is valuable—it acknowledges that the design is reasonable but not proven optimal.

The channel approach prevents torn writes—partial lines interleaved from different threads. Each path write is atomic because it comes from a single thread. The main threads do the directory walking and send haystacks; the printing thread receives and writes them. When all senders are dropped (including the clones in the worker closures and the explicit `drop(tx)`), the channel closes and the printing thread's `iter()` loop terminates naturally.

See: Companion Code Section 6

## Section 7: Special and Generate Modes

The `special` and `generate` functions handle ripgrep's meta-modes—operations that produce information about ripgrep itself rather than searching files. Special modes are those that should

work even when normal initialization fails, like help and version output. Generate modes produce supplementary content like man pages and shell completions.

The special mode implementation highlights a key design insight noted in the doc comment: these modes "short-circuit most of ripgrep's initialization logic." If the current working directory is inaccessible, ripgrep's normal search initialization would fail. But you still want `rg --version` to work from anywhere. By detecting these modes early in argument parsing and routing them here, ripgrep maintains usability even in degraded environments.

The PCRE2 version mode is particularly interesting—it returns an error exit code if PCRE2 support isn't compiled in. This makes it usable in scripts that need to check for PCRE2 availability. The function signature returning `(String, bool)` for this variant is unusual compared to the others, reflecting its dual purpose of providing information and indicating availability.

See: Companion Code Section 7

---

## Section 8: The Types Listing Mode

The `types` function implements `--type-list`, which displays all known file type definitions. It's a simple loop over type definitions, but it demonstrates several Rust idioms for I/O-heavy code.

The function writes directly to stdout using byte slices rather than string formatting. This is slightly more efficient but more importantly shows the low-level I/O patterns that underpin ripgrep's output handling. The `write_all` method writes exact bytes, while the comma-joining logic uses a `first` flag pattern that's common in Rust for separating items without a trailing separator.

The exit code logic reflects Unix conventions: exit 0 if something was printed, exit 1 if the list was empty. An empty type list shouldn't happen in normal usage, but the handling makes the function robust against edge cases like a hypothetical build with all file types disabled.

See: Companion Code Section 8

---

## Section 9: Statistics Printing and Dual-Format Output

The `print_stats` function demonstrates how ripgrep handles its `--stats` flag with awareness of output format. When JSON output is requested, statistics must be JSON too—mixing formats would break pipeline consumers. The function checks the search mode and produces either plain text or a JSON summary message.

The JSON output deliberately matches the format used by the `grep-printer` crate's JSON printer. Ripgrep "extends" this format with a `summary` message type, maintaining consistency with the other JSON message types that appear during search. This design enables reliable parsing of ripgrep's complete JSON output stream—statistics are just another message type rather than a special case.

The timing calculation shows elapsed time from the start instant. The plain text format includes both search time (CPU time spent matching) and process time (wall clock from start). These can differ significantly—search time doesn't include directory traversal, I/O waiting, or output formatting. The JSON format includes only total elapsed time with both seconds and nanoseconds for precision, plus a human-readable string.

See: Companion Code Section 9

---

## Section 10: Helper Functions and User Communication

The `eprint_nothing_searched` function handles a specific UX problem: when ripgrep searches nothing and produces no output, users might think it's broken when actually their ignore rules are too aggressive. This helper prints a diagnostic message pointing users toward `--debug` mode.

The function is only called when the user provided an implicit path (relying on current directory default) and nothing was searched. If the user explicitly said "search this empty directory," they probably meant it. But if they just ran `rg pattern` and got nothing because `.gitignore` excluded everything, they need guidance.

This helper exemplifies ripgrep's attention to user experience. Many tools would silently succeed with no output, leaving users confused. Ripgrep proactively explains the likely cause and solution. The `err_message!` macro ensures this goes to stderr, keeping stdout clean for pipeline consumers.

See: Companion Code Section 10

## Key Takeaways

First, conditional compilation with `#[cfg]` and `#[global_allocator]` enables platform-specific optimizations without runtime overhead—the decision is made at compile time and affects the entire binary's behavior.

Second, the `ExitCode` return type from `main` represents modern Rust's approach to process termination, providing explicit control over exit status while preserving normal cleanup and destructor execution.

Third, broken pipe errors deserve special handling in CLI tools that might be piped to other programs—treating them as graceful termination rather than errors matches Unix conventions and user expectations.

Fourth, parallel code in Rust requires explicit synchronization primitives like `AtomicBool` and `Mutex`, but the type system ensures you can't accidentally share data unsafely—if it compiles, the synchronization is correct.

Fifth, separating special modes (help, version) from normal initialization ensures these critical commands work even when the environment is degraded, improving robustness for users in unusual situations.

Sixth, designing output formats for pipeline consumption—like ensuring JSON statistics match the JSON search output format—makes tools composable and scriptable, multiplying their utility.

---

## What to Read Next

How does the pattern matching work under the hood? Follow the Matcher Abstraction learning thread starting with `crates/matcher/src/lib.rs` to understand the `Matcher` trait that enables both the standard regex and PCRE2 backends.

What happens during argument parsing before `run` is called? Read `crates/core/flags/mod.rs` to see how ripgrep transforms command-line strings into the `HiArgs` structure used throughout this file.

How does the parallel directory walker coordinate threads? Explore the `ignore` crate to understand the `WalkBuilder` and `WalkParallel` types that power ripgrep's file discovery.

What makes the search worker composable? Read `crates/core/search.rs` to see how the matcher, searcher, and printer are combined into a single search operation.

# ripgrep crates/grep/examples/simplegrep.rs: Minimal Working Example

## What This File Does

This file is a teaching tool disguised as a program. In approximately sixty lines of code, it demonstrates how to combine the major components of ripgrep's grep crate into a functional search utility. Think of it as the "hello world" of ripgrep development—except instead of printing a greeting, it searches files for patterns.

What makes this example particularly valuable for learning is that it exposes the architecture you studied in Lesson 1. Where the facade pattern in `lib.rs` showed you *what* ripgrep exposes, this example shows you *how* those pieces fit together. Every component here—the matcher, the searcher, the printer—represents a carefully designed abstraction that you'll explore in depth throughout this course. This is your roadmap.

## Section 1: The Error Handling Entry Point

The program begins with a pattern you'll see throughout professional Rust codebases: a thin `main` function that delegates to a fallible `try_main`. This separation exists because `main` in Rust has limited return type options—it can return `()`, `ExitCode`, or a type implementing `Termination`. But error handling in real applications often needs the flexibility of `Result`.

By moving the actual logic to `try_main`, which returns `Result<(), Box<dyn Error>>`, the code gains access to Rust's `?` operator for early returns on errors. The `main` function then becomes a simple error reporter: if `try_main` fails, print the error and exit with code 1. This is idiomatic Rust error handling, and you'll find this exact pattern in ripgrep's actual binary.

The use of `Box<dyn Error>` is worth noting. This is the "easy path" for error handling—it accepts any type that implements the `Error` trait. In production code, you might define custom error types for better error messages and programmatic error handling. But for an example like this, the flexibility of `Box<dyn Error>` keeps the focus on the search logic rather than error plumbing.

See: Companion Code Section 1

## Section 2: Argument Parsing and Platform Safety

The `try_main` function handles command-line arguments with an interesting choice: it uses `env::args_os()` rather than `env::args()`. This distinction matters more than you might expect. The `args()` function returns `String` values, but it panics if any argument contains invalid UTF-8. On Unix systems, filenames can be arbitrary byte sequences—they don't have to be valid UTF-8. Using `args_os()` returns `OsString` values that can represent any platform-native string.

This is defensive programming at its finest. A search tool will inevitably encounter files with unusual names, and panicking on a directory traversal because of a filename encoding issue would be a poor user experience. The trade-off is slightly more complex string handling throughout the code, but it's the right choice for a file system tool.

The argument validation is minimal but functional: require at least a pattern, and default to searching the current directory if no paths are provided. This mirrors simplified versions of how Unix tools like grep behave—sensible defaults that let users type less while still being explicit about requirements.

See: Companion Code Section 2

---

## Section 3: Pattern Conversion with cli::pattern_from_os

Before the search can begin, the pattern argument needs conversion from `OsString` to a regular string slice. This is where `cli::pattern_from_os` enters the picture—a helper function from ripgrep's CLI utilities that handles this conversion.

Why is this a separate function rather than a simple conversion? Because the conversion can fail, and the failure mode matters. An `OsString` might contain bytes that aren't valid UTF-8, and regular expressions are defined in terms of UTF-8 strings. The `pattern_from_os` function performs this conversion and provides a meaningful error message if it fails, rather than leaving users wondering why their search didn't work.

This is also your first encounter with ripgrep's `cli` module in action. In Lesson 1, you saw that the facade exports this module. Here you see why—common operations like pattern conversion and output handling are shared infrastructure that both ripgrep itself and programs using the grep crate need.

See: Companion Code Section 3

---

## Section 4: The Matcher — Your First Builder Pattern

The `search` function begins by constructing a `RegexMatcher` using `new_line_matcher`. This particular constructor creates a matcher optimized for line-by-line searching—it ensures the pattern never matches across line boundaries. For a grep-like tool, this is usually what you want.

The `RegexMatcher` wraps ripgrep's regex implementation with an interface that the searcher understands. This abstraction is crucial: the searcher doesn't need to know about regex internals, and you could theoretically swap in a different matcher implementation (like a literal string matcher) without changing the searcher code.

Notice that the pattern is passed by reference to a string slice. The matcher takes ownership of the compiled pattern internally—what you're passing is just the source text that describes the pattern. If the pattern has syntax errors, `new_line_matcher` returns an error, which the `?` operator propagates back to the caller.

See: Companion Code Section 4

---

## Section 5: The Searcher Builder — Configuration Through Method Chaining

Here's your first substantial encounter with the **builder pattern**, one of the architectural themes you'll trace through ripgrep. The `SearcherBuilder::new()` call creates a builder with default settings, and then method calls like `binary_detection()` and `line_number()` modify those settings before `build()` produces the final `Searcher`.

The builder pattern solves a real problem in API design: how do you handle types with many optional configuration parameters? You could use a constructor with many arguments, but that's error-prone and hard to read. You could use a struct with public fields, but that exposes implementation details. The builder pattern gives you a fluent interface where each configuration option is self-documenting.

Two settings are configured here. First, `binary_detection(BinaryDetection::quit(b'\x00'))` tells the searcher to stop searching a file if it encounters a null byte—a strong indicator of binary content. Searching binary files is usually pointless and can produce garbled output, so this is a sensible default. Second, `line_number(false)` disables line number tracking, which slightly simplifies the output and improves performance when you don't need line numbers.

The searcher is declared `mut` because the search operation will modify internal state—buffers, position tracking, and similar bookkeeping. The builder itself is consumed by `build()`, following Rust's ownership conventions.

See: Companion Code Section 5

---

## Section 6: The Printer Builder — Terminal-Aware Output

The printer configuration demonstrates a more complex builder with conditional logic based on the environment. The `StandardBuilder` creates a printer that formats search results in the traditional grep style: filename, optional line number, and matching line content.

The `color_specs(ColorSpecs::default_with_color())` call sets up syntax highlighting for matches. But color output needs to know where it's going—terminals support ANSI color codes, but piping to a file or another program usually shouldn't include raw escape sequences.

This is where `std::io::stdout().is_terminal()` comes in. This method, stabilized in Rust 1.70, checks whether stdout is connected to an interactive terminal. If it is, `ColorChoice::Auto` lets the printer decide whether to use color based on its own heuristics. If stdout is a pipe or file, `ColorChoice::Never` ensures no escape sequences pollute the output.

The `cli::stdout()` function wraps the standard output handle in a way that the `termcolor` crate understands. This wrapper supports Windows console colors in addition to ANSI codes, making ripgrep's color output work correctly across platforms.

See: Companion Code Section 6

---

## Section 7: Directory Walking with WalkDir

The search loop begins by iterating over paths provided on the command line, then recursively walking each path with `WalkDir`. This is a separate crate from ripgrep's internal directory walking code, but it demonstrates the same concept: given a starting path, enumerate all files beneath it.

The outer loop handles multiple starting paths—you might run `simplegrep pattern src/ tests/` to search two directories. The inner loop, `WalkDir::new(path)`, produces an iterator of directory entries, traversing the directory tree depth-first by default.

Each iteration yields a `Result<DirEntry, walkdir::Error>`. Directory traversal can fail for many reasons: permissions denied, symbolic links that form cycles, file system errors. Rather than aborting the entire search on one bad directory, the code prints an error message and continues. This resilient error handling is important for real-world tools—you don't want a single unreadable file to halt a search across thousands of files.

The file type check `dent.file_type().is_file()` ensures we only try to search regular files, not directories, symbolic links, or other file system objects. Trying to "search" a directory would fail in confusing ways.

See: Companion Code Section 7

---

## Section 8: The Search Operation — Bringing Components Together

The actual search happens in one method call: `searcher.search_path()`. This single line is where all the components you've assembled finally work together. The method takes three arguments: the matcher, the path to search, and a "sink" that receives the results.

The sink is created by `printer.sink_with_path(&matcher, dent.path())`. This is an adapter pattern—it creates an object that implements the `Sink` trait expected by the searcher, but routes output through the printer with knowledge of the current file path. The matcher is passed again here because the printer needs it to highlight matches within lines.

Notice the error handling pattern: errors from `search_path` are caught and printed with the file path as context. This is crucial for debugging—if a file can't be searched (perhaps it's been deleted between the directory listing and the search attempt), the user needs to know which file caused the problem.

The searcher doesn't return the matches; it pushes them to the sink as they're found. This streaming architecture means ripgrep can display results immediately rather than buffering everything in memory. For large files or many files, this is a significant usability improvement.

See: Companion Code Section 8

## Section 9: The Component Architecture Visualized

Stepping back, this example reveals ripgrep's fundamental architecture: a pipeline of loosely coupled components. The data flows from files on disk, through the searcher which uses the matcher to identify matches, into a sink that formats and displays results.

Each component has a single responsibility. The matcher knows about patterns. The searcher knows about reading files and managing buffers. The printer knows about output formatting. None of them need to understand the others' internal details—they communicate through well-defined trait interfaces.

This separation is what makes ripgrep maintainable despite its complexity. When someone wants to add a new output format, they create a new printer implementation. When someone optimizes the regex engine, the searcher doesn't need to change. When someone adds memory-mapped file support, the printer doesn't care.

The builder pattern ties this together by giving each component a consistent, discoverable configuration interface. As you explore the files in the builder pattern learning thread—`searcher/mod.rs`, `printer/standard.rs`, `regex/config.rs`, `ignore/walk.rs`—you'll see this same architectural philosophy applied with much more sophisticated options.

See: Companion Code Section 9

## Key Takeaways

1. **First**, the pattern of separating `main` from a fallible `try_main` is idiomatic Rust for programs that need to handle errors gracefully while still using the `?` operator throughout their logic.
2. **Second**, using `OsString` instead of `String` for file paths and command-line arguments is defensive programming that prevents panics on valid but unusual file system names.
3. **Third**, the builder pattern provides a fluent, self-documenting API for configuring complex objects, and ripgrep uses it consistently across matchers, searchers, and printers.
4. **Fourth**, terminal detection with `is_terminal()` enables programs to behave appropriately whether their output goes to an interactive user or another program.
5. **Fifth**, the sink pattern enables streaming output—results appear immediately rather than being buffered—which is essential for responsive user experiences when searching large codebases.
6. **Sixth**, resilient error handling in file system operations means printing errors and continuing rather than aborting, because real-world directory trees are full of edge cases.

## What to Read Next

How does the searcher builder expand to support all of ripgrep's search options? Read `crates/grep-searcher/src/searcher/mod.rs` to see the full configuration surface.

What makes the printer configurable for different output formats? Read `crates/grep-printer/src/standard.rs` to understand how the standard printer handles colors, context lines, and formatting.

How does ripgrep's actual directory walking differ from the simple `WalkDir` used here? Read `crates/ignore/src/walk.rs` to discover how it handles gitignore patterns, parallel traversal, and file type filtering.

What other matcher implementations exist beyond `RegexMatcher`? Read `crates/grep-regex/src/config.rs` to see how regex matching is configured in the full implementation.

# ripgrep crates/core/logger.rs: Logging Infrastructure

## What This File Does

This file implements a minimal logging system for ripgrep that integrates with Rust's standard `log` crate. Rather than pulling in a full-featured logging framework with configuration files, multiple output targets, and complex filtering, ripgrep takes a refreshingly pragmatic approach: it defines just enough logging infrastructure to output debug messages to stderr during development and troubleshooting.

The logger demonstrates a common Rust pattern where the `log` crate provides the *interface* (the macros like `log!`, `debug!`, `trace!`) while applications provide their own *implementation* of where log messages actually go. This separation of concerns means ripgrep can use standard logging throughout its codebase without committing to any particular logging backend until the very end.

## Section 1: The Philosophy of Minimal Dependencies

The file opens with a module documentation comment that explains the design philosophy in refreshingly direct terms. The developers explicitly chose to avoid external dependencies for logging, even though dozens of logging implementations exist in the Rust ecosystem. This isn't accidental—it's a deliberate architectural decision.

Ripgrep as a project values startup time and binary size. Every dependency adds compilation time, potential security surface area, and cognitive overhead for contributors. For something as straightforward as "print debug messages to stderr," the cost of an external crate outweighs its benefits. The built-in implementation is around 50 lines of code, trivial to understand, and does exactly what ripgrep needs—nothing more.

This philosophy connects to what we saw in the main entry point. Ripgrep is careful about what it includes. The jemalloc allocator is conditional. The PCRE2 regex engine is optional. And the logger is hand-rolled. Each decision reflects a focus on keeping the core tool lean while allowing power users to opt into additional capabilities.

See: Companion Code Section 1

## Section 2: The Logger Struct and the Singleton Pattern

The `Logger` struct itself is almost comically simple: it's a unit struct containing an empty tuple. There's no configuration, no state, no fields at all. The struct exists solely to have a type that can implement the `Log` trait.

What's more interesting is how the logger is instantiated. Rather than creating new logger instances, the code defines a single static constant called `LOGGER`. This is Rust's approach to the singleton pattern—but without the complexity and runtime overhead you'd see in languages with mutable singletons. The `&'static Logger` type means this is a reference with a `'static` lifetime, guaranteed to be valid for the entire duration of the program.

The `pub(crate)` visibility modifier on the struct tells us this logger is internal to the core crate. It's not part of ripgrep's public API. External code that wants to observe ripgrep's behavior would need to use different mechanisms. This is private infrastructure, not a reusable library component.

See: Companion Code Section 2

---

## Section 3: Initialization and the Log Crate's Global State

The `init` function shows how Rust's `log` crate expects loggers to be registered. The `log::set_logger` function takes a reference to something implementing `Log` and registers it as the global logger for the entire application. This is a one-time operation—you can only set the logger once, and attempting to set it again returns an error.

The return type `Result<(), log::SetLoggerError>` signals that initialization can fail, but the only failure mode is "a logger was already set." In practice, ripgrep calls this early in startup before any parallel code runs, so the error case represents a programming mistake rather than a runtime condition.

This function doesn't set the log *level*—that happens elsewhere via `log::set_max_level`. This separation might seem redundant, but it allows the filtering threshold to be adjusted independently of which logger handles the output. Ripgrep sets the log level based on command-line flags, keeping this logger focused purely on the output mechanism.

See: Companion Code Section 3

---

## Section 4: Implementing the Log Trait

The `Log` trait from the `log` crate defines three methods that any logging backend must implement: `enabled`, `log`, and `flush`. This is the **trait-based abstraction** pattern we've been tracking through the codebase, though in a simpler form than the `Matcher` trait hierarchy.

The trait acts as a contract between the `log` crate's macros (which are used throughout code) and the actual logging implementation (which is provided once at startup). When code anywhere in ripgrep calls `debug!("searching {}", path)`, that macro eventually calls through to these trait methods on whatever logger was registered.

This design means ripgrep's search code doesn't know or care where log messages go. The code uses generic logging macros, and the core binary's logger implementation decides that "where" is stderr. You could theoretically swap in a different logger that writes to a file or sends messages over the network, and none of the search code would need to change.

See: Companion Code Section 4

---

## Section 5: The enabled Method and Deferred Filtering

The `enabled` method returns `true` unconditionally, which might seem like a bug or oversight. Shouldn't a logger decide which messages it wants to handle? The answer lies in understanding the `log` crate's two-tier filtering system.

The `log` crate maintains a global maximum log level. When code calls `debug!(...)`, the macro first checks this global level. If debug logging is disabled, the macro short-circuits immediately—the message string isn't even formatted, and the logger's methods are never called. This filtering happens at the call site, before any logger involvement.

By returning `true` unconditionally in `enabled`, ripgrep's logger says "if a message made it past the global filter, I'll handle it." This is the simplest correct implementation. A more sophisticated logger might implement per-module filtering or dynamic level changes, but ripgrep doesn't need those features. The comment in the code explicitly acknowledges this design choice, making the intent clear to future maintainers.

See: Companion Code Section 5

## Section 6: The log Method and Pattern Matching on Optional Data

The `log` method is where messages actually get written, and it showcases Rust's pattern matching on `Option` types. A log record might or might not include the source file and line number where the log macro was called. Rather than using nested `if let` statements or unwrapping with defaults, the code matches on a tuple of both optional values simultaneously.

This tuple-matching pattern is idiomatic Rust for handling multiple optional values when you want different behavior for different combinations. The `(Some(file), Some(line))` arm handles the case where both are available—the most informative output. The `(Some(file), None)` arm handles having a file but no line number. And the catch-all `_` arm handles everything else, including cases where neither is available.

The format strings show the output structure: level, target (usually the module path), optional file and line, then the message itself. This pipe-delimited format makes log output grep-able—appropriately enough for a tool called ripgrep.

See: Companion Code Section 6

---

## Section 7: The eprintln_locked Macro

The code uses `eprintln_locked!` rather than the standard library's `eprintln!` macro. This custom macro isn't defined in this file—it comes from elsewhere in the core crate—but its use here is significant for understanding ripgrep's output handling.

Standard `eprintln!` acquires a lock on stderr, writes, and releases the lock on each call. When multiple threads are logging simultaneously, their messages can interleave at line boundaries. The `_locked` variant likely holds the lock for the entire write operation, ensuring each log message appears as a complete unit.

This connects to ripgrep's parallel search architecture. Multiple worker threads might be searching different files simultaneously, and each might generate log messages. Without proper synchronization, debugging output would become an unreadable jumble. The locked macro ensures that even under heavy parallelism, each log line remains intact.

See: Companion Code Section 7

## Section 8: The flush Method and Write Semantics

The `flush` method has an empty body with a comment explaining why: the `eprintln_locked!` macro flushes on every call. This is important context about Rust's I/O model.

Buffered I/O improves performance by batching small writes into larger ones, but it means data might not reach its destination immediately. For stderr output that a developer is watching in real-time, this delay would be confusing. You'd see log messages appear in bursts rather than as they occur.

The `log` crate calls `flush` when it wants to ensure all pending messages have been written—typically before the program exits or when a critical error occurs. By documenting that flushing happens implicitly on every write, the code explains why an explicit flush operation is unnecessary. This is documentation-as-code: the comment prevents future maintainers from "fixing" the seemingly empty method.

See: Companion Code Section 8

---

## Section 9: Debug Derive and Interior Mutability Absence

The `#[derive(Debug)]` attribute on the `Logger` struct enables automatic implementation of the `Debug` trait. This might seem pointless for a unit struct with no fields—what is there to debug-print? But it's a good habit that costs nothing and enables better error messages if the logger ever appears in a debug context.

More interesting is what's *absent* from this logger: any form of interior mutability. There's no `Mutex`, no `AtomicBool`, no `RefCell`. The logger is truly stateless. This means it's automatically thread-safe with no synchronization overhead. Multiple threads can call `log` simultaneously, and each call is independent.

This statelessness is only possible because ripgrep's logger delegates all the actual state to other components: the `log` crate holds the filter level, stderr itself holds the output buffer (managed by `eprintln_locked!`), and the operating system handles the actual I/O. The logger is pure behavior with no stored data.

See: Companion Code Section 9

## Section 10: Connecting to the Trait Abstraction Pattern

Though simpler than the matcher hierarchy we've been studying, this logger exemplifies the same **trait-based abstraction** pattern. The `log` crate defines a `Log` trait as the interface. Ripgrep provides one implementation. Other parts of the codebase use the interface without knowing the implementation details.

Compare this to the matcher system: `grep-matcher` defines the `Matcher` trait, `grep-regex` and `grep-pcre2` provide implementations, and `grep-searcher` uses the trait generically. The pattern scales from this tiny logger (one trait, one implementation, 50 lines) to the complex search infrastructure (multiple traits, multiple implementations, thousands of lines).

The key insight is that Rust's traits enable this abstraction without runtime overhead. There's no virtual method table lookup when logging—the compiler knows at compile time that `LOGGER` is a `Logger`, so calls to trait methods are direct function calls. You get the software engineering benefits of abstraction with the performance characteristics of concrete code.

See: Companion Code Section 10

---

## Key Takeaways

1. **First**, sometimes the best dependency is no dependency at all. Ripgrep's hand-rolled logger is trivial to understand, adds no external crates, and does exactly what's needed—nothing more.
2. **Second**, Rust's `log` crate separates the logging *interface* (the macros you call) from the logging *implementation* (where messages go). This lets libraries log without committing to a particular backend.
3. **Third**, the singleton pattern in Rust uses static constants rather than mutable global state. The `&'static` lifetime guarantees the reference is valid forever, with no runtime initialization cost.
4. **Fourth**, returning `true` from `enabled` isn't lazy—it's correct delegation. The `log` crate's global level filter handles message filtering before the logger is ever consulted.
5. **Fifth**, pattern matching on tuples of `Option` values is idiomatic Rust for handling multiple optional fields with different behavior for different combinations.
6. **Sixth**, a stateless struct can implement traits that describe behavior. The `Logger` has no fields but still implements `Log` because all required state lives elsewhere (in the `log` crate and in stderr itself).

## What to Read Next

How does ripgrep parse and validate command-line arguments before the logger is initialized? Read `crates/core/args.rs` to see how the `--debug` flag sets the log level.

What is `eprintln_locked!` and how does it ensure thread-safe output? Look at `crates/core/messages.rs` where the output macros are defined.

How does the parallel search architecture use logging for debugging? The searcher code in `grep-searcher` uses the `log` crate's macros, which flow through this logger when debugging is enabled.

How does the `Matcher` trait hierarchy we've been tracking compare to this simpler `Log` trait? Review `grep-matcher/src/lib.rs` to see the more complex trait-based abstraction that ripgrep uses for its core search functionality.

# ripgrep crates/matcher/src/lib.rs: The Matcher Trait

## What This File Does

This file defines the foundational abstraction that makes ripgrep's modular architecture possible: the `Matcher` trait. At its core, this trait provides a generic interface for text search that decouples the search algorithm from the pattern matching implementation. This means ripgrep's search infrastructure—the code that reads files, handles line boundaries, manages output—doesn't need to know whether you're using a simple substring search, a basic regex, or a complex PCRE2 pattern with lookahead assertions.

The design here reflects a mature understanding of systems programming trade-offs. Rather than exposing an iterator-based API that Rust developers might expect, the trait uses **internal iteration** (also called the "push" model), where the matcher drives the search and calls back into user-provided closures. This choice, while less ergonomic on the surface, enables maximum performance across wildly different regex implementations and avoids boxing or dynamic dispatch in hot paths.

See: Companion Code Section 1

---

## Section 2: The Match Type - A Better Range

Before examining the trait itself, we need to understand the `Match` type—a deceptively simple struct that represents the byte offsets of a matched region. You might wonder why ripgrep defines its own type instead of using `std::ops::Range<usize>`. The answer lies in Rust's design philosophy around **newtypes** and **invariants**.

A `Match` enforces that `start <= end` at construction time through an assertion in `new()`. This invariant can never be violated without unsafe code, which means every function that receives a `Match` can trust this property without checking. The standard `Range` type makes no such guarantee—you can construct `5..3` without any error.

Beyond the invariant, `Match` implements `Copy`, which `Range` does not. This seemingly minor difference has significant ergonomic and performance implications. Copying a `Match` is always

cheap (it's just two `usize` values), and you never need to clone or borrow it. The helper methods like `with_start()`, `with_end()`, and `offset()` return new `Match` values rather than mutating, enabling a functional style that's both readable and efficient.

The `Index` trait implementations allow you to use a `Match` directly as a slice index, bridging the gap between this custom type and Rust's standard slicing syntax. This pattern—creating domain-specific types that integrate smoothly with standard library traits—is something you'll see throughout ripgrep.

See: Companion Code Section 2

---

## Section 3: Line Terminators and Cross-Platform Reality

The `LineTerminator` type reveals how ripgrep handles one of computing's most persistent compatibility headaches: different platforms use different conventions for marking the end of a line. Unix systems use `\n`, while Windows traditionally uses `\r\n` (CRLF). A grep tool that ignores this reality will produce confusing results for a significant portion of its users.

The implementation uses a private enum `LineTerminatorImp` wrapped in a public struct `LineTerminator`. This pattern, common in Rust library design, hides the representation while exposing a clean API. Users construct line terminators through factory methods (`byte()` and `crlf()`) rather than directly accessing the enum variants.

The `as_byte()` method deserves special attention. When dealing with CRLF, it returns just `\n`. This might seem like a bug, but it's actually a pragmatic design decision documented in the method. Many line-oriented algorithms can treat `\n` as the significant byte even in CRLF mode—the `\r` is essentially optional padding. This simplifies implementations while still handling Windows files correctly.

See: Companion Code Section 3

---

## Section 4: ByteSet - Optimization Through Constraint

The `ByteSet` type represents a set of bytes that can never appear in any match. At first glance, this might seem like an obscure optimization detail, but it's actually central to how ripgrep achieves its speed on line-oriented searches.

Consider searching for a pattern that cannot match across lines—perhaps a simple literal like `"error"`. If the matcher can communicate that `\n` will never appear in a match, the search

infrastructure can use this information to skip entire lines after finding a match, or to choose a faster algorithm that doesn't need to handle multi-line cases.

The implementation uses a compact bit vector: four `u64` values provide 256 bits, one for each possible byte value. The bit manipulation is straightforward—dividing by 64 selects which `u64` to use, and the remainder selects which bit within that `u64`. This gives O(1) insertion, removal, and membership testing with minimal memory overhead (just 32 bytes for the entire set).

The `Debug` implementation for `BitSet` is worth noting. Rather than printing the raw bit patterns, it iterates through all 256 possible bytes and prints only those that are present. This transforms an unreadable array of hex values into something a human can actually understand during debugging.

See: Companion Code Section 4

---

## Section 5: The Captures Trait - Abstracting Group Extraction

Capture groups are a powerful regex feature that let you extract specific portions of a match. The pattern `(\d{4})-(\d{2})-(\d{2})` matching against `"2024-01-15"` doesn't just tell you "there's a match"—it lets you extract the year, month, and day separately.

The `Captures` trait abstracts over different implementations of capture group storage. Different regex engines have different internal representations—some might use a `Vec<Option<Match>>`, others might use a more compact representation. By defining a trait rather than a concrete type, ripgrep allows each matcher implementation to use whatever representation is most efficient.

The `interpolate` method is particularly interesting. It handles the common case of replacement strings like `$1` or `${name}` that reference captured groups. Rather than requiring each matcher to implement this logic, it's provided as a default implementation on the trait, delegating to a separate `interpolate` module for the actual parsing and substitution. This follows the principle of putting shared functionality in the trait's default methods.

The `NoCaptures` type provides an escape hatch for matchers that don't support capture groups. Rather than forcing every matcher to implement capture functionality, they can simply use `NoCaptures` as their `Captures` associated type. This is a form of the **null object pattern**—providing a valid but empty implementation that satisfies the interface without doing useful work.

See: Companion Code Section 5

---

## Section 6: The NoError Type - When Failure Is Impossible

Some matcher implementations can never fail. A simple substring search, for instance, just compares bytes—there's no I/O, no memory allocation, no possibility of runtime errors. Yet the `Matcher` trait needs an associated `Error` type.

The `NoError` type solves this elegantly. It's an empty struct (containing only a unit value `()`) that implements `std::error::Error` and `Display`. But here's the clever part: the `Display` implementation panics. This isn't a bug—it's intentional. If you're using `NoError` correctly, you'll never actually have an error value to display. If somehow you do have one, that's a bug in your code, and panicking surfaces it immediately rather than silently producing garbage output.

The `From<NoError> for std::io::Error` implementation similarly panics. This allows `NoError` to satisfy trait bounds that require error conversion, while ensuring that any actual use would be caught as a programming error.

In the future, Rust's "never type" (`!`) will provide a more elegant solution—a type that can never be instantiated at all. Until that stabilizes, `NoError` serves the same purpose with runtime rather than compile-time enforcement.

See: Companion Code Section 6

---

## Section 7: The Matcher Trait - Core Design

Now we arrive at the heart of this file: the `Matcher` trait itself. The trait has two required methods (`find_at` and `new_captures`) and a large collection of provided methods that build on these foundations. This design follows the **minimal interface principle**—require implementations to provide only what's absolutely necessary, then offer convenience methods as defaults.

The choice of `find_at` as the primitive rather than `find` is significant. The `at` parameter specifies where in the haystack to start searching, and crucially, this affects context-sensitive patterns. A regex like `\Astart` (which matches "start" only at the beginning of the input) should only match when `at == 0`. If we only had `find`, implementing this correctly would require slicing the haystack, which would destroy this context information.

The associated types `Captures` and `Error` make the trait generic over both the capture group representation and the error type. This flexibility comes at a cost—the trait cannot be made into a trait object (`dyn Matcher`) because the associated types aren't fixed. For ripgrep, this is an acceptable trade-off because the matcher type is typically known at compile time through generic parameters.

See: Companion Code Section 7

## Section 8: Internal Iteration - The Push Model

The iteration methods (`find_iter`, `find_iter_at`, and their `try_` variants) demonstrate the internal iteration pattern. Instead of returning an iterator that the caller pulls values from, these methods accept a closure and call it repeatedly for each match.

The default implementation of `try_find_iter_at` reveals the complexity hidden behind this simple interface. It must handle empty matches specially—a regex like `a*` matching against `"bbb"` would find an empty match at every position, potentially causing an infinite loop. The implementation tracks `last_end` and `last_match` to ensure progress: after an empty match, it advances by one byte, and it skips empty matches that occur immediately after a previous match.

The nested `Result<Result<(), E>, Self::Error>` return type might look intimidating, but it represents a clean separation of concerns. The outer `Result` handles errors from the matcher itself (like a regex engine encountering invalid state), while the inner `Result` handles errors from the user-provided callback. This allows the caller to distinguish between "the search failed" and "my callback failed."

Why internal iteration? The module documentation explains: some regex implementations are inherently push-based, and converting them to pull-based iterators ranges from difficult to impossible without significant overhead. By choosing the lowest common denominator, ripgrep can support any regex implementation efficiently.

See: Companion Code Section 8

---

## Section 9: Capture Group Methods

The capture-related methods mirror the find methods but populate a `Captures` value with group information. The pattern is consistent: `captures` delegates to `captures_at`, and `captures_iter` builds on `captures_iter_at`.

Notice that `captures_at` has a default implementation that always returns `Ok(false)`, indicating no match. This might seem wrong—shouldn't it call `find_at` and at least report the overall match? The documentation clarifies the intent: matchers that support captures must override this method. The default exists for matchers that don't support captures at all, where the sensible behavior is "capturing never succeeds."

This design puts the burden on implementors to remember to override `captures_at` if they support captures, but it avoids the alternative of having a default that would be wrong for capture-supporting matchers. It's a pragmatic choice that trades a potential implementor mistake (forgetting to override) against a guaranteed runtime bug (wrong default behavior).

See: Companion Code Section 9

---

## Section 10: Replacement Operations

The `replace` and `replace_with_captures` methods provide find-and-replace functionality. The implementation pattern is elegant: iterate over matches, copy the non-matching text between matches to the destination buffer, and let the user-provided closure handle the replacement text for each match.

The closure receives both the `Match` (or `Captures`) and a mutable reference to the destination buffer. This design gives the caller complete control over replacement content without requiring allocation for each replacement string. The caller can append literal bytes, format numbers, perform conditional logic—whatever the use case demands.

The closure returns `bool`, allowing the caller to stop replacement early. This might be useful for implementing a "replace first N matches" feature or stopping when the output buffer reaches a certain size.

See: Companion Code Section 10

---

## Section 11: Line-Oriented Searching

The final group of methods relates to line-oriented searching, which is ripgrep's primary use case. The `line_terminator` method lets a matcher declare what byte (or byte sequence) marks line boundaries, while `non_matching_bytes` provides the optimization hint discussed earlier.

The `find_candidate_line` method introduces a fascinating optimization opportunity. It can return `LineMatchKind::Confirmed` when it's certain a match exists, or `LineMatchKind::Candidate` when the line might contain a match but needs verification.

Why would you return a candidate instead of a confirmed match? Consider a complex regex like `\b\w+error\w+\b`. A matcher might quickly scan for the literal `error` using optimized substring search (like SIMD-accelerated `memmem`), then return those lines as candidates. The caller then runs

the full regex only on candidate lines. For patterns where the literal is rare, this can be dramatically faster than running the full regex on every line.

The default implementation simply delegates to `shortest_match`, always returning `Confirmed` or nothing. Matchers that can't benefit from candidate-based optimization don't need to do anything special.

See: Companion Code Section 11

---

## Section 12: Blanket Implementation for References

The file concludes with an implementation of `Matcher` for `&'a M` where `M: Matcher`. This blanket implementation lets you use a reference to a matcher anywhere a matcher is expected, which is essential for sharing a single matcher across multiple search operations without moving it.

Every method simply dereferences `self` and delegates to the underlying matcher. This is pure boilerplate, but necessary boilerplate—Rust doesn't automatically implement traits for references. The `#[inline]` attributes hint to the compiler that these trivial delegations should be eliminated entirely in release builds.

This pattern appears frequently in Rust libraries. The standard library does the same thing for traits like `Read` and `Write`. It's a form of **ergonomic tax**—code that exists purely to make the library more pleasant to use, with no semantic content of its own.

See: Companion Code Section 12

---

## Key Takeaways

First, **internal iteration trades ergonomics for universality**. By having matchers push results through callbacks rather than exposing iterators, ripgrep can support any regex implementation regardless of its internal architecture.

Second, **newtypes with invariants prevent bugs**. The `Match` type's guarantee that `start <= end` eliminates an entire class of potential errors at the type level.

Third, **associated types enable zero-cost abstraction**. The `Captures` and `Error` types let each matcher use its optimal representation without boxing or dynamic dispatch.

Fourth, **default trait methods reduce implementor burden**. Only `find_at` and `new_captures` are required; everything else builds on these foundations.

Fifth, **the candidate/confirmed distinction enables optimization**. Line-oriented matchers can use fast literal search as a pre-filter, only running expensive regex matching on promising candidates.

Sixth, **null object types like `NoCaptures` and `NoError` satisfy interfaces cleanly**. They provide valid implementations for matchers that don't need certain features.

---

## What to Read Next

How does a real regex implementation satisfy this trait? Read `crates/regex/src/matcher.rs` to see `Matcher` implemented for ripgrep's primary regex engine.

How does the searcher use these abstractions? Read `crates/searcher/src/searcher/mod.rs` to see how the `Matcher` trait integrates with file reading and line handling.

What patterns does the `grep` facade expose to users? Revisit `crates/grep/src/lib.rs` to see how these low-level traits become a convenient API.

How do builders configure matchers? Read `crates/regex/src/config.rs` to see the builder pattern applied to regex construction, connecting to the broader builder ecosystem across ripgrep.

# ripgrep crates/matcher/src/interpolate.rs: Capture Group Interpolation

## What This File Does

This file implements capture group interpolation—the mechanism that powers search-and-replace functionality in ripgrep. When you run a command like `rg 'foo(\d+)' --replace 'bar$1'`, this code is responsible for parsing that `$1` reference, looking up what the first capture group actually matched, and constructing the final replacement string. It's a focused, self-contained module that handles the surprisingly tricky task of parsing replacement strings that contain variable references.

The interpolation system supports multiple reference syntaxes: numeric references like `$1`, named references like `$name`, and braced variants like `${1}` or `${name}` for disambiguation. It also handles escape sequences (`$$` becomes a literal `$`) and gracefully ignores malformed references. This flexibility mirrors what users expect from tools like sed or Perl, making ripgrep feel familiar while maintaining Rust's emphasis on correctness and performance.

## Section 1: The Interpolation Function Signature

The `interpolate` function exemplifies a powerful Rust pattern: dependency injection through closures. Rather than taking concrete types for looking up capture groups and writing results, it accepts two closure parameters that abstract these operations. This design decouples the interpolation logic from any specific regex engine or output mechanism.

Look at the function signature: it takes `append` (a closure that writes captured text given an index) and `name_to_index` (a closure that resolves named groups to indices). The caller provides the implementation details, while this function handles the parsing logic. This is why the same interpolation code works whether you're using Rust's `regex` crate, PCRE2, or any other matcher implementation.

The trait bounds `A: FnMut(usize, &mut Vec<u8>)` and `N: FnMut(&str) -> Option<usize>` tell us exactly what capabilities these closures need. They're `FnMut` rather than `Fn` because they might need to mutate internal state—perhaps caching lookup results or tracking statistics. The choice of `&mut Vec<u8>` for output rather than returning strings keeps allocations minimal and gives callers control over buffer management.

See: Companion Code Section 1

---

## Section 2: Efficient Scanning with memchr

The interpolation loop demonstrates a common high-performance pattern in ripgrep: using the `memchr` crate for fast byte searching. Rather than iterating byte-by-byte through the replacement string, the code jumps directly to each `$` character using SIMD-optimized searching when available on the platform.

The loop structure is elegant. When `memchr` finds a `$`, everything before it is literal text that gets copied directly to the output. The slice `replacement` is then narrowed to start at the `$`, and parsing logic takes over. When no more `$` characters exist, the loop breaks and any remaining literal text is appended.

This "scan and narrow" approach avoids the need for explicit index tracking. Instead of maintaining a position variable that could get out of sync, the code keeps shrinking the `replacement` slice. Each iteration works with a slice that starts at the next potential reference. This is idiomatic Rust: using slice operations to maintain invariants rather than manual bookkeeping.

See: Companion Code Section 2

---

## Section 3: Escape Sequence Handling

Before attempting to parse a capture reference, the code handles the escape sequence `$$`. This ordering matters—we need to catch escaped dollar signs before we try to interpret them as reference starts. The check uses the pattern `replacement.get(1).map_or(false, |&b| b == b'$')`, which safely handles the case where `$` appears at the very end of the string.

The `get` method returns an `Option<&u8>`, and `map_or` provides a default value of `false` if the index is out of bounds. This is more idiomatic than checking the length first—it combines the bounds check and the value check into a single expression. When we find `$$`, we push a single `$` to the output and advance past both characters.

This pattern of "check for special case, handle it, continue" appears throughout the function. Each `continue` statement restarts the loop, which will either find the next `$` via memchr or exit when the slice is exhausted. The explicit `continue` makes the control flow clear: we've handled this case completely, move on.

See: Companion Code Section 3

---

## Section 4: The CaptureRef and Ref Types

The `CaptureRef` and `Ref` types model the result of parsing a capture reference. This is a case where Rust's type system helps document the code's intent. A capture reference can be either a number (like `$1`) or a name (like `$foo`), and the `Ref` enum makes this distinction explicit rather than using a stringly-typed approach.

Notice the lifetime parameter on `CaptureRef<'a>`. The `Ref::Named` variant borrows a string slice from the input, avoiding allocation during parsing. This is a zero-copy design: we're identifying where in the input the name lives, not copying the name into a new string. The lifetime ties the `CaptureRef` to the input slice, preventing use-after-free bugs at compile time.

The `end` field in `CaptureRef` tells the caller how many bytes the reference consumed. This is crucial for advancing through the input—after processing a reference like `${foo}`, we need to know to skip 6 bytes, not 4. Returning this information alongside the parsed reference keeps the parsing and consumption logic decoupled.

See: Companion Code Section 4

## Section 5: The From Trait Implementations

The `From` trait implementations for `Ref` serve the test infrastructure but also demonstrate a broader Rust pattern. By implementing `From<&str>` and `From<usize>`, the code enables convenient construction of `Ref` values using `.into()`. The test macro `c!` exploits this to create `CaptureRef` values concisely.

Notice the difference in lifetimes between the two implementations. `From<&'a str>` produces a `Ref<'a>` because the named reference borrows from the input string. But `From<usize>` produces a `Ref<'static>` because numeric references don't borrow anything—they're self-contained values. This distinction is subtle but correct: a number doesn't have a lifetime dependency.

The `#[inline]` attributes on these tiny functions hint to the compiler that it should inline them at call sites. For such trivial conversions, the function call overhead would exceed the work done. These hints don't guarantee inlining—the compiler makes the final decision—but they express the author's intent.

See: Companion Code Section 5

---

## Section 6: Parsing Capture References

The `find_cap_ref` function is where the actual parsing happens. It handles three syntactic forms: `$N` for numeric references, `$name` for named references, and `${...}` for braced references of either type. The braced form exists for disambiguation—`$1test` could mean "group 1 followed by test" or "a group named 1test", while `${1}test` is unambiguous.

The parsing proceeds in stages: check for the leading `$`, optionally consume an opening brace, scan valid identifier characters, then optionally require a closing brace. Each stage either advances the position or returns `None` to indicate parsing failure. The `map_or` pattern appears again for safe bounds-checked access.

The comment about UTF-8 validity is instructive. The `is_valid_cap_letter` function only accepts ASCII alphanumerics and underscore, so we know the captured slice is valid ASCII and therefore valid UTF-8. The `from_utf8` call will never fail, but Rust requires us to handle the theoretical error case. The `expect` documents this invariant while satisfying the type system.

See: Companion Code Section 6

---

## Section 7: Number vs Name Disambiguation

After extracting the identifier portion of a capture reference, the code must decide whether it's a number or a name. The approach is pragmatic: try to parse it as a `u32`, and if that fails, treat it as a name. This means `$42` is a numeric reference, but `$42a` is a named reference—the trailing letter makes it unparseable as a number.

The cast from `u32` to `usize` deserves attention. Capture group indices are logically `usize` (the type used for indexing in Rust), but parsing to `u32` first prevents accepting astronomically large numbers on 64-bit systems. A replacement string with `$18446744073709551615` would parse successfully as `usize` on 64-bit platforms, potentially causing confusion or resource exhaustion. Limiting to `u32` keeps the indices reasonable.

This defensive parsing is characteristic of ripgrep's attention to edge cases. User input from the command line can't be trusted, and replacement strings are a form of user-provided code. By constraining the parsed values, the code prevents a class of potential issues without complicating the API for normal use.

See: Companion Code Section 7

---

## Section 8: Character Validation

The `is_valid_cap_letter` function defines what characters can appear in capture group names. It accepts ASCII digits, lowercase letters, uppercase letters, and underscore—a conservative set that matches most regex engines' expectations for group names. The match expression compiles to a simple comparison chain, likely optimized to a few compare-and-branch instructions.

The function takes `&u8` rather than `u8` to match the return type of slice methods like `get`. This avoids a dereference at each call site when used with `.map_or(false, is_valid_cap_letter)`. It's a small ergonomic choice, but in hot loops these details matter.

This validation function also implicitly defines the end of a capture reference. When scanning `$foo123bar`, the function returns `true` for all characters, so the entire `foo123bar` is consumed as the name. But for `$foo.bar`, the dot fails validation, stopping the scan at `foo`. This is why the braced syntax exists—`${foo}.bar` clearly delimits the reference.

See: Companion Code Section 8

---

## Section 9: Handling Parse Failures Gracefully

Back in the main `interpolate` function, notice how parse failures are handled. When `find_cap_ref` returns `None`, the code doesn't panic or return an error. Instead, it pushes a literal `$` to the output and advances by one byte. The user wrote something that looked like it might be a reference but wasn't—perhaps `$` followed by whitespace or punctuation. Treating this as literal text is the least surprising behavior.

This graceful degradation philosophy pervades ripgrep. A replacement string of `$$$` produces `$` followed by attempting to parse a reference at the second `$`, which fails, producing another literal `$`, leaving a trailing `$` that's appended as literal text. The final output is `$$$`. Every byte of input contributes to output, just perhaps not in the way the user intended.

For tools that process arbitrary user input, this robustness is essential. Users don't read documentation carefully. They experiment, make typos, and expect tools to do something reasonable. By never failing on malformed input, the interpolation code maintains this social contract.

See: Companion Code Section 9

---

## Section 10: Applying the Parsed Reference

When parsing succeeds, the code dispatches on the reference type. For numeric references, it calls `append` directly with the index. For named references, it first calls `name_to_index` to resolve the name, then calls `append` only if resolution succeeded. This asymmetry reflects the reality that named groups might not exist—perhaps the user typo'd the name.

The `if let Some(i) = name_to_index(name)` pattern handles missing names silently. No error is raised; the named reference simply produces no output. Combined with the closure-based design, this gives callers flexibility. A caller could log a warning, fall back to a default, or do nothing—the interpolation code doesn't prescribe behavior.

This section also shows why the closure approach is powerful. The `append` function receives the index and the output buffer, deciding what to write. It might write the captured text directly, apply transformations, or write nothing for missing captures. The interpolation code remains agnostic, focused purely on parsing.

See: Companion Code Section 10

## Section 11: The Test Infrastructure

The test module showcases two patterns worth studying. First, the declarative test macros `find!` and `interp!` transform repetitive test cases into single-line declarations. This makes adding new tests trivial and keeps the test cases readable as a specification of expected behavior.

Second, the `interpolate_string` helper function demonstrates how to use the `interpolate` API. It constructs closures that look up capture group names via binary search and retrieve captured text from a vector. This is simpler than real usage (which involves actual regex matches), but it validates the interpolation logic in isolation.

The test cases themselves document edge cases: empty braces `${}`, unterminated braces `${42`, names containing digits `$42a` vs pure numbers `$42`, and adjacent references `$bar$foo`. Reading these tests tells you what the code handles, often more clearly than prose documentation.

See: Companion Code Section 11

## Key Takeaways

1. **Dependency injection through closures** decouples parsing logic from execution details. The `interpolate` function doesn't know how capture groups are stored or how output is accumulated—it just parses and delegates.
2. **Zero-copy parsing** with lifetimes allows extracting string slices from input without allocation. The `Ref::Named(&'a str)` variant borrows from the input, and the lifetime tracks this dependency at compile time.
3. **The `memchr` crate** provides fast byte searching that ripgrep uses throughout its codebase. Jumping to the next `$` is faster than iterating byte-by-byte, especially for long replacement strings.
4. **Graceful degradation** means never failing on malformed input. Unparseable references become literal text, keeping the tool predictable and forgiving.
5. **Type-level disambiguation** using enums like `Ref` makes the code self-documenting. A numeric reference is structurally different from a named reference, and the type system enforces correct handling of both.
6. **Defensive parsing** constrains input values to reasonable ranges. Parsing to `u32` before converting to `usize` prevents accepting absurdly large capture group indices.

---

## What to Read Next

How does ripgrep actually use this interpolation code? Look at `crates/printer/src/standard.rs` where the search results are formatted and replacement text is constructed using actual regex matches.

What provides the capture groups that get interpolated? The `Matcher` trait from `crates/matcher/src/lib.rs` (covered in Lesson 3) defines how matchers expose capture information.

How do builders configure replacement behavior? The `crates/printer/src/standard.rs` file uses the Builder pattern to let callers specify replacement strings, connecting to the Builder Pattern Ecosystem thread.

# ripgrep crates/searcher/src/sink.rs: The Sink Trait

## What This File Does

This file defines the `Sink` trait, which is the output interface for ripgrep's search engine. While the `Matcher` trait you studied earlier describes *how* to find matches, the `Sink` trait describes *what to do* with those matches once they're found. Every search operation in ripgrep needs somewhere to send its results—whether that's printing to a terminal, counting matches, collecting them into a data structure, or streaming them to another process.

The design follows a "push" model where the searcher drives execution, calling methods on the sink whenever significant events occur: matches found, context lines encountered, binary data detected, or search completion. This inverts the typical iterator pattern and creates a callback-style API that proves surprisingly flexible for building diverse output behaviors.

---

## Section 1: Push vs Pull Iteration Models

Before diving into the code, let's understand the fundamental architectural choice this file represents. When designing a search API, you have two options for how results flow from the searcher to the consumer.

In a "pull" model, the consumer controls execution. You'd call something like `searcher.next_match()` repeatedly, getting results one at a time. This is how Rust iterators work—the caller decides when to advance. The pull model is intuitive and composes well with iterator adapters like `map`, `filter`, and `collect`.

The "push" model inverts this relationship. The searcher controls execution and *pushes* results to a callback as they're discovered. The caller provides a handler—in this case, a `Sink` implementation—and the searcher calls its methods when events occur. This model appears in JavaScript event handlers, Rust's `std::fmt::Write`, and streaming parsers.

Ripgrep chose the push model for several practical reasons. Search involves complex state management: tracking line numbers, managing buffers, handling context lines, detecting binary content. Exposing all this through an iterator interface would require either duplicating state on every yield or maintaining complex suspension points. The push model keeps that complexity inside the searcher, presenting a clean callback interface to consumers. The trait documentation is

refreshingly honest about this—it explicitly acknowledges that the choice stems from "the complexity of the searcher implementation."

See: Companion Code Section 1

---

## Section 2: The SinkError Trait and Error Flexibility

Before the main `Sink` trait, the file defines `SinkError`—a companion trait that describes what error types are acceptable. This pattern might seem like over-engineering at first, but it solves a real problem in library design.

Different consumers of the search API have different error handling needs. A command-line tool might use `std::io::Error` for everything. A library embedded in a larger application might need to wrap errors in a custom type. A testing harness might want `Box<dyn Error>` for maximum flexibility. The `SinkError` trait lets the `Sink` trait be generic over error types without restricting what those types can be.

The trait requires a single method: `error_message`, which constructs an error from anything implementing `Display`. This is the universal escape hatch—any error can be turned into a string representation. Two additional methods, `error_io` and `error_config`, provide specialized constructors for common error sources, but they default to calling `error_message`. This means implementing `SinkError` requires writing only one method while still allowing optimizations for specific error kinds.

Notice that `io::Error` implements `SinkError` with an optimization: its `error_io` implementation just returns the error unchanged rather than converting it to a string and back. This pattern—provide reasonable defaults but allow optimizations—recurs throughout ripgrep's design.

See: Companion Code Section 2

---

## Section 3: The Core Sink Trait

The `Sink` trait itself is the centerpiece of this file. It defines an associated type for errors and seven methods representing different events during a search. Only one method, `matched`, is required—all others have default implementations that do nothing and return success.

The associated type `type Error: SinkError` creates a powerful constraint. Any error type used with `Sink` must implement `SinkError`, ensuring that the searcher can always construct appropriate errors regardless of what specific type the sink uses. This is Rust's trait system

enabling late binding: the concrete error type isn't determined until someone implements `Sink`, but we're guaranteed it will have the capabilities we need.

The `matched` method is the heart of the trait. When the searcher finds a match, it calls this method with a reference to itself (allowing the sink to query searcher configuration) and a `SinkMatch` struct containing match details. The return type `Result<bool, Self::Error>` encodes three possible outcomes: continue searching (`Ok(true)`), stop searching gracefully (`Ok(false)`), or abort with an error (`Err(...)`). This tri-state return appears throughout the trait's methods and provides fine-grained control over search termination.

See: Companion Code Section 3

---

## Section 4: Lifecycle and Event Methods

Beyond `matched`, the trait defines six additional methods forming a complete lifecycle for search operations. Understanding when each is called reveals the searcher's execution model.

The `begin` method fires once when a search starts, before examining any input. This is your hook for initialization—opening output files, writing headers, resetting counters. The `finish` method bookends the search, called after all input is processed (unless an error occurred). It receives a `SinkFinish` struct containing summary statistics like total bytes searched.

Between begin and finish, three methods handle the main search flow. We've discussed `matched` for actual matches. The `context` method handles context lines—the lines before and after matches that tools like grep traditionally show with `-B` and `-A` flags. The `context_break` method signals gaps between groups of context lines, typically rendered as `--` separator lines in grep output.

The `binary_data` method addresses a practical concern: what happens when searching encounters binary content. By default, it simply continues, but implementations might want to skip the file, report a warning, or switch to a binary-aware display mode.

All these methods follow the same `Result<bool, Self::Error>` pattern, meaning any event can stop the search. You could implement a sink that stops after finding ten matches by counting in `matched` and returning `Ok(false)` on the eleventh call.

See: Companion Code Section 4

---

## Section 5: Blanket Implementations for Indirection

The file includes two implementations of `Sink` that don't define new behavior—they forward to existing implementations through indirection. These are `impl Sink for &mut S` and `impl Sink for Box<S>`.

The mutable reference implementation lets you pass a borrowed sink to a search function. Without this, you'd need to move the sink into the searcher and recover it afterward. With this implementation, you can write `searcher.search(&mut my_sink)` and retain ownership of `my_sink` for further use—checking how many matches were found, for instance.

The `Box` implementation enables dynamic dispatch. Sometimes you don't know at compile time which sink implementation you'll use. Perhaps a command-line flag determines whether output goes to a terminal with colors or a file without them. Boxing the sink allows runtime selection: `let sink: Box<dyn Sink<Error = io::Error>> = if use_colors { Box::new(ColorSink::new()) } else { Box::new(PlainSink::new()) }`.

Note the `?Sized` bound on the `Box` implementation: `impl<S: Sink + ?Sized> Sink for Box<S>`. This allows `Box<dyn Sink>` to work, not just `Box<ConcreteSink>`. The `?Sized` relaxes Rust's default assumption that generic types have known sizes at compile time.

These implementations consist entirely of delegation—each method just calls `(**self).method(...)`. The double dereference unwraps the reference or box to reach the underlying sink. The `#[inline]` attributes hint that the compiler should eliminate this indirection when possible.

See: Companion Code Section 5

---

## Section 6: Match Data with SinkMatch

When the searcher finds a match, it doesn't just report the matching bytes—it packages extensive metadata into a `SinkMatch` struct. This struct is the primary currency of information exchange between searcher and sink.

The struct contains the raw match bytes, an absolute byte offset indicating where in the input the match starts, an optional line number (available only if the searcher was configured to track them), the line terminator being used, and access to the underlying buffer. The fields are marked `pub(crate)`, meaning the searcher can construct these structs, but external code must use the accessor methods.

The `lines` method returns an iterator over individual lines within the match. This matters for multi-line matches, which can span several lines. The iterator uses `LineIter`, an internal utility

that respects the configured line terminator. On Unix, that's usually `\n`; on Windows, it might be `\r\n`; and in unusual cases, it could be something else entirely.

The `buffer` method and `bytes_range_in_buffer` method provide a window into the searcher's internal state. This is unusual—most APIs would hide such details—but it enables advanced use cases. A syntax highlighter, for example, might need to look at surrounding context that isn't part of the formal match. By exposing the buffer and the match's location within it, ripgrep enables such extensions without complicating the common case.

See: Companion Code Section 6

---

## Section 7: Context Lines and Search Summary

The `SinkContext` struct parallels `SinkMatch` but describes non-matching lines shown for context. It includes a `SinkContextKind` enum distinguishing three cases: `Before` (lines preceding a match), `After` (lines following a match), and `Other` (used in "passthru" mode, which shows all lines regardless of match status).

Context handling is surprisingly complex in a real grep implementation. Consider overlapping contexts: if matches are close together, their before and after contexts might overlap. The searcher handles this complexity internally, and the sink just sees a stream of context lines with appropriate kind markers. Context breaks (signaled by `context_break`) indicate gaps where lines were skipped.

The `SinkFinish` struct provides summary statistics after a search completes. Currently, it contains the total byte count searched and, if binary detection is enabled, the offset where binary data was first encountered. This struct uses `pub(crate)` fields like `SinkMatch`, keeping construction internal while exposing read-only accessors.

The binary detection offset is particularly useful for tools that want to report "binary file matches" messages. By recording where binary data was first seen, the sink can make informed decisions about how to present results—perhaps switching from line-by-line output to a simple "Binary file matches" message.

See: Companion Code Section 7

## Section 8: The Convenience Sinks Module

The file concludes with a `sinks` module containing pre-built `Sink` implementations for common use cases. These are adapter types that wrap closures, trading flexibility for convenience.

The `UTF8` sink is the most commonly useful. It wraps a closure taking a line number and a string slice, handling the conversion from raw bytes internally. If the match contains invalid UTF-8, it returns an error. If line numbers weren't enabled in the searcher, it returns an error. These restrictions simplify the closure: you're guaranteed valid UTF-8 and a line number.

The `Lossy` sink relaxes the UTF-8 requirement. Invalid bytes are replaced with the Unicode replacement character (`�`) rather than causing an error. This uses `String::from_utf8_lossy`, which the code notes could theoretically be optimized to avoid allocation but currently isn't due to standard library limitations.

The `Bytes` sink skips UTF-8 conversion entirely, passing raw bytes to the closure. This is appropriate when you need to handle arbitrary binary data or when you'll do your own encoding handling.

All three convenience sinks wrap tuple structs around the closure: `pub struct UTF8<F>(pub F)`. The public inner field allows construction without a `new` method: `sinks::UTF8(|line, text| { ... })`. This is a common Rust pattern for simple wrapper types.

See: Companion Code Section 8

## Section 9: Connection to the Builder Pattern Ecosystem

The `Sink` trait integrates with ripgrep's broader builder pattern infrastructure. Looking at how `Searcher` is configured (in `searcher/mod.rs`) and how printers work (in `printer/standard.rs`), you'll see a recurring pattern: builders construct configured objects, and those objects interact through traits like `Sink`.

When you build a `Searcher` with specific options—line numbers enabled, context lines set, binary detection configured—those choices affect what the `Sink` receives. A searcher configured without line counting will never populate the `line_number` field in `SinkMatch`. A searcher without context configured will never call the `context` method. The `Sink` implementations in the `sinks` module enforce these expectations, returning errors if required features aren't enabled.

The printer implementations from the `grep-printer` crate are sophisticated `Sink` implementations. They handle color output, JSON formatting, summary statistics, and more. By implementing `Sink`, they can be passed to any searcher without the searcher knowing anything

about how results will be displayed. This separation of concerns—finding matches vs. displaying them—is the fundamental abstraction this trait enables.

See: Companion Code Section 9

---

## Key Takeaways

First, the push model inverts typical iteration, having the searcher call into your code rather than your code pulling from the searcher—a design driven by the complexity of search state management.

Second, the `SinkError` trait demonstrates how to make a generic trait flexible over error types while ensuring necessary capabilities through trait bounds.

Third, the tri-state return pattern of `Result<bool, Error>` elegantly encodes continue, stop gracefully, and abort with error in a single type.

Fourth, blanket implementations for `&mut S` and `Box<S>` enable borrowed sinks and dynamic dispatch without requiring implementors to think about these patterns.

Fifth, separating match discovery from match handling through the `Sink` trait creates a clean abstraction boundary that allows searchers and output formatters to evolve independently.

Sixth, the convenience sinks in the `sinks` module show how wrapper types around closures can provide ergonomic APIs for common cases while the full trait remains available for complex ones.

---

## What to Read Next

How does the searcher actually call these sink methods? Read `crates/searcher/src/searcher/mod.rs` to see the search loop that drives sink invocation.

What do production-quality sink implementations look like? The `crates/printer/src/standard.rs` file implements `Sink` for terminal output with colors, line numbers, and context.

How does this trait connect to the configuration builders? Examine `crates/searcher/src/searcher/mod.rs` for `SearcherBuilder`, which creates searchers whose options determine what information sinks receive.

# ripgrep crates/regex/src/lib.rs: Regex Crate Entry

## What This File Does

This file serves as the entry point for the `grep-regex` crate, which bridges Rust's powerful regex engine with ripgrep's abstract matching system. In just a few lines, it establishes the public API that the rest of ripgrep uses to perform regular expression searches while hiding the considerable complexity required to make regex matching both correct and fast.

The file exemplifies a common Rust pattern for crate organization: a minimal `lib.rs` that re-exports public types from internal modules. This keeps the public surface area small and intentional while organizing implementation details into focused submodules. What you see here is the tip of an iceberg—beneath these few lines lies sophisticated machinery for optimizing regex patterns, extracting literals for acceleration, and adapting regex features to work within ripgrep's streaming search model.

## Section 1: The Documentation Attribute

The file opens with a documentation comment using Rust's `/*!` syntax, which attaches documentation to the enclosing item rather than the following one. In this context, at the top of `lib.rs`, it becomes the crate-level documentation that appears when you run `cargo doc` or browse the crate on docs.rs.

The description is notably precise: this crate implements the `Matcher` trait from `grep-matcher` specifically for "Rust's regex engine." This phrasing matters. It signals that the crate is an adapter—it doesn't define its own matching abstraction or implement its own regex engine. Instead, it connects two existing pieces: the abstract `Matcher` trait you studied in lesson 3 and the `regex` crate that provides the actual pattern matching.

This architectural clarity helps developers understand where this crate fits in the dependency graph. If you need a different regex engine—perhaps one with different Unicode handling or backreference support—you could theoretically create a parallel crate that implements `Matcher` for that engine instead.

See: Companion Code Section 1

## Section 2: The Missing Docs Lint

The `#![deny(missing_docs)]` attribute converts a warning into a compilation error. Any public item—functions, structs, enums, traits, or their public members—must have documentation comments, or the crate won't compile.

This is a deliberate quality gate that ripgrep applies consistently across its crates. For a library that other code depends on, documentation isn't optional polish—it's essential infrastructure. The lint enforcement means that as the crate evolves and new public APIs are added, documentation must be added simultaneously. You can't forget to document something and discover the oversight months later.

The `#!` syntax makes this a crate-level attribute rather than an item-level one. It applies to the entire crate, not just the module where it appears. This is a subtle but important distinction in Rust's attribute system. Inner attributes (with `!`) configure the thing they're inside; outer attributes (without `!`) configure what follows them.

See: Companion Code Section 2

---

## Section 3: The Public API Through Re-exports

The `pub use` statement defines the crate's entire public API in a single declaration. From the `error` module, it exports `Error` and `ErrorKind`. From the `matcher` module, it exports `RegexCaptures`, `RegexMatcher`, and `RegexMatcherBuilder`.

This is exactly five types, and that minimalism is intentional. Users of this crate don't need to know about AST manipulation, literal extraction, or the other internal concerns. They need to create a matcher (using the builder), use it to search (through `RegexMatcher`), capture groups when needed (`RegexCaptures`), and handle errors (`Error` and `ErrorKind`).

The `crate::` prefix in the paths explicitly references the current crate root. While you could write `pub use error::Error` and it would work the same way here, the explicit form makes the code more self-documenting and resistant to confusion in more complex module hierarchies.

See: Companion Code Section 3

---

## Section 4: The Builder Pattern in Context

The export of `RegexMatcherBuilder` alongside `RegexMatcher` signals that this crate uses the builder pattern for construction. This connects directly to the builder pattern ecosystem you'll encounter throughout ripgrep—in the searcher, the printer, and the directory walker.

Builders make sense when objects have many optional configuration parameters. A `RegexMatcher` needs to know about case sensitivity, multi-line mode, Unicode handling, line terminators, and more. Rather than a constructor with a dozen parameters (most of which you'd set to defaults), the builder pattern lets you chain only the options you care about.

This pattern appears in the re-exports because it's part of the public API. Users are expected to write code like `RegexMatcherBuilder::new(&pattern).case_insensitive(true).build()`. The builder itself is a public type that users interact with directly, not just an implementation detail hidden behind a constructor.

See: Companion Code Section 4

---

## Section 5: The Internal Module Structure

After the public re-exports, the file declares eight private modules: `ast`, `ban`, `config`, `error`, `literal`, `matcher`, `non_matching`, and `strip`. These modules are private by default in Rust—the `mod` keyword without `pub` means they're internal to the crate.

The module names hint at the complexity hidden beneath this simple entry point. The `ast` module likely works with abstract syntax trees of regex patterns. The `ban` module suggests that certain patterns or features are prohibited in some contexts. The `literal` module almost certainly handles literal string extraction—a key optimization technique where the regex engine identifies fixed strings in patterns that can be searched with faster algorithms.

The `config` module is particularly interesting given our learning thread about the builder pattern ecosystem. Configuration is a cross-cutting concern in ripgrep, and seeing a dedicated module for it suggests that regex configuration is substantial enough to warrant its own file.

See: Companion Code Section 5

## Section 6: Why These Specific Modules?

Understanding why these eight modules exist illuminates the challenges of integrating regex matching into a grep tool. Let's consider what each likely handles.

The `error` module provides the crate's error types. Regex compilation can fail in many ways—syntax errors, invalid character classes, patterns that would take too long to compile. Having dedicated error types with an `ErrorKind` enum gives users structured information about what went wrong.

The `matcher` module contains the actual `Matcher` trait implementation. This is where the `grep-matcher` abstraction meets the `regex` crate's concrete types. The adapter logic lives here.

The `literal` module supports one of regex's most important optimizations. If a pattern like `error: .*` starts with the literal string `error:`, the search can use highly optimized string searching algorithms (like the one in the `memchr` crate) to find candidate positions, then verify the full regex only at those positions. This can make searches orders of magnitude faster.

See: Companion Code Section 6

---

## Section 7: The Supporting Modules

The remaining modules handle edge cases and transformations that might not be obvious from the outside.

The `ast` module works with the parsed structure of regex patterns. When you write a pattern like `foo|bar`, the regex crate first parses it into a tree structure representing "alternation between 'foo' and 'bar'". Working at the AST level lets this crate analyze and transform patterns before compilation.

The `ban` module likely handles pattern restrictions. Some regex features that are theoretically valid might be problematic for grep-style searching. Backreferences, for instance, can make regex matching exponentially slow. The ban module might detect and reject such patterns.

The `strip` and `non_matching` modules sound like they transform patterns. Stripping might remove parts of a pattern that don't affect matching semantics, while non-matching could relate to portions of a pattern that don't contribute to the match result.

See: Companion Code Section 7

---

## Section 8: The Separation of Concerns

This file demonstrates excellent separation of concerns at the crate level. The `grep-regex` crate has one job: adapt the `regex` crate to work with `grep-matcher`. Everything else—how patterns are used in searching, how matches are displayed, how files are traversed—lives in other crates.

This separation enables the facade pattern you studied in lesson 1. The `grep` crate can re-export types from `grep-regex` alongside types from `grep-searcher` and `grep-printer`, creating a unified interface. But if a user only needs regex matching without the full grep infrastructure, they can depend on `grep-regex` directly.

The pattern also enables testing in isolation. The regex adaptation logic can be tested without involving file I/O or output formatting. Bugs can be traced to specific crates, and changes to one crate have limited blast radius.

See: Companion Code Section 8

---

## Section 9: Connecting to the Matcher Trait

In lesson 3, you studied the `Matcher` trait that defines the abstract interface for pattern matching. This crate provides the primary implementation of that trait. When ripgrep creates a search, it doesn't directly use the `regex` crate—it uses a `RegexMatcher` from this crate, which implements the `Matcher` trait.

This indirection serves several purposes. It allows ripgrep to potentially support alternative matching backends. It enables caching and optimization strategies specific to grep-style searching. And it provides a consistent interface that the searcher and printer can rely on regardless of the underlying regex implementation.

The `RegexCaptures` type in the exports corresponds to the `Captures` associated type you saw in the `Matcher` trait. When a pattern contains groups like `(error|warning): (.*)`, `RegexCaptures` holds the matched substrings so they can be referenced in replacements or highlighted in output.

See: Companion Code Section 9

---

## Key Takeaways

1. First, crate entry points in Rust often serve as API curation—this file's re-exports define exactly what users can access while hiding eight modules of implementation complexity.

2. Second, the `#![deny(missing_docs)]` lint transforms documentation from optional polish into a compile-time requirement, ensuring that public APIs always have explanations.
3. Third, the builder pattern export (`RegexMatcherBuilder`) follows a consistent pattern across ripgrep's crates, making the ecosystem predictable for developers who learn one crate's conventions.
4. Fourth, module names like `literal`, `ban`, and `ast` reveal that regex integration involves significant preprocessing—patterns aren't just passed directly to the regex engine but are analyzed, optimized, and sometimes restricted.
5. Fifth, this crate's focused responsibility—adapting the regex engine to the Matcher trait—exemplifies the single-responsibility principle at the crate level, enabling the facade pattern seen in the main `grep` crate.

---

## What to Read Next

How is the `RegexMatcher` actually constructed with its many options? Read `crates/regex/src/config.rs` to see the builder pattern in action within this crate.

What optimizations does literal extraction enable? Explore `crates/regex/src/literal.rs` to understand how fixed strings are extracted from patterns for faster searching.

How does `RegexMatcher` implement the `Matcher` trait you studied earlier? Read `crates/regex/src/matcher.rs` for the core adapter logic that bridges two APIs.

What patterns does ripgrep actually ban, and why? The `crates/regex/src/ban.rs` file will reveal what's considered too dangerous or slow for grep-style matching.

# ripgrep crates/regex/src/config.rs: Regex Configuration

## What This File Does

This file defines the configuration system for ripgrep's regex matching engine. It establishes a `Config` struct that captures all the knobs and switches that affect how patterns are parsed, compiled, and matched—everything from case sensitivity to memory limits to line terminator handling. But the file does more than just hold configuration values; it introduces the `ConfiguredHIR` type, which pairs a compiled regex representation (the HIR, or High-level Intermediate Representation) with the configuration that produced it.

The design here solves a fundamental problem in regex compilation: configuration affects not just how you parse a pattern, but how you can transform and optimize it later. By bundling the HIR with its configuration, ripgrep can perform subsequent transformations—like adding word boundaries or line anchors—while preserving the original settings. This is a key piece of the Builder Pattern ecosystem you've seen developing across ripgrep's crates, where configuration objects get progressively enriched until they're ready to produce their final output.

---

## Section 1: The Configuration Structure

The `Config` struct is the central type in this file, and it's a comprehensive collection of boolean flags, size limits, and optional values that control regex behavior. What makes this struct interesting is how it balances two different domains: the standard regex configuration options (like `case_insensitive`, `multi_line`, and `dot_matches_new_line`) and ripgrep-specific concerns (like `line_terminator`, `ban`, and `whole_line`).

Notice that nearly all fields use `pub(crate)` visibility. This is deliberate—the configuration is meant to be set through the builder API in `lib.rs`, not manipulated directly by external code. The struct itself is `Clone` and `Debug`, which are essential traits for any configuration type: `Clone` because configurations often need to be duplicated and modified, and `Debug` for diagnostics and logging.

The field names themselves tell a story about ripgrep's features. You see `case_smart`, which enables the intelligent case-sensitivity detection that makes `rg foo` case-insensitive but `rg Foo` case-sensitive. You see `fixed_strings` for literal matching, `word` for word-boundary wrapping,

and `whole_line` for full-line matching. Each of these represents a user-facing feature that must be threaded through the entire regex compilation process.

See: Companion Code Section 1

---

## Section 2: Default Values and Their Rationale

The `Default` implementation for `Config` reveals important design decisions. Most boolean flags default to `false`, which makes sense—you want the simplest, most predictable behavior unless the user explicitly requests otherwise. But there are notable exceptions that reflect ripgrep's priorities.

The `unicode` field defaults to `true`, meaning ripgrep treats patterns as Unicode by default. This is user-friendly but has performance implications—Unicode character classes are larger and more complex than their ASCII counterparts. The choice reflects a philosophy of correctness over raw speed in the default case.

The size limits deserve special attention. The comments note that these limits are "much bigger than what's in the regex crate by default." The `size_limit` of 100 megabytes and `dfa_size_limit` of 1 gigabyte are deliberately generous because ripgrep users often search with complex patterns across large codebases. The regex crate's conservative defaults are appropriate for a general-purpose library, but ripgrep can afford to be more permissive since it's an end-user tool where the user has implicitly consented to some resource usage.

The `nest_limit` of 250 controls how deeply nested parentheses can be in a pattern. This prevents stack overflow during parsing while still allowing reasonably complex patterns.

See: Companion Code Section 2

---

## Section 3: Smart Case Detection

The `is_case_insensitive` method implements ripgrep's signature "smart case" feature. The logic is elegant: if the user explicitly requested case-insensitive matching, honor that. If smart case is disabled, don't do anything clever. But if smart case is enabled, examine the pattern to decide.

The decision hinges on two questions about the pattern: does it contain any literals at all, and does it contain any uppercase characters? If the pattern has literals but no uppercase letters, it's treated as case-insensitive. This matches the intuition that `rg foo` should find "foo", "Foo", and "FOO", while `rg Foo` specifically means "I care about this capital F."

The method takes an `AstAnalysis` parameter—this is a type from another module that pre-computes information about the pattern's AST. By analyzing the AST once and passing the results around, ripgrep avoids repeatedly walking the pattern structure. This is a common optimization pattern: do expensive analysis once, then consult the cached results.

See: Companion Code Section 3

---

## Section 4: Fixed String Detection

The `is_fixed_strings` method addresses a subtle optimization opportunity. When patterns are just literal strings with no regex metacharacters, ripgrep can skip the full parsing and translation pipeline and directly construct an HIR from the string bytes. This is faster and uses less memory.

But the method must be conservative. If case-insensitivity is involved (either explicit or smart), the patterns need to go through the normal translation process so that case folding tables get applied correctly. You can't just search for the literal bytes "foo" when you need to match "FOO" too.

The method also handles the `fixed_strings` configuration explicitly set by the user (like grep's `-F` flag). Even then, it checks for line terminators in the patterns—if a literal contains a newline, that's going to cause problems downstream, so the method returns `false` to force normal processing where proper error handling can occur.

The final fallback checks whether patterns without the `fixed_strings` flag happen to contain no metacharacters. If someone searches for `hello` without using `-F`, ripgrep can still take the fast path because the pattern is indistinguishable from a literal.

See: Companion Code Section 4

---

## Section 5: The ConfiguredHIR Type

`ConfiguredHIR` is where this file gets architecturally interesting. It bundles an HIR (the compiled pattern representation) with the configuration that produced it. This pairing enables something crucial: subsequent transformations can be applied while respecting the original configuration.

Why is this necessary? Consider the `word` feature. To make a pattern match only at word boundaries, ripgrep wraps it with word boundary assertions. But which assertions? If Unicode mode is enabled, you need Unicode-aware word boundaries. If CRLF mode is enabled, line anchors

need to handle carriage returns. The configuration must travel with the HIR so these decisions can be made correctly.

The type provides accessor methods for both its components—`config()` and `hir()`—and transformation methods like `into_whole_line()` and `into_word()` that consume the value and produce a new `ConfiguredHIR` with the modified pattern. This is an ownership-based transformation pattern: you move the original, modify it, and get a new instance back.

See: Companion Code Section 5

---

## Section 6: Building HIR from Patterns

The `ConfiguredHIR::new` constructor is the most complex method in this file, and it demonstrates the full pattern compilation pipeline. It takes a configuration and a slice of patterns, then produces a single HIR that represents an alternation of all patterns—searching for any of them.

The method begins by checking `is_fixed_strings`. If it returns true, the fast path directly constructs HIR literals from the pattern bytes and joins them with alternation. The logging statement here is valuable for debugging: when ripgrep processes thousands of patterns from a file, knowing it took the fast path helps diagnose performance issues.

The slow path is more involved. Each pattern gets wrapped in a non-capturing group `(?:...)` to prevent operator precedence issues when they're joined with `|`. If `fixed_strings` is enabled but we couldn't take the fast path (due to case sensitivity requirements), the patterns get escaped with `regex_syntax::escape` to neutralize any metacharacters.

The parsing happens in two stages: first AST parsing with the configured limits, then translation to HIR with all the regex flags applied. Between these stages, `AstAnalysis` examines the pattern to inform the smart case decision.

See: Companion Code Section 6

---

## Section 7: Post-Processing the HIR

After the basic HIR is constructed, several post-processing steps may occur. The `ban` check examines the HIR for forbidden bytes—if the configuration specifies that a certain byte must never appear in matches, this validation ensures the pattern doesn't violate that constraint.

Line terminator stripping is another crucial step. When ripgrep is doing line-oriented searching, matches shouldn't include the line terminators themselves. The `strip_from_match` function transforms the HIR to ensure that `\n` (or `\r\n`) can't be part of a match. The comments explain why this doesn't apply to the fixed-strings fast path: `is_fixed_strings` already returns false if any pattern contains a line terminator, so stripping is unnecessary.

The comment about avoiding this step for fixed strings mentions that building a new HIR "can actually take a little time" when ripgrep is given a huge set of literals. This is a reminder that even seemingly simple operations have costs at scale, and the ripgrep developers have profiled real-world usage to find these bottlenecks.

See: Companion Code Section 7

---

## Section 8: Converting HIR to a Regex

The `to_regex` method transforms the configured HIR into an actual `Regex` object from the `regex-automata` crate—this is what will actually perform matching. The method configures the regex engine with limits derived from the configuration.

The comments here are particularly illuminating about ripgrep's performance philosophy. The one-pass DFA size limit gets extra room because "the one-pass DFA is usually not a perf bottleneck for ripgrep." The full DFA limits are set higher than defaults because "with ripgrep we can afford to spend a bit more time on building them."

The `hybrid_cache_capacity` setting connects to the lazy DFA (also called hybrid DFA), which is ripgrep's secret weapon for fast searching. Unlike a full DFA that's built upfront, the lazy DFA constructs states on demand during matching. The cache capacity determines how many states can be remembered, trading memory for speed.

Notice the error handling: `build_from_hir` returns a Result, and errors get wrapped with `Error::regex`. This maintains the error abstraction throughout the crate.

See: Companion Code Section 8

## Section 9: Line Terminator Complexity

The `line_terminator` method has an extensive comment explaining a subtle interaction between anchors and line-oriented searching. When a pattern uses text anchors (`\A` and `\z` for start/end

of the entire input, not just lines), the fast line-by-line search path doesn't work correctly.

The issue is that the fast path operates on individual lines, but text anchors are about the entire input. If you're searching for `^hello` (with multi-line mode disabled), that only matches at the very beginning of the file, not at the start of each line. The line-by-line search path would incorrectly match at the start of every line.

The solution is conservative: if text anchors are present, don't return a line terminator. Without a line terminator, the higher-level search infrastructure knows it can't use the fast path and falls back to slower but correct multi-line searching.

The second half of the comment describes a later optimization where if the line terminator is guaranteed to never appear in a match (determined by `non_matching_bytes`), the fast path can still be used even without an explicit line terminator configuration. This is the kind of layered optimization that makes ripgrep fast: multiple mechanisms working together, with fallbacks when conditions aren't met.

See: Companion Code Section 9

---

## Section 10: HIR Transformations

The `into_whole_line` and `into_word` methods demonstrate how `ConfiguredHIR` enables safe transformations. Both methods wrap the existing HIR with additional assertions, but they must choose the right assertions based on configuration.

For whole-line matching, the choice depends on CRLF handling. On Unix, lines end with `\n`, so `StartLF` and `EndLF` anchors work. On Windows (or with CRLF mode enabled), lines might end with `\r\n`, requiring `StartCRLF` and `EndCRLF` anchors. The helper methods `line_anchor_start` and `line_anchor_end` encapsulate this decision.

For word boundaries, the choice depends on Unicode mode. Unicode word boundaries recognize characters like `é` as word characters, while ASCII word boundaries only recognize `[a-zA-Z0-9_]`. The `WordStartHalfUnicode` and `WordEndHalfUnicode` variants handle Unicode-aware boundaries.

Both methods consume `self` and return a new `ConfiguredHIR`. This ownership pattern ensures the original can't be accidentally used after transformation—you must use the transformed version.

See: Companion Code Section 10

---

## Section 11: Helper Functions

The `has_line_terminator` function is a small utility that checks whether a literal string contains any line terminator bytes. It handles both the CRLF case (checking for `\r` or `\n`) and the single-byte case (checking for whatever byte the line terminator is configured as).

This function is used during the `is_fixed_strings` check to ensure that literal patterns don't contain embedded line terminators. If they do, the fast path can't be used because the line terminator handling needs special treatment.

The function operates on bytes, not characters, using `iter().copied()` to iterate over the byte slice. This is appropriate because line terminators are ASCII characters with single-byte representations, and checking bytes is faster than decoding UTF-8.

See: Companion Code Section 11

---

## Key Takeaways

First, configuration objects that bundle values with the data they configure enable safe transformations—`ConfiguredHIR` can be modified while respecting its original settings because it carries those settings along.

Second, default values in configuration types should reflect the tool's priorities and use cases. Ripgrep's generous size limits and Unicode-by-default choices serve end users, not library consumers.

Third, fast paths with careful bailout conditions are a powerful optimization pattern. The `is_fixed_strings` method enables a fast path when possible but has multiple conditions that force fallback to the slower but more correct general case.

Fourth, smart features like case sensitivity detection require analyzing the input before deciding how to process it—the `AstAnalysis` type captures this pattern of pre-computation.

Fifth, complex subsystems often have subtle interactions documented in comments. The line terminator handling shows how multiple optimizations can conflict and require careful coordination.

Sixth, ownership-based transformation methods like `into_whole_line` prevent bugs by making it impossible to use the pre-transformation value accidentally.

## What to Read Next

How does the builder API expose these configuration options to users? Read `crates/regex/src/lib.rs` to see how `Config` gets populated.

What does `AstAnalysis` compute, and how is it used? Read `crates/regex/src/ast.rs` to understand pattern analysis.

How does `strip_from_match` actually modify the HIR? Read `crates/regex/src/strip.rs` to see HIR transformation in action.

What bytes does `non_matching_bytes` identify, and why does that matter for optimization? Read `crates/regex/src/non_matching.rs` for that analysis.

# ripgrep crates/regex/src/matcher.rs: Implementing Matcher

## What This File Does

This file bridges the abstract world of trait definitions with the concrete reality of regular expression execution. It implements ripgrep's `Matcher` trait using Rust's `regex-automata` crate, creating a `RegexMatcher` that the searcher can use without knowing anything about the underlying regex engine. This is where pattern matching actually happens—every search ripgrep performs ultimately flows through the code in this file.

The file follows the builder pattern we've seen throughout ripgrep, providing `RegexMatcherBuilder` with dozens of configuration options that translate into regex behavior. But it does more than just wrap the regex crate. It applies ripgrep-specific optimizations like line-aware matching and literal extraction, transforming user patterns into forms that can be searched more efficiently. This is the concrete implementation that makes the abstract `Matcher` trait useful.

## Section 1: The Builder Pattern Completed

The `RegexMatcherBuilder` represents the final stage of a configuration journey that began with command-line arguments and flowed through `Config`. This builder doesn't just store settings—it provides a fluent API for constructing matchers while maintaining the invariants that ripgrep requires.

Notice how the builder is `Clone` and `Debug`. These derive traits matter for a builder because they allow the same configuration to be reused for multiple matchers, and they enable logging and debugging of the build process. The builder holds a single field: a `Config` struct that we examined in the previous lesson. This composition means the builder delegates storage to `Config` while providing a more ergonomic API for external users.

The `Default` implementation creates a new builder, which in turn creates a default `Config`. This chaining of defaults ensures that ripgrep has sensible behavior out of the box—case sensitive, Unicode enabled, reasonable resource limits. Users only need to configure the options they care about changing.

See: Companion Code Section 1

---

## Section 2: From Pattern to Matcher

The `build` method is deceptively simple—it just calls `build_many` with a single pattern. But `build_many` is where the real work happens, and understanding it reveals how ripgrep transforms user patterns into optimized search machinery.

The method first calls `self.config.build_many(patterns)`, which as we saw in the config lesson, produces a "configured HIR"—a high-level intermediate representation that has already been preprocessed based on user settings. This HIR captures the semantic meaning of the pattern while stripping away details the regex engine doesn't need to know about.

The next step handles two special matching modes: whole-line and word matching. These transformations modify the pattern structure itself. If `whole_line` is enabled, the HIR is wrapped to only match complete lines. If `word` is enabled (and whole-line isn't, since whole-line subsumes word), the HIR is transformed to require word boundaries. The comment notes that "whole_line is a strict subset of word"—meaning any whole-line match is necessarily also a word match, so we don't need both transformations.

Finally, `chir.to_regex()` compiles the transformed HIR into an actual regex automaton. The logging statement reveals the final pattern that will actually be executed—invaluable for debugging complex pattern transformations.

See: Companion Code Section 2

---

## Section 3: The Fast-Line Optimization

The most interesting optimization in `build_many` is the construction of `fast_line_regex`. This represents a key insight: sometimes you can find matches faster by first finding candidate lines, then verifying them with the full regex.

The code extracts literal substrings from the pattern using `InnerLiterals::new`. For a pattern like `\w+foo\w+`, the inner literal `foo` can be searched much faster than the full regex. When found, ripgrep can locate the containing line and run the complete pattern only on that line. This optimization is particularly powerful for line-oriented search because it dramatically reduces how much text the full regex engine needs to process.

The `one_regex` method attempts to build a simplified regex from these literals. If successful, `fast_line_regex` becomes `Some(regex)`. If the pattern doesn't contain extractable literals, or if the extraction wouldn't provide a speedup, it remains `None`. The builder handles both cases gracefully—the optimization is opportunistic, not required.

The final step creates the `RegexMatcher` with all four components: the configuration, the main regex, the optional fast regex, and the set of bytes that can never appear in a match. This last component enables another optimization: if the searcher knows certain bytes never match, it can skip over them quickly.

See: Companion Code Section 3

---

## Section 4: Builder Methods as Documentation

The builder methods reveal ripgrep's feature set through their documentation. Each method configures a single aspect of regex behavior, and the doc comments explain not just what the option does, but why you might want it.

Consider `case_smart`. The documentation carefully explains the algorithm: case insensitivity is enabled only when the pattern contains literal characters AND none of those literals are uppercase. This is a user-experience decision—people searching for `Foo` probably want case sensitivity, while those searching for `foo` probably don't care. The builder encodes this heuristic as a simple boolean flag.

The `line_terminator` method documentation deserves careful reading. It explains the guarantee that enabling a line terminator provides: the matcher will never produce a match containing that byte. This guarantee is what enables line-oriented search optimizations. But the documentation also explains the cost: patterns that explicitly include the line terminator will cause an error. The pattern `a\nb` cannot be transformed to exclude the newline without changing its fundamental meaning.

Each builder method returns `&mut RegexMatcherBuilder`, enabling method chaining. This is idiomatic Rust builder pattern usage: `builder.case_smart(true).word(true).build(pattern)`. The mutable reference return type means each method can modify the builder in place while still allowing fluent chaining.

See: Companion Code Section 4

---

## Section 5: The RegexMatcher Structure

The `RegexMatcher` struct itself tells a story about optimization trade-offs. It contains four fields, each serving a distinct purpose in the search process.

The `config` field preserves the original configuration, primarily so the matcher can report its line terminator setting. This is important because the searcher needs to know what the matcher considers a line boundary—different matchers might use different terminators, and the searcher adapts accordingly.

The `regex` field is the workhorse—the compiled regular expression that actually performs matching. This is a `regex_automata::meta::Regex`, the most flexible regex type from the regex-automata crate. It automatically selects the best matching strategy (DFA, NFA, or hybrid) based on the pattern.

The `fast_line_regex` field holds our optional optimization regex. Its `Option` type makes the optimization's optionality explicit in the type system. Code that uses this field must handle both cases, preventing bugs where we assume the optimization is always available.

Finally, `non_matching_bytes` is a `ByteSet`—a compact representation of which byte values can never appear in a match. If a pattern only matches ASCII letters, this set would contain all non-ASCII bytes and all ASCII non-letters. The searcher can use this information to quickly skip over bytes that couldn't possibly be part of a match.

See: Companion Code Section 5

---

## Section 6: Implementing the Matcher Trait

Now we reach the heart of this file: the `impl Matcher for RegexMatcher` block. This is where abstract interface meets concrete implementation. Every method defined in the `Matcher` trait must be implemented here, translating the trait's requirements into regex-automata API calls.

The type declarations at the start of the impl block are crucial. `type Captures = RegexCaptures` tells users of this matcher what type to use for capture groups. `type Error = NoError` is even more interesting—it declares that this matcher's operations are infallible. The `NoError` type from grep-matcher is an uninhabited type (like `!`), meaning it can never be constructed. This provides a compile-time guarantee that these methods cannot fail, allowing callers to safely unwrap results.

Each method is marked `#[inline]`, hinting to the compiler that these small methods should be inlined into their callers. For hot paths like `find_at` that might be called millions of times during a search, avoiding function call overhead matters. The compiler may or may not follow this hint, but for performance-critical methods, it's worth providing.

See: Companion Code Section 6

---

## Section 7: Core Matching Methods

The `find_at` method is the most fundamental matching operation. It takes a haystack (the bytes to search) and an offset, returning the first match starting at or after that offset. The implementation shows the translation between grep-matcher's types and regex-automata's types.

The `Input::new(haystack).span(at..haystack.len())` construction creates a regex-automata input that starts searching at position `at`. The `span` method restricts the search to a portion of the haystack—we don't want to find matches before our starting position. When a match is found, we translate from regex-automata's match type to grep-matcher's `Match::new(m.start(), m.end())`.

The `try_find_iter` method demonstrates handling iteration with early termination. The closure `matched` receives each match and returns `Result<bool, E>`. Returning `Ok(true)` continues iteration, `Ok(false)` stops it, and `Err(err)` propagates an error. This three-way control flow is wrapped in `Result<Result<(), E>, NoError>`—the outer Result is for matcher errors (which can't happen), and the inner Result is for callback errors (which might happen). This nested Result pattern appears throughout ripgrep's callback-based APIs.

See: Companion Code Section 7

---

## Section 8: Capture Group Management

The capture-related methods show how ripgrep handles one of regex's more complex features. `new_captures` creates a fresh `RegexCaptures` instance, allocating the storage needed to hold capture group offsets. The `create_captures` call on the regex allocates enough space for all groups in the pattern.

The `capture_count` method returns how many capture groups exist. Note that this includes group 0 (the entire match), so a pattern with no explicit groups still has a capture count of 1. This matches user expectations—even `grep` without parentheses can report what it matched.

The `capture_index` method looks up a named capture group, returning its numeric index. The call to `group_info().to_index(PatternID::ZERO, name)` reveals regex-automata's multi-pattern support—`PatternID::ZERO` specifies we want the first (and usually only) pattern. For ripgrep's alternation of patterns case (`build_many`), this still works because all patterns share the same group numbering.

The `captures_at` method fills in capture group positions for a match starting at a given offset. It mutably borrows the captures, fills them via `search_captures`, and returns whether a match was found. This separation between "did it match" and "where did the groups match" allows callers to reuse capture storage across multiple searches.

See: Companion Code Section 8

---

## Section 9: The Candidate Line Optimization in Action

The `find_candidate_line` method is where the `fast_line_regex` optimization pays off. This method implements the `Matcher` trait's optional optimization hook for line-oriented searching.

When `fast_line_regex` is `Some`, we use it to quickly find potential matches. The `search_half` method performs a "half match"—it finds where a match ends but doesn't track where it starts or what groups matched. This is faster than a full match and sufficient for identifying candidate lines. The result is wrapped in `LineMatchKind::Candidate`, signaling to the searcher that this is a potential match that needs verification.

When `fast_line_regex` is `None`, we fall back to `shortest_match`, which uses the full regex. The result is wrapped in `LineMatchKind::Confirmed` because a match from the full regex doesn't need additional verification.

This two-tier system lets the searcher work efficiently: quickly scan for candidates, then verify with the full regex only on the lines that might match. For patterns with good inner literals, this can provide significant speedup. For patterns without extractable literals, we lose nothing—the full regex is used directly.

See: Companion Code Section 9

## Section 10: The RegexCaptures Wrapper

The `RegexCaptures` type wraps regex-automata's capture storage while implementing grep-matcher's `Captures` trait. This wrapper exists because the trait system requires a concrete type—we can't just use regex-automata's `Captures` directly since it's from a different crate.

The `Captures` trait implementation is straightforward. The `len` method returns the total number of capture groups across all patterns via `all_group_len()`. The `get` method retrieves a specific group's span, translating between regex-automata's representation and grep-matcher's `Match` type.

The `pub(crate)` visibility on `new` and `captures_mut` reveals the design intent: these methods are implementation details needed within the regex crate but shouldn't be part of the public API. The wrapper provides a clean public interface (the `Captures` trait methods) while keeping the messy internals hidden.

See: Companion Code Section 10

---

## Section 11: Testing the Edge Cases

The test module demonstrates ripgrep's approach to testing: focus on behavior that differs from naive expectations. Each test targets a specific feature that might be confusing or surprising.

The `word` test contrasts ripgrep's word matching with regex `\b` assertions. The pattern `-2` with word matching enabled matches in "abc -2 foo" because word boundaries in ripgrep only require non-word characters on each side. But `\b-2\b` doesn't match because `\b` requires a word character on one side—and neither `-` nor space are word characters. This subtle distinction justifies having a separate `word` option rather than just telling users to add `\b`.

The `line_terminator` and `line_terminator_error` tests verify the guarantee discussed earlier: setting a line terminator prevents matches that span lines, and patterns that require the line terminator cause build errors. The `candidate_lines` test (currently ignored, awaiting fixes to literal extraction) would verify the fast-line optimization produces candidate matches rather than confirmed ones when appropriate.

See: Companion Code Section 11

## Key Takeaways

First, trait implementations bridge abstract interfaces with concrete implementations—`RegexMatcher` makes `Matcher` useful by providing actual regex-based matching while hiding the complexity of the underlying regex engine.

Second, the builder pattern's final stage often performs significant transformations, not just configuration storage—here, patterns are modified for word matching, whole-line matching, and line terminator handling before compilation.

Third, optional optimizations like `fast_line_regex` should be represented with `Option` types, making the optimization's availability explicit and forcing calling code to handle both cases.

Fourth, wrapper types like `RegexCaptures` exist to implement foreign traits for foreign types—Rust's orphan rules require this indirection when bridging crates.

Fifth, `NoError` as an associated error type provides compile-time proof that operations are infallible, allowing callers to safely unwrap without runtime cost.

Sixth, test cases should target surprising behavior and edge cases that documentation alone might not clarify—the word matching vs `\b` distinction is a perfect example.

## What to Read Next

How does the line terminator transformation actually modify patterns? Read `crates/regex/src/strip.rs` to see how ripgrep removes line terminators from regex syntax trees.

What happens when the searcher receives candidate vs confirmed matches? Read `crates/searcher/src/searcher/mod.rs` to see how line-oriented search uses the `Matcher` trait's optimization hooks.

How does literal extraction work to create the fast-line regex? Read `crates/regex/src/literal.rs` to understand the `InnerLiterals` type referenced in this file.

# ripgrep crates/regex/src/literal.rs: Literal Optimization

## What This File Does

This file implements one of ripgrep's most important performance optimizations: extracting literal strings from regular expressions to enable fast pre-filtering. The core insight is deceptively simple—if a regex must match certain literal bytes to succeed, you can search for those bytes first using highly optimized SIMD routines, then only run the full regex engine on the matching lines. For a regex like `\s+Sherlock\s+`, the word "Sherlock" must appear for any match to occur, so ripgrep can scan gigabytes of text looking only for that literal, dramatically reducing the work the regex engine needs to do.

The implementation centers on an `Extractor` struct that traverses the regex's internal representation (the HIR, or High-level Intermediate Representation) and attempts to extract useful literals. This isn't a simple task—the extractor must handle concatenations, alternations, repetitions, and character classes, all while maintaining correctness about whether extracted literals are "exact" (the complete match) or "inexact" (just part of it). The file also includes sophisticated heuristics for deciding when extracted literals are actually helpful versus when they might slow things down.

## Section 1: The Inner Literal Strategy

The `InnerLiterals` struct encapsulates the entire optimization strategy. Unlike prefix or suffix literal extraction (where you look for what the regex starts or ends with), inner literal extraction finds literals that must occur *somewhere* in any match. This is particularly powerful for line-oriented searching because of a crucial property: if you find an inner literal on a line, you can run the full regex on just that line rather than the entire file.

The validity of this optimization depends entirely on ripgrep's line-oriented search model. When you search line-by-line rather than across the entire file as one continuous string, finding an inner literal tells you exactly which line to test with the full regex. This is why the `new` constructor immediately bails out if no line terminator is configured—without line boundaries, you can't safely narrow down where to apply the full regex after finding a literal.

The constructor also includes important short-circuit logic. If the regex engine reports that it's already "accelerated" (meaning it has its own internal literal optimizations), ripgrep defers to the engine unless the pattern contains Unicode word boundaries. Unicode word boundaries are expensive enough that the regex engine might fall back to slower paths, making ripgrep's custom optimization worthwhile even when the engine claims to be accelerated.

See: Companion Code Section 1

---

## Section 2: The Extractor Architecture

The `Extractor` struct is where the real work happens. It's designed as a recursive tree-walker that descends through the HIR and accumulates literal sequences at each node. The struct carries configuration limits that prevent the extraction process from exploding in time or space—you don't want to spend more time extracting literals than you'd save by using them.

Four limits govern the extraction: `limit_class` caps how many characters from a character class will be expanded into literals, `limit_repeat` controls how many iterations of a repetition will be unrolled, `limit_literal_len` prevents any single literal from growing too long, and `limit_total` bounds the total number of literals in the final sequence. These limits exist because literal extraction has combinatorial complexity—a regex like `[abc][def][ghi]` could expand to 27 different literals, and with longer character classes or deeper nesting, this explodes rapidly.

The main entry point is `extract_untagged`, which calls the recursive `extract` method, then applies post-processing. The "untagged" name refers to stripping away the internal `TSeq` wrapper (a "tagged sequence" that tracks additional metadata) and returning just the raw `Seq` of literals. The post-processing includes optimization passes and a critical "is this actually good?" check that can throw away the entire result if the heuristics suggest the literals won't help.

See: Companion Code Section 2

---

## Section 3: Handling Concatenation

The `extract_concat` method handles sequential patterns like `abc` or `foo[a-z]bar`. The fundamental operation here is the **cross product**: if you have literals `["a", "b"]` from one part and `["x", "y"]` from another, concatenating them yields `["ax", "ay", "bx", "by"]`. This is where the combinatorial explosion comes from, and why limits are essential.

The implementation uses a clever optimization for patterns with "gaps"—places where literal extraction fails in the middle of a concatenation. Consider `foo[a-z]+bar[a-z]+baz`: the `[a-z]+` parts produce no useful literals, but `foo`, `bar`, and `baz` are all perfectly good candidates. Rather than giving up when hitting an inexact sequence, the method tracks a `prev` candidate and uses the `choose` method to pick the best option among multiple possible literal sets.

This "keep the best so far" strategy allows the extractor to scan through an entire concatenation and identify the most useful literals, even when they're separated by unpredictable content. The `is_really_good` check enables short-circuiting: if we find a sequence that's clearly excellent (long literals, small count), we can stop looking and avoid wasting time on the rest of the pattern.

See: Companion Code Section 3

---

## Section 4: Handling Alternation

Alternation (`a|b|c`) requires a different operation: **union** rather than cross product. If one branch matches `"foo"` and another matches `"bar"`, the combined pattern can match either, so the literal set becomes `["foo", "bar"]`. The `extract_alternation` method builds this union incrementally.

The short-circuit condition here is different from concatenation: once the sequence becomes infinite (meaning we've hit something we can't represent as finite literals), every subsequent union will also be infinite, so we can stop early. This prevents wasted work when one branch of an alternation contains something like `.*` that defeats literal extraction entirely.

The union operation in the `Extractor` includes a fascinating optimization for when the total would exceed limits. Rather than immediately giving up and returning an infinite sequence, it tries to **trim** the existing literals—keeping only their first few bytes. The magic number is 4 bytes, chosen specifically because downstream the literals might be fed to the Teddy algorithm, which can search for literals up to 4 bytes efficiently. By trimming rather than abandoning, the extractor preserves at least some filtering capability.

See: Companion Code Section 4

## Section 5: Handling Repetition

Repetitions are perhaps the trickiest case because the semantics depend heavily on the bounds. The `extract_repetition` method handles `?`, `*`, `+`, `{n}`, and `{n,m}` quantifiers, each requiring

different treatment to maintain correctness about exactness.

For optional patterns (`?`), both the present and absent cases are valid, so the result is a union. For zero-or-more (`*`), the empty match is always possible, making it difficult to extract useful required literals. For one-or-more (`+`), at least one occurrence must appear, so the sub-expression's literals become inexact (they mark the start of a match but not the whole thing).

Fixed repetitions like `{3}` are handled by unrolling: we cross-product the sub-expression with itself the specified number of times, up to the `limit_repeat` threshold. This correctly produces `"aaa"` from `a{3}`. Bounded ranges like `{3,5}` work similarly but immediately mark results as inexact since the actual number of repetitions is variable.

The handling of greediness (`*` vs `*?`) affects the order of alternatives: greedy puts the non-empty match first, lazy puts the empty match first. This matters for optimizations that might prefer one interpretation over another.

See: Companion Code Section 5

---

## Section 6: The Tagged Sequence Wrapper

The `TSeq` type wraps the standard `Seq` from regex-syntax with an additional piece of state: the `prefix` boolean. This tracks whether the literals in the sequence are known to start at a predictable position in the match. When `prefix` is false, the literals could appear anywhere, which changes how they compose with other sequences.

This distinction matters for correctness during cross-product operations. If a sequence isn't a prefix, you can't simply concatenate it onto another sequence because you don't know where it starts. Instead, you must `choose` between the sequences rather than combining them. The `make_not_prefix` method marks a sequence when extraction continues past an inexact segment.

The wrapper also adds quality-assessment methods that don't exist on the raw `Seq` type. The `is_good` and `is_really_good` methods implement heuristics about whether the extracted literals are actually useful. These consider minimum literal length and total count—a sequence with many very short literals might match so frequently that filtering provides no speedup.

See: Companion Code Section 6

## Section 7: The Quality Heuristics

The quality heuristics deserve special attention because they're critical to the optimization's success. A prefilter that matches too often can actually slow down search—you spend time checking matches that rarely lead to real results.

The `is_good` method implements conservative thresholds. If any literal is "poisonous" (empty, or a single very common byte), the sequence is rejected. For short literals (length 1), the sequence must have at most 3 literals. For longer literals, the minimum length must be at least 2 and the total count at most 64. The `is_really_good` method uses stricter thresholds (minimum length 3, count at most 8) and triggers early short-circuiting.

The `is_poisonous` function identifies literals that would match far too frequently. Empty strings match everywhere, obviously. Single-byte literals are checked against a frequency rank—common bytes like spaces, tabs, and common ASCII letters score above 250 on a 0-255 scale and are considered poisonous. This ranking comes from typical byte frequency distributions in text files.

The `choose` method compares two candidate sequences when the extractor must pick one. It prefers finite over infinite, non-poisonous over poisonous, longer minimum literals over shorter, and fewer total literals over more. When all else is equal, it just picks the first one.

See: Companion Code Section 7

## Section 8: Character Class Extraction

Character classes like `[a-z]` or `[0-9]` can sometimes be expanded into literal sets. The `extract_class_unicode` and `extract_class_bytes` methods handle this expansion, but only when the class is small enough. A class like `[abc]` becomes `["a", "b", "c"]`, which is useful. A class like `[a-z]` would become 26 literals, which might still be okay. But `[A-Za-z0-9]` would be 62 literals, likely exceeding limits.

The `limit_class` threshold (default 10) controls when classes get expanded versus abandoned. The methods iterate through the class ranges, counting characters, and return an infinite sequence if the count exceeds the limit. This prevents pathological cases where a single character class would blow up the entire extraction.

Unicode classes are particularly tricky because a single Unicode range can contain thousands of codepoints. The `class_over_limit_unicode` method counts efficiently using the range's `len()` method, bailing out early if the limit is exceeded partway through iteration.

See: Companion Code Section 8

## Section 9: Building the Fast Regex

The `one_regex` method on `InnerLiterals` is the final step: converting the extracted literal sequence back into an actual regex that can be used for searching. This creates a simple alternation of literal patterns—if we extracted `["foo", "bar", "baz"]`, the resulting regex is essentially `foo|bar|baz`.

The returned regex is built with `utf8_empty(false)`, allowing it to match empty strings if present (though quality heuristics should have filtered those out). The regex is constructed by converting each literal back into an HIR node and combining them with `Hir::alternation`. This might seem circular—we started with a regex, extracted literals, and now we're building another regex—but the resulting regex is dramatically simpler and will compile to much faster search code.

The `Some` / `None` return type reflects that extraction might not produce anything useful. If the sequence is infinite (represented as `None` from `literals()`) or empty, there's nothing to build. Callers must handle both cases, falling back to the original regex when no prefilter is available.

See: Companion Code Section 9

---

## Section 10: Connecting to the Configuration Pattern

This file participates in ripgrep's broader **Builder Pattern Ecosystem** in a supporting role. The `ConfiguredHIR` type passed to `InnerLiterals::new` comes from the configuration infrastructure explored in earlier lessons. It bundles together the regex's HIR with the configuration options that affect how the regex should be matched.

The configuration dependency is most visible in the line terminator check. The `chir.config().line_terminator` access retrieves the configured line terminator (usually `\n`), and its absence disables the entire optimization. This is a concrete example of how configuration flows through the codebase—a command-line flag like `--null-data` (which uses `\0` as the line terminator) changes this configuration, which in turn affects whether inner literal optimization is valid.

The logging statements throughout this file (using `log::trace!` and `log::debug!`) connect to ripgrep's broader observability story. When troubleshooting performance, you can enable trace logging to see exactly what literals were extracted, whether they passed quality checks, and what fast regex was built. This introspection is invaluable for understanding why a particular search is fast or slow.

See: Companion Code Section 10

---

## Section 11: The Test Suite Philosophy

The comprehensive test suite at the end of the file serves as both verification and documentation. Each test encodes a specific expectation about how literal extraction should behave for a given pattern. Reading through these tests reveals the nuanced correctness requirements.

The tests use helper functions `E` and `I` to create exact and inexact literals concisely. The patterns range from simple literals (`r"foo"`) through complex combinations of repetitions, alternations, and character classes. Many tests verify edge cases: empty patterns, impossible patterns (like `[a&&b]` which can never match), and patterns with unusual quantifier combinations.

The `heuristics` tests are particularly interesting because they verify the quality-based selection. The test `r"[a-z]+(ab|cd|ef)[a-z]+hiya[a-z]+"` demonstrates that the extractor should prefer `"hiya"` over `["ab", "cd", "ef"]` because a single longer literal is better than three shorter ones. This kind of test documents intended behavior that might otherwise be opaque from the implementation alone.

See: Companion Code Section 11

## Key Takeaways

1. **Inner literal extraction enables line-oriented pre-filtering**: By finding strings that must appear in any match and searching for them with fast SIMD routines, ripgrep can skip vast portions of input that could never match.
2. **Correctness depends on search semantics**: The optimization is only valid for line-by-line searching; without line boundaries, you can't safely narrow the regex search after finding a literal.
3. **Combinatorial limits prevent extraction from becoming expensive**: Character class expansion and repetition unrolling can explode exponentially, so hard limits ensure extraction stays fast.
4. **Quality heuristics are essential**: Not all literal sets are helpful—short literals or common bytes can match so frequently that filtering provides no benefit or even hurts performance.
5. **The "choose best" strategy handles gaps in patterns**: When parts of a regex don't yield literals, the extractor tracks multiple candidates and selects the most useful one.

6. **Exactness tracking enables correct composition**: Knowing whether a literal represents the complete match or just part of it is essential for correctly combining literals across concatenations and alternations.

---

## What to Read Next

How does the regex matcher actually use these extracted literals? Look back at `crates/regex/src/matcher.rs` to see how `InnerLiterals` integrates with the matching infrastructure.

What's the HIR that this file analyzes? The `regex_syntax::hir` module defines the High-level Intermediate Representation that all these extraction methods traverse.

How does configuration reach this code? Review `crates/regex/src/config.rs` to understand the `ConfiguredHIR` type and how options like line terminators flow through the system.

Want to understand the Sink trait that receives matches found through this optimization? See `crates/searcher/src/sink.rs` from an earlier lesson.

# ripgrep crates/regex/src/ast.rs: AST Manipulation

## What This File Does

This file provides Abstract Syntax Tree analysis for regular expressions, enabling ripgrep's "smart case" feature. When you search for `foo`, ripgrep automatically makes the search case-insensitive, but when you search for `Foo`, it respects the uppercase and stays case-sensitive. This file contains the logic that makes that determination possible.

The implementation demonstrates a common pattern in compiler and language tool design: rather than trying to analyze raw pattern strings with string manipulation, ripgrep parses the regex into a structured tree and then walks that tree to extract semantic information. This approach is more robust and handles edge cases that simple string scanning would miss, like distinguishing between a literal `S` and the escape sequence `\S`.

---

## Section 1: The Analysis Result Structure

The heart of this file is the `AstAnalysis` struct, which captures exactly two pieces of information about a regex pattern: whether it contains any uppercase literal characters, and whether it contains any literals at all. These two boolean flags are all ripgrep needs to implement smart case matching.

The design is deliberately minimal. Rather than trying to capture every possible characteristic of a regex, this analysis focuses only on what's needed for the immediate use case. This is a good example of YAGNI (You Aren't Gonna Need It) in practice—the analysis could track many more properties, but doing so would add complexity without benefit.

The `Clone` and `Debug` derives are standard for configuration-like structs that might need to be copied or inspected during debugging. The `pub(crate)` visibility keeps this as an internal implementation detail of the regex crate, not part of its public API.

See: Companion Code Section 1

---

## Section 2: The Entry Points

The struct provides two ways to construct an analysis: from a pattern string or from an already-parsed AST. The `from_pattern` method handles the full pipeline—parsing the regex string into an AST and then analyzing it. Notice that it's marked `#[cfg(test)]`, meaning it only exists in test builds. This is because in production, ripgrep has usually already parsed the regex elsewhere and can pass the AST directly.

The `from_ast` method is the production entry point. It creates a fresh analysis, runs the tree traversal, and returns the result. This separation between "get me an AST somehow" and "analyze this AST" is clean because it avoids duplicating the analysis logic and lets callers decide where the AST comes from.

The `from_pattern` returning `Option<AstAnalysis>` rather than `Result` is interesting—it collapses all parse errors into `None`. This works because the analysis is typically a "nice to have" optimization; if the pattern doesn't parse, ripgrep will discover that later anyway when it tries to compile the regex for matching.

See: Companion Code Section 2

---

## Section 3: The Accessor Methods

The accessor methods `any_uppercase` and `any_literal` follow Rust convention: they take `&self`, return the field values directly, and have names that read naturally as yes/no questions. The documentation comments are particularly helpful here because they clarify edge cases.

The `any_uppercase` documentation explains that `\pL` (a Unicode property escape meaning "any letter") doesn't count as containing an uppercase character, even though the `L` in the pattern is uppercase and the class matches uppercase letters. This is crucial: the analysis cares about **literal characters that the user typed**, not about what a regex class might match. Similarly, `any_literal` returns false for `\pL` because there's no literal text to match—it's all regex metacharacters.

This distinction is what makes smart case work correctly. If someone writes `\pL` to match any Unicode letter, they probably don't want case sensitivity forced on just because they wrote a capital L in the escape sequence.

See: Companion Code Section 3

## Section 4: The Tree Traversal Strategy

The `from_ast_impl` method implements recursive tree traversal with an important optimization: it checks `done()` before processing each node. If the analysis has already found both uppercase characters and literals, there's no point continuing—both flags are set to `true` and can never change back.

This early termination pattern is common in tree algorithms where you're searching for properties that, once found, remain found. It transforms what would be an O(n) traversal into a best-case O(1) operation when the interesting patterns appear early in the regex.

The match expression categorizes AST nodes into three groups: nodes that don't contain literals (like `Dot` or `Assertion`), nodes that contain literals directly (the `Literal` variant), and nodes that contain sub-trees requiring recursive traversal (like `Repetition`, `Group`, `Alternation`, and `Concat`). This exhaustive handling ensures no regex construct can slip through without being analyzed.

See: Companion Code Section 4

---

## Section 5: Handling Character Classes

Character classes like `[a-zA-Z]` require special handling because they have their own internal structure. The `from_ast_class_set` and `from_ast_class_set_item` methods mirror the structure of the AST for character classes, recursively descending into unions, ranges, and nested brackets.

The key insight is that character class ranges like `A-Z` are stored as literals for their start and end points. When you write `[A-Z]`, the AST contains literal `A` and literal `Z`, and the analysis correctly identifies these as uppercase characters. This means a pattern like `foo[A-Z]` will trigger case-sensitive matching because the character class contains uppercase literal boundaries.

Binary operations on character classes (intersection, subtraction) are handled by recursively analyzing both sides. This ensures that complex class expressions like `[[a-z]&&[^aeiou]]` are fully traversed.

See: Companion Code Section 5

## Section 6: Literal Detection

The `from_ast_literal` method is where the actual detection happens, and it's remarkably simple: just two lines that set `any_literal` to true and check if the character is uppercase. The method

uses Rust's `is_uppercase()` method on `char`, which handles Unicode correctly—accented uppercase characters like `É` are properly recognized.

The simplicity here is enabled by all the infrastructure above it. The tree traversal funnels every literal in the entire regex—whether it appears in a concatenation, inside a character class range, or within a capturing group—through this single method. This is the power of working with structured data: once you have an AST, operations become straightforward.

See: Companion Code Section 6

---

## Section 7: The Early Termination Condition

The `done` method encapsulates the termination condition: once both `any_uppercase` and `any_literal` are true, further analysis cannot change the result. This is a monotonic property—both flags can only transition from false to true, never backwards.

This pattern appears frequently in program analysis and type checking. When you're computing properties that can only accumulate (like collecting all the types used in a program, or all the variables referenced), you can often stop early once you've found enough information.

The condition being `&&` rather than `||` is important. If we stopped when **either** flag became true, we'd miss cases where we find a literal first but haven't yet found an uppercase character. We need both pieces of information before we can stop.

See: Companion Code Section 7

---

## Section 8: Test Coverage and Edge Cases

The test module demonstrates the nuances of the analysis through carefully chosen examples. The tests progress from simple cases (empty pattern, lowercase only, uppercase present) to increasingly subtle distinctions that verify the analysis correctly handles regex syntax.

The test for `foo\S` versus `foo\\S` is particularly illuminating. The first pattern contains a Perl character class (non-whitespace), and the `S` is part of the escape sequence, not a literal. The second pattern contains an escaped backslash followed by a literal `S`, so it should (and does) trigger `any_uppercase`. Getting this right requires working at the AST level; string-based analysis would struggle to reliably distinguish these cases.

The test for `\p{Ll}` (Unicode lowercase letters) confirms that Unicode property escapes don't register as literals, while `a\u0061` shows that Unicode escape sequences for specific characters do count as literals. These tests document the expected behavior and protect against regressions.

See: Companion Code Section 8

---

## Section 9: Connecting to the Builder Pattern Ecosystem

This analysis module fits into ripgrep's larger configuration system. Looking back at the `config.rs` file you studied earlier, the `Config` struct has a `case_smart` option. When enabled, ripgrep uses this AST analysis to decide whether a pattern should be case-insensitive.

The flow works like this: the user provides a pattern, the pattern gets parsed into an AST, this analysis runs on the AST, and if smart case is enabled and no uppercase literals were found, the configuration adds case-insensitivity flags to the regex compilation. This is why the analysis is `pub(crate)`—it's an internal detail used by the configuration layer, not something external code needs to access.

This represents the Builder Pattern extended into runtime: configuration isn't just about what the user explicitly requested, but also about intelligent defaults derived from analyzing their input. The builder accumulates not just user choices but also inferred settings.

See: Companion Code Section 9

---

## Key Takeaways

First, working with ASTs is fundamentally more robust than string manipulation when analyzing structured languages like regular expressions. The tree representation separates syntax from semantics, making it easy to distinguish `\S` (an escape) from `S` (a literal).

Second, early termination in tree traversals can significantly improve performance when searching for monotonic properties—ones that, once found, remain true regardless of what else is discovered.

Third, minimal data structures that capture only what's needed (two booleans here) are easier to reason about and less likely to accumulate bugs than comprehensive analyses that track everything possible.

Fourth, the `#[cfg(test)]` attribute lets you include helper functions that are only compiled for testing, keeping production binaries lean while still allowing convenient test utilities.

Fifth, Rust's exhaustive pattern matching on enums ensures that when the AST adds new node types, the compiler will flag any analysis code that doesn't handle them—a powerful safety net for maintainability.

Sixth, this analysis demonstrates how configuration systems can be intelligent: rather than forcing users to specify every option, ripgrep infers sensible defaults from the structure of their input.

---

## What to Read Next

How does ripgrep actually use this analysis to modify regex compilation? Revisit `crates/regex/src/config.rs` and look for where `AstAnalysis` gets called.

What other information could be extracted from regex ASTs? The `regex-syntax` crate's documentation shows the full AST structure and suggests other analyses that might be useful.

How do the literal optimizations from lesson 5 relate to this AST analysis? Both examine regex structure to enable optimizations, but they work at different levels of the compilation pipeline.

# ripgrep crates/searcher/src/lib.rs: Searcher Overview

## What This File Does

This file serves as the public interface for ripgrep's searcher crate, the component responsible for orchestrating the actual search process across files and input streams. While previous lessons explored how ripgrep builds and configures regex matchers, this crate answers a different question: how do you efficiently read bytes from a source, apply pattern matching, and report results in a way that handles all the complexity of real-world text search?

The searcher crate sits at a crucial architectural boundary. It consumes matchers (like the regex implementation we studied) and produces results for consumers (like the printer crate). This lib.rs file establishes the crate's public API through careful re-exports and module organization, following a pattern we'll see repeatedly throughout ripgrep's codebase.

## Section 1: The Crate Documentation as Architecture Document

The module-level documentation comment (`//!`) in this file isn't just helpful prose—it's an architectural specification. Rust's documentation conventions encourage developers to treat these comments as the primary way users discover how a crate works. The searcher crate's documentation introduces three core concepts that define its entire design: the `Searcher`, the `Matcher` trait, and the `Sink` trait.

Notice how the documentation describes a clear data flow: bytes come from a source, pass through a `Searcher` that applies a `Matcher`, and results flow to a `Sink`. This three-part architecture creates clean separation of concerns. The `Searcher` doesn't need to know what pattern system you're using—it just needs something that implements `Matcher`. Similarly, it doesn't care whether results go to stdout, a file, or an in-memory buffer—it just needs a `Sink`.

This design reflects a principle you'll see throughout well-designed Rust libraries: define traits at boundaries where multiple implementations make sense. The `Matcher` trait allows ripgrep to use different regex engines. The `Sink` trait allows different output formats without changing the search logic.

See: Companion Code Section 1

---

## Section 2: Public API Design Through Re-exports

The `pub use` declarations near the middle of the file reveal how Rust crates carefully curate their public interfaces. Rather than exposing the internal module structure directly, this file re-exports specific types from internal modules, creating a flat and intentional public API.

The `crate::` prefix in these paths refers to the current crate's root—a common pattern you'll see in Rust 2018 edition and later. Each re-export makes a deliberate choice about what users of this crate need to access. The `lines` module contributes `LineIter` and `LineStep` for iterating over lines. The `searcher` module provides the core `Searcher` type along with `SearcherBuilder` and configuration types like `BinaryDetection` and `MmapChoice`.

This pattern of selective re-export offers significant advantages. Internal modules can have whatever structure makes implementation clearest, while the public API remains stable and focused. Users write `grep_searcher::Searcher` rather than `grep_searcher::searcher::Searcher`—a cleaner path that doesn't leak implementation details.

See: Companion Code Section 2

---

## Section 3: The Builder Pattern Ecosystem

The re-export of `SearcherBuilder` alongside `Searcher` signals that this crate participates in ripgrep's consistent use of the builder pattern. You've already seen this pattern in the regex configuration code, where `RegexMatcherBuilder` accumulates settings before constructing a `RegexMatcher`. The searcher crate follows the same philosophy.

The builder pattern solves a genuine problem in Rust: types with many optional configuration settings. Without builders, you'd either have constructors with many parameters (error-prone, hard to read) or make all fields public (loses encapsulation, makes invariants hard to maintain). Builders let you chain configuration calls in any order and validate the final configuration when building.

What makes this pattern especially powerful in ripgrep is its consistency across crates. Whether you're configuring a regex matcher, a searcher, a printer, or a directory walker, the mental model is the same: create a builder, call configuration methods, call `build()`. This consistency reduces cognitive load when working with the codebase.

See: Companion Code Section 3

---

## Section 4: The Sink Abstraction

The `Sink` trait and its supporting types form one of the most important abstractions in the searcher crate. The re-exports show us the pieces: `Sink` itself, plus `SinkContext`, `SinkContextKind`, `SinkError`, `SinkFinish`, and `SinkMatch`. This collection of types suggests a rich callback-based API for receiving search results.

A `Sink` is essentially a collection of callback methods that the `Searcher` invokes during the search process. When the searcher finds a matching line, it calls the sink with a `SinkMatch`. When it encounters context lines (lines before or after a match), it uses `SinkContext` with a `SinkContextKind` to distinguish between "before" and "after" context. The `SinkFinish` type carries end-of-search summary information.

This design inverts the typical control flow. Instead of the searcher returning a collection of results that the caller processes, the searcher pushes results to the sink as they're discovered. This inversion enables streaming results—crucial for searching large files where you don't want to accumulate all matches in memory before processing them.

See: Companion Code Section 4

---

## Section 5: Convenience Sinks and Closures

The re-export of `sinks` as a module (not just individual types) indicates there's a collection of ready-made sink implementations for common use cases. The example in the documentation demonstrates one: `UTF8`, which wraps a closure and handles the common case of processing matches as UTF-8 strings.

This pattern of providing both a trait for full control and convenience implementations for simple cases appears throughout Rust libraries. The `Sink` trait gives you complete flexibility—you can build a complex printer that handles colors, line numbers, and multiple output formats. But if you just want to collect matches into a vector, the `UTF8` sink lets you do that with a simple closure.

The closure-based approach also demonstrates Rust's **`Fn` traits** in action. The `UTF8` sink takes a closure that receives a line number and a string slice, returning a `Result<bool, _>`. The boolean return value creates an early-exit mechanism: return `Ok(false)` to stop the search, `Ok(true)` to continue.

See: Companion Code Section 5

---

## Section 6: The Documentation Example Dissected

The example in the module documentation isn't just a usage demonstration—it's a teaching tool that shows how ripgrep's crates compose together. It imports from three crates: `grep_matcher` for the trait, `grep_regex` for the implementation, and `grep_searcher` for the search machinery.

The flow starts by constructing a `RegexMatcher`, then creates a `Searcher` with default settings (via `Searcher::new()`), and finally calls `search_slice` to search an in-memory byte slice. The `UTF8` sink receives each matching line along with its line number, and the closure extracts the specific match text.

Notice the nested matching: the searcher finds lines that contain matches, but the closure re-runs the matcher on each line to extract the precise match boundaries. This layering reflects the searcher's design philosophy—it handles the complexity of reading files, buffering, and line iteration, while leaving detailed match extraction to the caller when needed.

See: Companion Code Section 6

---

## Section 7: Module Organization and Visibility

The bottom of the file declares the crate's internal module structure. The `#[macro_use]` attribute on the `macros` module makes any macros defined there available throughout the crate without explicit imports. The remaining modules—`line_buffer`, `lines`, `searcher`, and `sink`—organize the implementation.

The `testutil` module has `#[cfg(test)]` applied, meaning it only compiles during test runs. This pattern keeps test utilities available for internal tests without including them in the release binary.

None of these modules have `pub` visibility, making them private to the crate. Users of the searcher crate interact only with the types re-exported at the crate root. This encapsulation means the internal module structure could change completely—splitting modules, renaming them, reorganizing code—without breaking any external code that depends on the public API.

See: Companion Code Section 7

## Section 8: Enforcing Documentation with Lint Attributes

The `#![deny(missing_docs)]` attribute at the top of the file turns missing documentation into a compile error. This isn't just a style preference—it's an enforcement mechanism that ensures every public item in the crate has documentation.

In a crate that defines important traits like `Sink`, comprehensive documentation matters enormously. Users implementing the trait need to understand not just the method signatures but the semantic contract—when methods get called, what invariants they can rely on, what the return values mean.

The `#![...]` syntax (with the exclamation mark) makes this a **crate-level attribute** that applies to the entire crate, not just the current file. This single line establishes a documentation requirement that affects every public function, struct, enum, trait, and method throughout the crate.

See: Companion Code Section 8

---

## Section 9: Configuration Types as API Surface

The re-exported configuration types—`BinaryDetection`, `ConfigError`, `Encoding`, and `MmapChoice`—reveal the searcher's configurability. Each type represents a dimension of search behavior that users can customize.

`BinaryDetection` controls how the searcher handles files that appear to contain binary data. Should it search them anyway, skip them, or stop when it detects binary content? `MmapChoice` determines when to use memory-mapped file I/O, which can dramatically affect performance for large files. `Encoding` handles character encoding considerations.

The presence of `ConfigError` indicates that configuration can fail—not all combinations of settings may be valid. This follows a pattern of fail-fast validation: rather than silently accepting invalid configuration and producing confusing runtime behavior, the builder reports configuration errors when you try to build an invalid searcher.

See: Companion Code Section 9

## Section 10: The Searcher's Responsibilities

The documentation enumerates what the `Searcher` handles: consuming bytes from a source, applying a `Matcher` efficiently, managing line iteration, inverting searches (finding non-matching lines), counting lines, reporting contextual lines, detecting binary data, and deciding whether to use memory maps. This list reveals the complexity hidden behind the simple `search_slice` or `search_reader` API.

Each responsibility could be its own substantial piece of code. Line counting must handle different line terminators. Context lines require buffering lines before and after matches. Binary detection needs heuristics for identifying non-text content. Memory map decisions involve considering file size, operating system, and available memory.

By centralizing all this logic in the `Searcher`, ripgrep ensures consistent behavior across different search modes and output formats. A `Sink` implementation doesn't need to worry about any of these concerns—it just receives already-processed results with line numbers, context, and match information ready to use.

See: Companion Code Section 10

---

## Key Takeaways

First, this lib.rs file demonstrates how Rust crates use selective re-exports to create clean, stable public APIs while keeping implementation details private—users see a flat namespace of carefully chosen types.

Second, the three-part architecture of `Searcher`, `Matcher`, and `Sink` creates clear separation of concerns through trait boundaries, enabling different implementations of pattern matching and result handling to compose seamlessly.

Third, the builder pattern appears here as part of ripgrep's ecosystem-wide consistency—`SearcherBuilder` follows the same conventions as `RegexMatcherBuilder` and other builders throughout the codebase.

Fourth, the `Sink` trait demonstrates callback-based APIs for streaming results, avoiding the memory overhead of collecting all results before processing them.

Fifth, the `#![deny(missing_docs)]` attribute shows how Rust's lint system can enforce documentation standards at compile time, ensuring public APIs remain well-documented.

Sixth, the module documentation serves as both user guide and architectural specification, establishing the mental model that users need to work with the crate effectively.

## What to Read Next

How does the `Searcher` actually configure itself? Read `crates/searcher/src/searcher/mod.rs` to explore the `SearcherBuilder` implementation and understand all the configuration options.

What does implementing a `Sink` look like in practice? Read `crates/printer/src/standard.rs` to see how ripgrep's standard printer builds sophisticated grep-like output through the `Sink` interface.

How does the searcher manage line buffers and handle large files efficiently? Read `crates/searcher/src/line_buffer.rs` to understand the buffering strategy that enables streaming search.

What exactly is the `Matcher` trait that searchers consume? Revisit `crates/matcher/src/lib.rs` to understand the interface contract that connects searchers to pattern implementations.

# ripgrep crates/searcher/src/line_buffer.rs: Efficient Line Reading

## What This File Does

This file implements ripgrep's line buffer, the critical infrastructure that transforms arbitrary byte streams into line-oriented data suitable for pattern matching. While it might seem like reading lines from a file should be straightforward, doing it efficiently at scale requires careful buffer management, memory allocation strategies, and handling of edge cases like binary data and lines longer than the buffer itself.

The line buffer sits at the heart of ripgrep's searcher architecture. Every file that ripgrep searches flows through this code, making its performance characteristics directly impact overall search speed. The design prioritizes minimal memory allocation, efficient buffer reuse, and the ability to handle pathological inputs without crashing or consuming unbounded memory.

## Section 1: The Builder Pattern in Action

This file showcases a mature implementation of the builder pattern, one that connects directly to the ecosystem of builders we've seen throughout ripgrep. The `LineBufferBuilder` follows the same conventions as `ConfiguredHIR` in the regex crate and the searcher builder we examined previously, creating a consistent API surface across the entire codebase.

The builder here manages a `Config` struct that captures all the fixed options for a line buffer: capacity, line terminator, allocation strategy, and binary detection mode. Notice how the config is marked with `#[derive(Clone, Copy, Debug)]`—because all its fields are either primitives or small enums, the entire configuration can be copied cheaply. This means builders can be cloned and modified without worrying about expensive deep copies.

Each builder method returns `&mut LineBufferBuilder`, enabling method chaining while still allowing the builder to be used incrementally. The `build` method consumes the configuration by copying it into the new `LineBuffer`, not by moving it. This subtle choice means you can call `build()` multiple times on the same builder to create multiple identically-configured buffers—useful when processing many files in parallel.

See: Companion Code Section 1

---

## Section 2: Memory Allocation Strategies

One of the most interesting design decisions in this file is the `BufferAllocation` enum, which lets callers choose how the buffer should behave when it encounters lines that don't fit in the initial capacity. This is a classic example of surfacing a trade-off to the API consumer rather than making an opinionated choice.

The `Eager` variant tells the buffer to grow without limit, allocating as much memory as needed to fit the current line. This is the default and works well for typical text files where lines are reasonably sized. However, it creates a potential denial-of-service vector: a maliciously crafted file with a multi-gigabyte "line" could exhaust system memory.

The `Error(usize)` variant provides a safety valve. It specifies the maximum additional memory the buffer may allocate beyond its initial capacity. If a line requires more, the buffer returns an error rather than continuing to allocate. The limit applies to *additional* memory, not total memory, which means a limit of zero is sensible—it guarantees the buffer never grows beyond its initial size.

This design reflects ripgrep's philosophy of being safe by default while allowing power users to tune behavior for their specific environments. A security-conscious deployment might set strict limits, while interactive use can rely on the more permissive defaults.

See: Companion Code Section 2

---

## Section 3: Binary Detection Heuristics

Text search tools face an interesting challenge: what should happen when you encounter binary data? Searching binary files with text patterns often produces garbage matches and can significantly slow down operations. The `BinaryDetection` enum encapsulates three different approaches to this problem.

`BinaryDetection::None` takes the simplest approach—treat everything as text, binary or not. This is the default because binary detection is inherently heuristic. There's no foolproof way to distinguish binary from text; you're really asking "does this data contain bytes that are unlikely in text files?"

`BinaryDetection::Quit(u8)` implements early termination. When the specified byte is encountered (typically a null byte, `\x00`), the buffer acts as if it reached end-of-file. Any data after the binary marker is never seen by the caller. This is useful when you want to completely skip binary files but have already started reading before realizing the file is binary.

`BinaryDetection::Convert(u8)` takes a more permissive approach. Binary bytes are replaced with the line terminator, effectively converting them into line breaks. This allows the search to continue while ensuring the problematic bytes never appear in output. The buffer records where binary data was first found, enabling callers to warn users that results may be incomplete.

The implementation carefully tracks the `binary_byte_offset`, recording only the first occurrence. This offset is an absolute position relative to the start of the input stream, not a position in the current buffer, making it suitable for user-facing error messages.

See: Companion Code Section 3

---

## Section 4: The Buffer State Machine

The `LineBuffer` struct maintains surprisingly complex state, tracking multiple positions within a single byte vector. Understanding these positions is essential for understanding how the buffer operates.

The `pos` field marks where unconsumed data begins. Everything before `pos` has already been processed by the caller. The `last_lineterm` field marks where the last complete line ends—everything between `pos` and `last_lineterm` is what `buffer()` returns. The `end` field marks where valid data ends; bytes between `last_lineterm` and `end` represent an incomplete line that shouldn't be exposed yet.

This three-position design elegantly handles the fundamental challenge of line-oriented buffering: when you read data, you typically get an incomplete line at the end. You can't give that partial line to your caller because they expect complete lines, but you also can't discard it because it contains data you'll need after the next read.

The `absolute_byte_offset` field tracks a logical position that may have no correspondence to any memory location. It represents "how many bytes into the input stream are we?" This grows monotonically as data is consumed, even as the physical buffer wraps around and reuses space.

See: Companion Code Section 4

---

## Section 5: The Fill and Roll Cycle

The `fill` method implements the core read loop, and its structure reveals careful attention to both correctness and performance. The method reads data from the underlying reader, performs binary detection, and locates line terminators—all while managing the complex state we just discussed.

The first thing `fill` does is check whether binary detection has already triggered an early EOF. If so, it reports whether any unconsumed data remains without attempting to read more. This early return is crucial for the `Quit` behavior to work correctly.

Next comes the `roll` operation, which moves unconsumed data to the front of the buffer. This is where the magic of buffer reuse happens. Rather than allocating new buffers for each read, ripgrep recycles the same buffer by moving leftover bytes to position zero and reading new data into the freed space. The assertion `assert_eq!(self.pos, 0)` after rolling serves as a sanity check—rolling should always reset the position.

The read loop continues until either a line terminator is found or EOF is reached. This is important: the buffer doesn't return after every read. It keeps reading until it can guarantee that `buffer()` will return at least one complete line, or until the input is exhausted. This batching behavior reduces the overhead of context-switching between the buffer and its caller.

See: Companion Code Section 5

---

## Section 6: Efficient Line Terminator Search

Within the fill loop, the search for line terminators uses `rfind_byte` rather than `find_byte`. This seemingly minor choice has significant performance implications.

When searching for the last line terminator in newly read data, you want to find the rightmost one. Finding the leftmost would work for correctness, but then you'd be returning smaller chunks to the caller—potentially just one line at a time. By finding the rightmost, the buffer returns as much complete data as possible in each call, reducing the total number of fill-consume cycles.

The binary detection code, in contrast, uses `find_byte` because it wants to find the *first* occurrence of the binary byte. For `Quit` behavior, this determines where to truncate. For `Convert` behavior, the `replace_bytes` helper function needs to find and replace all occurrences, starting from the first.

This attention to which direction to search—left-to-right versus right-to-left—appears throughout ripgrep and reflects a deep understanding of how these low-level decisions compound into significant performance differences at scale.

See: Companion Code Section 6

---

## Section 7: The Roll Operation

The `roll` method deserves special attention because it's the key to memory efficiency. Without rolling, either the buffer would need to be enormous (to hold all data simultaneously) or it would need to allocate fresh buffers constantly (expensive and cache-unfriendly).

The implementation handles two cases. When all data has been consumed (`pos == end`), it simply resets all positions to zero—no data movement required. When unconsumed data exists, it uses `copy_within` to move bytes from the tail of the buffer to the head. This is a single `memmove` operation, which modern CPUs execute very efficiently.

After rolling, `pos` is always zero, and `last_lineterm` equals `end`. This invariant means that after a roll, all unconsumed data is a partial line waiting for more input. The buffer is in a clean state, ready for the next read.

The assertion that rolling is idempotent (calling it twice produces the same result as calling it once) isn't explicitly checked in code, but the logic ensures it. If you roll when already at position zero, the copy operation has source and destination at the same location, producing no change.

See: Companion Code Section 7

---

## Section 8: Dynamic Capacity Growth

The `ensure_capacity` method manages buffer growth when the current capacity is insufficient. Its implementation reflects the allocation strategy configured via `BufferAllocation`.

When growing eagerly, the buffer doubles in size. Doubling is a standard amortized strategy that ensures the total allocation work is O(n) for n bytes of input—each byte is "charged" at most O(log n) resize operations. The code carefully handles the edge case where the buffer starts at capacity zero by ensuring the growth is at least one byte.

When using `Error(limit)`, the calculation is more nuanced. It tracks how much additional memory has already been used (`buf.len() - self.config.capacity`) and computes how much remains within the limit. If no growth is possible, it returns an error using the helper function `alloc_error`, which constructs a descriptive `io::Error`.

The final assertion `assert!(!self.free_buffer().is_empty())` is a defensive check ensuring that after `ensure_capacity` succeeds, there's definitely room to read more data. Without this guarantee, the read loop could spin forever.

See: Companion Code Section 8

---

## Section 9: The LineBufferReader Wrapper

`LineBufferReader` provides the public interface for using a line buffer with an actual reader. It combines a reader `R: io::Read` with a mutable reference to a `LineBuffer`, creating a composable design where buffers can be reused across multiple readers.

The lifetime parameter `'b` on `LineBufferReader` indicates that the reader borrows the line buffer. This design choice means you can create a `LineBuffer` once, then repeatedly create `LineBufferReader` instances for different input sources, each time reusing the same underlying buffer. The `clear()` call in the constructor resets all state, ensuring no data from a previous reader leaks through.

The methods on `LineBufferReader` are thin wrappers that delegate to `LineBuffer`. This separation of concerns keeps the buffer's core logic independent of the reader abstraction, making both easier to test. Notice how several methods are marked `#[cfg(test)]`—they exist only for testing convenience and aren't part of the production API.

See: Companion Code Section 9

---

## Section 10: Binary Byte Replacement

The `replace_bytes` function implements an optimized approach to replacing bytes in a slice. Rather than a naive loop that checks every byte, it uses `find_byte` (implemented via memchr) to quickly skip over regions without the target byte.

The function returns the offset of the first replacement, if any. This information is used by the caller to track binary byte offsets. Returning an `Option<usize>` rather than a boolean provides strictly

more information without additional cost.

The inner loop deserves attention. After finding and replacing a byte, the code has a special case for runs of consecutive target bytes. When binary data contains null bytes, they often appear in long sequences (think padding in binary formats). For these runs, comparing byte-by-byte is faster than repeatedly calling `find_byte`, which has setup overhead. The comment explains this optimization explicitly, a practice that makes the non-obvious code maintainable.

The early return when `src == replacement` is another optimization. If you're "replacing" a byte with itself, the operation is a no-op, so the function returns `None` immediately without scanning the data.

See: Companion Code Section 10

---

## Section 11: Testing Philosophy

The test module reveals how the ripgrep team approaches testing buffer behavior. Tests cover basics, edge cases, and the interaction between different configuration options.

The `SHERLOCK` constant provides a realistic text sample for testing, while simpler inputs like `"homer\nlisa\nmaggie"` make it easy to reason about positions and offsets. Tests verify not just the returned data but also metadata like `absolute_byte_offset()` and `binary_byte_offset()`.

Particularly valuable are the tests for pathological cases: zero capacity buffers, buffers that hit allocation limits, and various positions of binary bytes (beginning, middle, end). The `buffer_limited_capacity1` test shows that even when an allocation error occurs, the buffer remains usable—you can recover partial data and continue processing. This robustness is essential for a tool that processes millions of files where some might trigger edge cases.

The use of `bstr()` for assertions leverages the `bstr` crate's nice printing for byte strings, making test failures easier to debug than raw byte slice comparisons.

See: Companion Code Section 11

---

## Key Takeaways

First, buffer management in line-oriented I/O requires tracking multiple positions: where consumption starts, where complete lines end, and where valid data ends. These three positions enable the fill-consume-roll cycle that makes buffer reuse possible.

Second, the allocation strategy trade-off—eager versus limited—exemplifies how ripgrep surfaces important decisions to callers rather than making assumptions about the right behavior for all scenarios.

Third, binary detection is inherently heuristic, and providing multiple strategies (none, quit, convert) lets users choose the behavior that best fits their use case. The abstraction of tracking where binary data was first seen supports good error messages.

Fourth, searching for the rightmost line terminator rather than the leftmost is a subtle optimization that reduces the number of fill cycles by returning more data per call.

Fifth, the `replace_bytes` function demonstrates how to optimize for common patterns in data (runs of consecutive bytes) while still handling the general case correctly.

Sixth, testing edge cases like zero capacity, allocation limits, and binary bytes at every position ensures the buffer behaves correctly in pathological scenarios that will inevitably occur when processing millions of files.

---

## What to Read Next

How does the searcher coordinate multiple passes over the buffer when context lines are needed? Read `crates/searcher/src/searcher/mod.rs` to see how line buffers integrate with the higher-level search logic.

What drives the 64KB default buffer size? Read `crates/core/app.rs` to see how buffer configuration connects to command-line arguments.

How does ripgrep decide whether to use memory mapping versus line-buffered reading? Read `crates/searcher/src/searcher/mmap.rs` to understand the alternative code path.

# ripgrep crates/searcher/src/lines.rs: Line Handling Utilities

## What This File Does

This file provides the fundamental building blocks for working with lines of text in ripgrep's search engine. While the line buffer from the previous lesson handles efficient reading of data into memory, this module answers a different question: once you have bytes in memory, how do you efficiently find and navigate line boundaries?

The utilities here serve multiple purposes throughout the searcher. They iterate over lines in a buffer, count line occurrences for statistics, expand a match to include its complete surrounding line, and navigate backwards through text to find preceding lines for context display. These operations might seem simple, but they're called millions of times during a typical search, so their implementation emphasizes both correctness and performance.

## Section 1: The Two Iterator Designs

The file presents two different approaches to line iteration, and understanding why both exist reveals an important pattern in systems programming. The first is `LineIter`, a conventional Rust iterator that borrows the byte slice and yields references to each line. The second is `LineStep`, an unusual "explicit" iterator that doesn't borrow the bytes at all—callers must pass the bytes to each `next` call.

Why would you want an iterator that doesn't hold a reference to what it's iterating? The answer lies in Rust's borrowing rules. When `LineIter` borrows the byte slice, that borrow persists for the iterator's entire lifetime. This means you cannot modify the underlying buffer or use it in other ways while iterating. In many search scenarios, the code needs to read lines while simultaneously managing the buffer that contains them—perhaps refilling it or adjusting pointers. `LineStep` allows this by separating the iteration state (line terminator and position) from the data being iterated.

This is a common pattern in performance-critical Rust code: when standard borrowing becomes too restrictive, you can often restructure your API to pass data explicitly rather than holding

references. The trade-off is ergonomics—callers must be careful to pass the same slice each time—but the flexibility is essential for complex buffer management.

See: Companion Code Section 1

---

## Section 2: The Line Terminator Philosophy

Notice that line iteration uses a single `u8` byte as the line terminator, not the full `LineTerminator` type from `grep_matcher`. This design decision reflects a performance optimization throughout ripgrep. Most of the time, lines end with a single newline character (`\n`), and searching for a single byte is faster than searching for a multi-byte sequence.

The `LineTerminator` type in the matcher crate handles the complexity of CRLF (`\r\n`) line endings, but at the low-level iteration layer, we only need to find the final byte of the terminator sequence. Whether the line ends with `\n` or `\r\n`, finding the `\n` gives us the line boundary. The `without_terminator` function later handles stripping the complete terminator (including any `\r`) when needed.

This separation of concerns—fast single-byte search for boundaries, full terminator handling only when displaying—appears throughout ripgrep's architecture. It's a reminder that abstraction layers should match performance requirements, not just logical organization.

See: Companion Code Section 2

---

## Section 3: The Core Iteration Logic

The heart of line iteration lives in `LineStep::next_impl`. This function searches for the next line terminator using `find_byte`, a highly optimized SIMD-accelerated search from the `bstr` crate. When found, it returns the range from the current position to just after the terminator. When no terminator exists, it handles the final line of input—text that ends without a terminator.

The implementation carefully maintains invariants. It first clips the byte slice to the configured end position, ensuring iteration stays within bounds even if the underlying buffer is larger. The position tracking uses an inclusive start and exclusive end pattern, matching Rust's standard range conventions. After each line is yielded, the position advances past the terminator, ready for the next iteration.

The `assert!(m.0 <= m.1)` statements verify that ranges are never inverted. In debug builds, these catch logic errors immediately. In release builds with overflow checks disabled, they compile away,

adding no runtime cost. This is a common Rust idiom for adding safety checks during development that vanish in production.

See: Companion Code Section 3

---

## Section 4: Counting Lines Efficiently

The `count` function demonstrates how ripgrep leverages specialized libraries for common operations. Rather than iterating line by line and counting, it uses `memchr::memchr_iter`, which provides a SIMD-accelerated iterator over all occurrences of a byte in a slice.

The implementation is deceptively simple—just one line—but the `memchr` crate behind it contains thousands of lines of carefully tuned assembly for different CPU architectures. This is ripgrep's performance philosophy in action: don't reinvent algorithms that others have perfected, especially when they've been optimized at the assembly level.

The function returns `u64` for the count, anticipating that large files might contain billions of lines. While this seems unlikely for typical text files, ripgrep sometimes processes binary files or files with very short lines, and the cost of using `u64` over `usize` is negligible compared to the actual counting work.

See: Companion Code Section 4

---

## Section 5: Stripping Line Terminators

The `without_terminator` function addresses a common need: given a line that might end with a terminator, return the content without it. This is essential for display—when printing matches, you typically don't want to include the raw newline bytes in the output.

The implementation uses `saturating_sub` to handle the edge case of lines shorter than the terminator sequence. If `bytes.len()` is 0 and `line_term.len()` is 2, regular subtraction would underflow. `saturating_sub` instead returns 0, and the subsequent comparison correctly handles this case by returning the original bytes unchanged.

The function accepts a `LineTerminator` rather than a raw byte because stripping requires knowing the complete terminator sequence. For CRLF line endings, we need to remove both `\r` and `\n`, not just the final byte. This is where the abstraction from `grep_matcher` becomes necessary—boundary finding needs only the final byte, but content manipulation needs the full sequence.

See: Companion Code Section 5

---

## Section 6: Locating Lines Around Matches

The `locate` function solves a crucial problem: given a match range within a buffer, expand it to include the complete lines containing that match. This is fundamental to displaying search results—users expect to see entire lines, not arbitrary byte ranges.

The algorithm works bidirectionally. First, it searches backward from the match start to find the preceding line terminator (or the buffer start). Then it searches forward from the match end to find the next line terminator (or the buffer end). The result is a range that encompasses all lines touched by the original match.

A subtle optimization handles the case where the match already ends at a line terminator. If `bytes[range.end() - 1] == line_term`, the match ends exactly at a line boundary, so no forward search is needed. This avoids scanning forward into the next line unnecessarily, which matters for performance when processing many matches.

The `rfind_byte` call uses `map_or(0, |i| i + 1)` to handle the no-match case elegantly. If no preceding terminator exists, the line starts at position 0. If one is found at position `i`, the line starts at `i + 1` (just after the terminator). This pattern—using `map_or` to provide a default while transforming the success case—is idiomatic Rust for handling optional results concisely.

See: Companion Code Section 6

---

## Section 7: Navigating Backwards Through Lines

The `preceding` and `preceding_by_pos` functions enable ripgrep's context feature, where you can display lines before and after matches. Unlike forward iteration, which is straightforward, backward navigation through lines requires careful handling of edge cases.

The algorithm starts from a position and counts backward through line terminators. Each terminator found decrements the count until we've passed enough lines. The complexity arises from how positions relate to line boundaries. A position pointing at a line terminator is considered part of the line that terminator ends, so the function first adjusts for this by decrementing when at a terminator.

The loop structure here uses `rfind_byte` to search backward through the buffer. When a terminator is found and the count has reached zero, we've found our target—the line starts just after this terminator. When the count is still positive, we continue searching from before the found terminator. The early return when `i == 0` handles the case where we've reached the buffer start before finding enough lines.

This function's complexity is linear in the number of lines traversed, not the buffer size. For typical context values (a few lines), this is efficient. For pathological cases (requesting thousands of lines of context), performance could degrade, but such usage is rare and the function remains correct.

See: Companion Code Section 7

---

## Section 8: The Test Suite as Documentation

The extensive test suite in this file serves a dual purpose: verifying correctness and documenting expected behavior. The `SHERLOCK` constant provides a realistic multi-line test input, while targeted test functions explore edge cases and boundary conditions.

The `line_locate_weird` test is particularly valuable. It systematically tests empty strings, single newlines, consecutive newlines, and various positions within small strings. These edge cases are where bugs typically hide—the happy path usually works, but what happens when the match spans multiple lines, or starts at a newline, or the input is empty?

The `preceding_lines_*` tests demonstrate the careful thought behind backward navigation. They verify that positions at line terminators behave correctly, that requesting more context than available works, and that empty lines are handled properly. Each test case corresponds to a specific scenario that a user might encounter.

When reading test code like this, pay attention to what scenarios are being tested, not just that tests exist. The test cases reveal the edge cases the authors anticipated and the invariants they expect the code to maintain. This is documentation that the compiler verifies.

See: Companion Code Section 8

## Key Takeaways

First, the dual iterator design—`LineIter` and `LineStep`—demonstrates how to work around Rust's borrowing restrictions when you need more flexibility. By separating iteration state from the data

being iterated, you can maintain control over buffer lifetimes while still providing convenient iteration semantics.

Second, performance-critical code often uses different abstraction levels for different operations. Finding line boundaries uses single bytes for speed, while displaying results uses the full `LineTerminator` type for correctness. Match the abstraction to the performance requirement.

Third, backward navigation through text is significantly more complex than forward iteration. The `preceding_by_pos` function's careful handling of line terminator positions shows how much edge-case logic is required for correct bidirectional traversal.

Fourth, specialized libraries like `memchr` and `bstr` provide SIMD-accelerated implementations of common byte operations. Using these rather than writing custom loops is both faster and more reliable.

Fifth, extensive edge-case testing is essential for line handling utilities. The boundary between lines, the behavior at buffer edges, and the handling of empty lines all require explicit verification. The test suite here serves as executable documentation of expected behavior.

---

## What to Read Next

How do these line utilities get used in actual search operations? Read `crates/searcher/src/searcher/mod.rs` to see how `locate` expands matches and `preceding` provides context lines.

What is the `Match` type that appears throughout this code? Review `grep_matcher/src/lib.rs` to understand how match ranges are represented and manipulated across ripgrep's crates.

How does the line buffer from the previous lesson integrate with these utilities? Examine how `LineBuffer` provides the byte slices that these functions operate on, completing the picture of ripgrep's text handling pipeline.

# ripgrep crates/searcher/src/searcher/mod.rs: Searcher Core

## What This File Does

This file defines the `Searcher` type, which is the central orchestrator for all search operations in ripgrep's searcher crate. While previous lessons examined the supporting infrastructure—line buffers, line utilities, and the crate's public API—this file brings everything together into a cohesive search engine that can handle files, memory-mapped data, and arbitrary byte streams.

The `Searcher` type embodies a sophisticated strategy pattern, automatically selecting the optimal search approach based on configuration and input characteristics. It can search incrementally through a rolling buffer for memory efficiency, load entire files for multi-line pattern matching, or leverage memory maps for potentially faster access to large files. The file also defines the `SearcherBuilder`, continuing the builder pattern theme we've seen throughout ripgrep's codebase.

## Section 1: The Builder Pattern in Action

The `SearcherBuilder` represents ripgrep's commitment to ergonomic configuration. Rather than requiring users to construct a `Config` struct with dozens of fields, the builder provides a fluent interface where each configuration option is set through a dedicated method that returns `&mut Self`, enabling method chaining.

This pattern serves multiple purposes beyond mere convenience. It allows the library to set sensible defaults internally—the `Config::default()` implementation shows that line numbers are enabled, binary detection is off, and BOM sniffing is on, among other choices. Users only need to specify the options they want to change from these defaults. The builder also provides a natural place for validation logic and for transforming user-facing options into internal representations.

The `build` method demonstrates an important aspect of the builder pattern: the builder's job isn't just to collect configuration, but to construct a fully initialized object ready for use. Notice how it creates the decode builder for transcoding, pre-allocates the decode buffer, and initializes both the line buffer and multi-line buffer. This ensures that the returned `Searcher` is immediately usable without further initialization.

See: Companion Code Section 1

---

## Section 2: Configuration as a First-Class Concept

The `Config` struct centralizes all search parameters into a single, cloneable type. This design separates the concerns of "what options exist" from "how to construct a searcher with those options." The configuration includes line terminators, context settings, heap limits, memory map strategies, binary detection modes, encoding preferences, and match limits.

What makes this configuration design particularly elegant is how it handles interdependencies. The `max_context` method shows that some configuration values derive from others—the maximum context needed is the larger of before or after context. The `line_buffer` method shows how configuration translates into concrete infrastructure, building a `LineBuffer` with appropriate capacity limits and binary detection settings.

The `passthru` option illustrates configuration-level logic that affects multiple settings. When passthru is enabled in the builder, both before and after context are set to zero because passthru mode effectively provides unlimited context. This normalization happens at build time, ensuring the resulting `Config` is internally consistent.

See: Companion Code Section 2

---

## Section 3: Binary Detection Strategies

The `BinaryDetection` type exemplifies ripgrep's approach to handling real-world complexity through well-defined strategies. Binary detection is inherently heuristic—there's no universal definition of what makes data "binary"—so ripgrep provides three clear strategies: none, quit, and convert.

The "none" strategy simply ignores binary data, passing all bytes through unchanged. The "quit" strategy stops searching when a binary byte (typically null) is encountered, treating the file as if it ended at that point. The "convert" strategy replaces binary bytes with line terminators, effectively breaking the file at each binary byte occurrence.

The documentation carefully explains that these strategies behave differently depending on the search mode. When using a rolling buffer, every byte passes through the buffer and can be checked for binary content. When using memory maps or heap-allocated data, only matched

regions and an initial prefix are checked. This reflects the reality that different search strategies have different performance characteristics and capabilities.

See: Companion Code Section 3

---

## Section 4: Encoding and Transcoding Support

The `Encoding` type wraps the `encoding_rs` crate's encoding representation, providing a simple interface for specifying character encodings by label. This design demonstrates good encapsulation—users work with a ripgrep-specific type rather than directly with `encoding_rs`, allowing the internal representation to change without breaking user code.

The encoding system interacts with BOM (byte-order mark) sniffing in interesting ways. When no encoding is explicitly specified, the searcher can detect UTF-8 or UTF-16 encodings by examining the first few bytes of the input. When an encoding is specified, it's used unconditionally unless a BOM is present, in which case the BOM takes precedence. This layered approach provides sensible defaults while allowing explicit control when needed.

The `slice_needs_transcoding` method shows the practical implications: a slice only needs transcoding if an encoding was explicitly set or if BOM sniffing is enabled and a BOM is actually present. This optimization avoids unnecessary transcoding work when searching plain ASCII or UTF-8 data.

See: Companion Code Section 4

---

## Section 5: The Searcher Type and Interior Mutability

The `Searcher` struct uses `RefCell` for several of its buffers, which might seem surprising in a type that requires `&mut self` for search operations. This design enables lending borrows of the `Searcher` to sink implementations while still allowing the searcher to manage its internal buffers.

The key insight is that `RefCell` isn't being used to enable concurrent access—the `&mut self` requirement prevents that. Instead, it's enabling a form of reborrowing where the searcher can pass references to its configuration and other read-only parts to the sink, while still being able to mutably access its buffers through the `RefCell`. The comment in the code explicitly notes that runtime panics are prevented because a mutable borrow is required to execute a search.

The buffers themselves serve distinct purposes: `decode_buffer` provides scratch space for transcoding operations, `line_buffer` holds partial lines during incremental search, and

`multi_line_buffer` stores entire file contents when multi-line search requires it. Pre-allocating these buffers in the `Searcher` allows them to be reused across multiple searches, avoiding repeated allocation overhead.

See: Companion Code Section 5

---

## Section 6: Search Strategy Selection

The `search_path` and `search_file` methods demonstrate strategy selection in action. The searcher first checks if memory mapping is appropriate and available—the `mmap.open()` call returns `Some(mmap)` only if memory mapping is both enabled and succeeds for the given file. If so, the search proceeds via `search_slice` on the mapped memory.

When memory mapping isn't used, the searcher considers whether multi-line search is needed. If so, it takes a special fast path for files: it pre-allocates a buffer based on the file's metadata and reads the entire file into it. This is more efficient than the generic reader path because files have known sizes, enabling accurate pre-allocation.

The fallback path uses `search_reader`, which handles arbitrary `io::Read` implementations. This tiered approach—memory map first, then file-specific optimization, then generic reader—ensures that the most efficient strategy is used for each input type while maintaining a single public API.

See: Companion Code Section 6

---

## Section 7: The Generic Reader Search Path

The `search_reader` method handles the most general case: searching any type that implements `io::Read`. This method must handle transcoding, multi-line vs. line-by-line search, and proper error propagation, all while working with streaming input that might not fit in memory.

The method first checks configuration consistency via `check_config`, ensuring that the searcher and matcher agree on line terminators and that the configuration allows some search strategy to proceed. It then sets up transcoding by borrowing the decode buffer and building a decoder around the input reader.

For multi-line search, the entire input must be read into the multi-line buffer before searching can begin—there's no way to search for patterns that span lines without having all the lines available. For line-by-line search, the method creates a `LineBufferReader` that wraps the decoded input and passes it to `ReadByLine`, which handles the incremental search logic.

See: Companion Code Section 7

---

## Section 8: Slice Search and Fast Paths

The `search_slice` method searches in-memory byte slices, which provides opportunities for optimization that aren't available with streaming input. When the slice doesn't need transcoding, the searcher can work directly with the bytes without any copying or buffering overhead.

The transcoding check is particularly efficient: if no explicit encoding is set and either BOM sniffing is disabled or no BOM is present, the slice is searched directly. Only when transcoding is genuinely needed does the method fall back to `search_reader`, treating the slice as a reader.

For slices that don't need transcoding, the choice between `MultiLine` and `SliceByLine` depends on whether multi-line matching is needed. Both strategies operate on the slice without additional buffering, but they differ in how they iterate over and report matches.

See: Companion Code Section 8

---

## Section 9: Configuration Accessors and Multi-Line Detection

The `Searcher` exposes numerous accessor methods that allow `Sink` implementations to query the searcher's configuration. These methods enable sinks to adapt their behavior—for example, a sink might format output differently when line numbers are disabled, or might handle context lines specially when passthru mode is enabled.

The `multi_line_with_matcher` method deserves special attention because it demonstrates how the searcher makes runtime decisions based on matcher capabilities. Even if multi-line search is configured, the searcher can avoid the expensive multi-line strategy if the matcher guarantees it will never produce multi-line matches. This happens when the matcher's line terminator matches the searcher's, or when the matcher specifies that certain bytes can never appear in a match.

This optimization is subtle but important: a matcher for a pattern like `foo.*bar` with no newlines in the pattern can still match across lines, but a matcher for `foo` never will. By querying the matcher's capabilities, the searcher can use the more efficient line-by-line strategy even when multi-line mode is nominally enabled.

See: Companion Code Section 9

## Section 10: Buffer Filling for Multi-Line Search

The `fill_multi_line_buffer_from_file` and `fill_multi_line_buffer_from_reader` methods handle the task of loading entire inputs into memory for multi-line search. These methods must balance efficiency with safety, respecting heap limits while avoiding unnecessary allocations.

For files without heap limits, the code uses file metadata to pre-allocate a buffer of the exact size needed, then reads the entire file in one operation. This is optimal when memory is not constrained. When a heap limit is set, the code must read incrementally, growing the buffer as needed while ensuring it never exceeds the limit.

The manual buffer growth logic in `fill_multi_line_buffer_from_reader` is notable: it starts at the smaller of the default capacity or heap limit, then doubles the buffer size up to the limit on each expansion. This geometric growth strategy, common in dynamic arrays, provides amortized O(1) insertion cost while respecting memory constraints. The code also handles interrupted reads correctly, retrying on `ErrorKind::Interrupted`.

See: Companion Code Section 10

---

## Section 11: Configuration Validation and Error Handling

The `ConfigError` enum captures the ways that searcher configuration can be invalid. The `SearchUnavailable` variant handles the case where heap limit is zero and memory mapping is disabled—with no way to buffer data, searching is impossible. The `MismatchedLineTerminators` variant catches the subtle bug of having a matcher and searcher disagree on what constitutes a line boundary.

The `check_config` method is called at the start of each search operation, not when building the searcher. This timing is intentional: some configuration errors depend on the matcher being used, which isn't known at build time. The method validates the heap limit and memory map configuration, then checks line terminator consistency if the matcher specifies a line terminator.

The error type implements both `std::error::Error` and `Display`, providing clear error messages that include the relevant configuration values. This makes debugging configuration issues straightforward, as users see exactly what the conflict is.

See: Companion Code Section 11

---

## Section 12: BOM Detection and the Testing Strategy

The `slice_has_bom` function checks whether a byte slice begins with a UTF-8 or UTF-16 byte-order mark. This is used to determine whether transcoding is needed when searching slices. The function uses `encoding_rs::Encoding::for_bom` to detect and identify the BOM, then checks if it's one of the three encodings that require special handling.

The test module demonstrates ripgrep's testing philosophy for configuration edge cases. The tests verify that specific configuration errors are detected: a zero heap limit without memory maps enabled causes `search_slice` to fail, and mismatched line terminators between matcher and searcher produce an error. The UTF-8 BOM sniffing test verifies a specific bug fix, showing that BOMs are properly stripped before searching.

These tests operate at the integration level, exercising the full search path rather than testing individual functions in isolation. This approach catches interaction bugs that unit tests might miss while keeping tests focused on observable behavior.

See: Companion Code Section 12

---

## Key Takeaways

1. The `SearcherBuilder` exemplifies the builder pattern's strengths: sensible defaults, fluent configuration, validation at build time, and full initialization of the resulting object.
2. The `Searcher` uses `RefCell` not for concurrent access but to enable borrowing patterns that would otherwise require splitting the type, while `&mut self` on search methods prevents runtime panics.
3. Strategy selection happens at multiple levels: memory maps vs. file reads vs. generic readers, and multi-line vs. line-by-line, with the searcher automatically choosing the most efficient option.
4. Binary detection and encoding support demonstrate how to handle real-world messiness through well-defined strategies that users can select based on their needs.
5. Configuration validation is deferred until search time because some errors depend on the matcher, which isn't available when building the searcher.
6. The `multi_line_with_matcher` method shows how capability queries enable optimizations—the searcher can avoid expensive strategies when the matcher guarantees they aren't needed.

## What to Read Next

How does the multi-line search strategy actually work? Read `searcher/glue.rs` to see how `MultiLine` implements searching over complete data.

What about the line-by-line strategies? The `ReadByLine` and `SliceByLine` types in `searcher/glue.rs` show how incremental search operates.

How does the core matching loop function? Read `searcher/core.rs` to understand the low-level iteration over matches and context.

How do sinks receive and process search results? Read `sink.rs` to understand the `Sink` trait that all result consumers implement.

# ripgrep crates/searcher/src/searcher/core.rs: Search Algorithm

## What This File Does

This file implements the heart of ripgrep's search execution—the `Core` struct that orchestrates how patterns are matched against file content. While the `Searcher` we examined in the previous lesson handles high-level coordination and configuration, `Core` does the actual work of iterating through lines, tracking match positions, managing context windows, and delivering results to sinks.

The design reflects a tension that pervades all search tools: the need for speed versus the need for features. A simple "find all matches" algorithm is straightforward, but users expect line numbers, context lines before and after matches, inverted matching, passthrough mode, and more. This file shows how ripgrep maintains performance while supporting this rich feature set through a two-path algorithm—a "fast" path for simple cases and a "slow" path that handles complex scenarios gracefully.

---

## Section 1: The Core State Machine

The `Core` struct is a state machine that tracks everything needed to process a search from beginning to end. Understanding its fields reveals how ripgrep thinks about search as a problem of maintaining precise position information while streaming through potentially enormous files.

The struct carries references to configuration, the matcher, the searcher, and the sink—but it owns substantial state of its own. Position tracking happens at multiple granularities: `pos` tracks where we are in the current buffer, while `absolute_byte_offset` tracks where that buffer position maps to in the overall file. This dual-tracking is essential because buffers get rolled (partially consumed and refilled), so local positions must be translatable to file-global positions for accurate reporting.

Line number tracking reveals an optimization insight. Rather than counting lines from the beginning of the file for every match, `last_line_counted` remembers how far we've counted. When we need the line number for a match at position 1000, we only count lines from our last checkpoint, not from zero. This matters enormously for files with millions of lines.

See: Companion Code Section 1

## Section 2: The Fast Path vs. Slow Path Decision

One of the most important architectural decisions in this file is the split between "fast" and "slow" line-by-line matching. The choice happens at runtime based on the matcher's capabilities, and it has significant performance implications.

The `is_line_by_line_fast` method determines which path to take by examining the matcher's properties. The fast path requires that the regex engine can handle line terminators correctly on its own—specifically, it must be able to find candidate matches without the searcher having to strip line terminators first. This is possible when the matcher advertises through `line_terminator()` or `non_matching_bytes()` that it understands where lines end.

The slow path handles everything else. It iterates line by line, explicitly strips the line terminator from each line before matching, and handles the full complexity of context, passthrough mode, and inverted matching. The fast path achieves speed by letting the regex engine scan ahead to find matches, then working backward to determine line boundaries—but this optimization requires the regex and searcher to agree on what constitutes a line ending.

There's a fascinating workaround documented in the code: when the line terminator is NUL (`\x00`), the fast path is disabled even if the matcher claims to support it. This addresses a bug in the regex library where line anchors don't work correctly with non-newline terminators. This kind of defensive coding is common in performance-critical systems where theoretical capabilities don't always match practical behavior.

See: Companion Code Section 2

## Section 3: The Fast Line Search Algorithm

The fast path in `match_by_line_fast` uses a fundamentally different approach than traditional line-by-line iteration. Instead of checking every line for a match, it asks the regex engine to find "candidate" positions where matches might exist, then verifies those candidates.

The matcher's `find_candidate_line` method returns a `LineMatchKind` that distinguishes between confirmed matches and candidates. A confirmed match means the regex engine is certain the position contains a match—no further verification needed. A candidate means the regex engine found something promising but can't guarantee it's actually a match (perhaps due to line

terminator handling or other edge cases). For candidates, the searcher must extract the full line and verify with `is_match`.

This candidate-based approach enables significant optimization. Consider searching for "error" in a log file: the regex engine can use its internal optimizations (like literal scanning or SIMD) to quickly find positions containing "error", then the searcher only processes those specific lines. Compared to the slow path that must touch every line, this can be dramatically faster for selective patterns.

The `find_by_line_fast` method implements this loop. It maintains a position, asks for candidates, locates the containing line when one is found, and either returns the line (for confirmed matches) or verifies and advances (for candidates). The loop continues until the buffer is exhausted or a match limit is reached.

See: Companion Code Section 3

---

## Section 4: Inverted Match Handling

Inverted matching (showing lines that don't match) transforms the search problem in a subtle way. Instead of finding matching lines, we need to find the regions between matches. The `match_by_line_fast_invert` method shows how this inversion works in the fast path.

When we find a match, the "inverted match" is everything from our current position up to (but not including) the matching line. If we reach the end of the buffer without finding a match, the inverted match is everything from our current position to the end. This transforms the search from "find matches" to "find boundaries between matches."

Each inverted region must be processed line by line to report individual non-matching lines. This creates an interesting situation: even in the "fast" path, inverted matching requires iteration over the non-matching region. The speedup comes from quickly identifying the boundaries of non-matching regions, not from avoiding iteration entirely.

The interaction with match limits adds complexity. When inverted matching is active, each line in a non-matching region counts as a match (for purposes of the limit), so the loop must check `has_exceeded_match_limit` after each line is sunk. This ensures that `--max-count` works correctly even when showing non-matching lines.

See: Companion Code Section 4

---

## Section 5: The Slow Path for Complex Scenarios

The slow path in `match_by_line_slow` handles all the cases the fast path can't: passthrough mode, stop-on-nonmatch, matchers without proper line terminator support, and other edge cases. It's "slow" not because it's poorly implemented, but because it makes fewer assumptions and thus can't skip as much work.

The algorithm uses `LineStep`, an iterator from the lines module, to walk through lines in order. For each line, it strips the line terminator and asks the matcher for a shortest match (an optimization over finding the full match when we only need to know if a match exists). The result is XORed with `invert_match` to determine success.

The control flow handles four distinct outcomes for each line. If the line matches (accounting for inversion), we output before-context, then the match itself. If the line doesn't match but we're still emitting after-context from a previous match, we output it as context. If passthrough mode is enabled and none of the above apply, we output the line as "other" context. Finally, if stop-on-nonmatch is set and we've already had a match, a non-match ends the search.

This state machine approach ensures all features compose correctly. Passthrough mode with before-context and after-context and inverted matching all work together because each line is classified and routed through the appropriate handler.

See: Companion Code Section 5

---

## Section 6: Context Management

Context handling—showing lines before and after matches—is one of the trickier aspects of search. The `before_context_by_line` and `after_context_by_line` methods implement this, and their interaction with the position tracking reveals careful design.

Before-context is retrospective: when we find a match, we need lines we've already passed. The `last_line_visited` field tracks how far we've sent to the sink, allowing us to look back. The `before_context_by_line` method uses `lines::preceding` to find the start of the context region, then iterates forward through those lines, sending each as context.

After-context is prospective: after a match, we want subsequent lines even if they don't match. The `after_context_left` counter starts at `config.after_context` when a match is sunk and decrements with each context line. But there's a subtlety: if we encounter a match while outputting after-context (and we've exceeded the match limit), we still output it and reset the after-context counter. This ensures that clustered matches don't cause context truncation.

The break context mechanism handles the `--` separators between match groups. When there's a gap between `last_line_visited` and the start of new context, and we've already output something, the sink receives a break notification. This simple check enables the familiar grouped output format while keeping the core algorithm clean.

See: Companion Code Section 6

---

## Section 7: Binary Detection During Search

Binary detection in `detect_binary` shows how ripgrep handles non-text files during search, not just before it begins. The method scans the current range for a binary marker byte and, if found, notifies the sink and potentially aborts the search.

The detection checks `binary_byte_offset` first to avoid redundant scanning—once we've found binary data, we've found it. The configuration determines whether binary data means "quit immediately" or "convert and continue." When quitting, the search stops at the first binary byte. When converting, the search continues but the sink is informed, allowing output formatting to adjust.

The integration point is notable: binary detection happens in `sink_matched` and all context-sinking methods, checking each line before it's output. This catches binary data that appears mid-file, not just at the beginning. The early-return pattern—check, detect, potentially abort—appears repeatedly throughout the file, showing how cross-cutting concerns like binary handling weave through the core algorithm.

See: Companion Code Section 7

---

## Section 8: The Sink Protocol

The sink methods (`sink_matched`, `sink_before_context`, `sink_after_context`, `sink_other_context`, `sink_break_context`) implement the protocol between the search core and its output handler. Each follows a similar pattern but with context-specific details.

The `sink_matched` method is the most complex. It performs binary detection, handles the break context (for output grouping), counts lines up to the match position, calculates the absolute offset, constructs a `SinkMatch` structure with all the metadata a sink might need, and invokes the sink. The return value propagates back—if the sink returns false (perhaps due to output errors or user interruption), the search stops.

The `SinkMatch` structure passed to the sink reveals what metadata ripgrep considers essential: the line terminator (for proper display), the matched bytes, the absolute byte offset (for position reporting), the line number (if tracked), the full buffer (for extended context or highlighting), and the range of the match within that buffer. This rich context enables sophisticated output formatting without the sink needing to understand search internals.

Context sinking methods are simpler but follow the same pattern. They all update `last_line_visited` and set `has_sunk` to true, maintaining the invariants the main loop depends on. The `after_context` variant additionally decrements the counter, while `other_context` is used for passthrough mode where lines are neither matches nor traditional context.

See: Companion Code Section 8

---

## Section 9: Buffer Rolling and Position Maintenance

The `roll` method handles the transition between buffers during streaming search. When a buffer is consumed, the search needs to "roll" forward, keeping enough context to handle before-context requirements while discarding what's no longer needed.

The calculation of how much to consume balances efficiency against correctness. With no context requirements, we consume everything. With before-context, we must keep `before_context` lines at the end of the buffer so they're available if the next match appears at the start of the new data. The `lines::preceding` function finds this boundary efficiently.

After calculating the consumed amount, several state variables update. The absolute byte offset advances by the consumed amount. The line count variables reset to zero (they're relative to the buffer, which has changed). The position moves to point at the retained portion at the end of the buffer. This coordinated update ensures all the position-tracking invariants remain valid after the roll.

The `last_line_visited` handling adds complexity. We take the maximum of the context start and `last_line_visited` because we might have visited lines beyond what context requires. This prevents losing track of where we are when matches are sparse and large amounts of buffer would otherwise be consumed.

See: Companion Code Section 9

---

## Section 10: Match Limits and Early Termination

The `has_exceeded_match_limit` check appears throughout the code, guarding against continued work after reaching `--max-count`. But the integration is subtle: exceeding the limit doesn't immediately stop the search. Instead, it gates specific operations while allowing others to complete.

When the limit is exceeded in the slow path, the search continues only if passthrough mode is active or after-context is pending. This ensures clean output: you don't get a match followed by truncated context. In the fast path, exceeding the limit causes early return only after after-context is processed.

The match counting itself happens in `increment_count`, called when a match is confirmed. The counter lives in `Core` rather than configuration because it's mutable state that changes during search. The `count` method provides read access for limit checking without exposing the field directly.

This design enables the search to behave correctly even near limits. A search with `--max-count=1 --after-context=3` will show the one match plus its three context lines, then stop. The checks are positioned so that after-context can complete before the limit truly halts execution.

See: Companion Code Section 10

## Section 11: Builder Pattern Integration

Looking at `Core::new`, we see how this type fits into ripgrep's builder pattern ecosystem. The `Core` is created from a `Searcher` (which was created by a builder), a `Matcher` (also builder-created), a `Sink` (typically builder-created), and a binary flag determined by the search context.

The constructor extracts configuration from the searcher and sets up all tracking state to initial values. Line numbering is conditionally initialized based on configuration—if we're not tracking line numbers, the `Option` stays `None` throughout, avoiding counting work entirely. The logging call at the end helps debugging by announcing which search path will be used.

This type doesn't have its own builder because it's not meant for public construction. It's a private implementation detail (`pub(crate)`) created by the `Searcher` during execution. The builder pattern stops at the public API boundary; internal types are constructed directly with all parameters required.

The separation between public configuration (in `Searcher` and its builder) and internal execution state (in `Core`) is a common pattern in Rust. It allows the public API to be stable and ergonomic while giving internal code flexibility to reorganize state as needed for performance or correctness.

See: Companion Code Section 11

---

## Key Takeaways

First, the fast path versus slow path split demonstrates how search tools achieve performance: by identifying cases where assumptions enable optimizations, and gracefully falling back when those assumptions don't hold.

Second, state machine design with explicit tracking variables (`pos`, `last_line_visited`, `after_context_left`) enables complex features to compose correctly. Each piece of state has a clear purpose and well-defined update points.

Third, the candidate-based matching approach shows how cooperating components (searcher and regex engine) can achieve better performance than either could alone. The matcher provides hints, the searcher verifies them.

Fourth, context management illustrates the complexity hidden behind simple-seeming features. Before-context requires looking backward, after-context requires counting forward, and both must handle edge cases at buffer boundaries.

Fifth, binary detection woven throughout the sink methods shows how cross-cutting concerns integrate without dominating the main algorithm. Each sink method checks, but the check is a single line.

Sixth, the careful positioning of match limit checks ensures clean output termination. The limit gates new work without truncating in-progress output.

---

## What to Read Next

How does the multi-line searcher differ from this line-by-line approach? The `mmap.rs` and `glue.rs` files in this directory show the alternative execution paths.

What happens to matched lines after they leave the sink? The `printer/standard.rs` file shows how sunk matches become terminal output.

How does the matcher produce those candidate lines? The `grep-matcher` crate defines the trait that regex engines implement.

How does buffer rolling work at the lower level? Return to `line_buffer.rs` with fresh eyes now that you understand how `Core::roll` uses it.

# ripgrep crates/searcher/src/searcher/mmap.rs: Memory Mapping

## What This File Does

This file provides ripgrep's abstraction over memory-mapped file I/O, encapsulating the decision of when and how to use memory maps for reading file contents. Memory mapping is a technique where a file's contents are mapped directly into a process's address space, allowing the operating system to handle paging and caching rather than requiring explicit read system calls.

The design here is notable for two reasons: first, it treats memory map usage as a *policy choice* rather than an implementation detail, and second, it surfaces the inherent unsafety of memory maps through Rust's type system. This file demonstrates how ripgrep balances performance optimization opportunities against safety guarantees, giving callers explicit control over the trade-off.

## Section 1: The Memory Mapping Trade-Off

Memory-mapped I/O can be significantly faster than traditional read-based I/O for certain workloads. When you memory-map a file, you're essentially telling the operating system to make the file's bytes appear at a particular address in your process's memory. Subsequent reads from that memory region trigger page faults that the OS handles by loading data from disk into RAM. This approach can avoid copying data between kernel and user space, and it lets the OS's virtual memory system handle caching intelligently.

However, memory maps come with a fundamental problem that's difficult to solve in a memory-safe language: what happens if the underlying file changes while you're reading from the mapped region? If another process truncates or modifies the file, your mapped memory region might suddenly refer to invalid data, or worse, to data that no longer exists. On Unix systems, this typically results in a `SIGBUS` signal—an unrecoverable error that will crash your program.

This is why ripgrep doesn't just silently use memory maps when available. The decision to use them is an *assertion* by the caller that they're willing to accept this risk. For a command-line tool like ripgrep, this trade-off often makes sense: the performance benefit can be substantial, and the risk of file modification during a brief search operation is low enough to accept.

See: Companion Code Section 1

---

## Section 2: The Newtype Pattern for Configuration

The `MmapChoice` type follows a pattern we've seen throughout ripgrep's codebase: wrapping an internal enum in a public struct. This isn't just stylistic—it serves several important purposes in Rust library design.

By making `MmapChoice` a struct that contains an `MmapChoiceImpl` enum, the library reserves the right to change the internal representation without breaking API compatibility. If the team later decided to add fields (perhaps for platform-specific heuristics or file size thresholds), they could do so without affecting any code that uses `MmapChoice`. The inner enum is private to the module, so external code can't match on it or construct it directly.

This pattern also enables what I'd call "controlled construction." Notice that there's no way to create an `MmapChoice::Auto` variant without going through the `unsafe fn auto()` constructor. The enum variants themselves aren't `pub`, so the only way to get an `MmapChoice` is through the provided constructor functions. This is a powerful technique for enforcing invariants at the type level.

See: Companion Code Section 2

---

## Section 3: Safety Contracts and Unsafe Constructors

The `auto()` function is marked `unsafe`, which might seem surprising at first. After all, the function itself just creates a simple enum variant—there's no pointer manipulation or FFI involved. But Rust's `unsafe` keyword isn't just about what a function *does*; it's about what *contract* the caller must uphold.

The documentation for `auto()` explains that calling this function represents an assertion by the caller that "the underlying file won't be mutated" during the memory-mapped read. This isn't something the compiler can verify—it depends on the runtime environment and potentially on other processes. By marking the constructor as `unsafe`, the library makes this responsibility explicit.

This is an example of what's sometimes called "semantic unsafety." The memory safety violation doesn't happen inside `auto()`—it might happen much later, when the memory map is actually accessed. But by requiring `unsafe` at the point where the choice is made, the library ensures that

whoever enables memory mapping has acknowledged the risk. Every `unsafe` block in the calling code serves as documentation that someone made a deliberate decision about safety.

See: Companion Code Section 3

---

## Section 4: The Safe Default

The `Default` implementation returns `MmapChoice::Never`, meaning memory maps are disabled unless explicitly enabled. This follows a principle common in security-sensitive and systems programming: safe defaults with explicit opt-in for risky features.

This design decision reflects ripgrep's philosophy of being safe by default while still exposing advanced features for users who want them. The command-line tool can choose to enable memory maps (and does, in some configurations), but libraries built on top of the searcher crate get the safe behavior automatically.

The `never()` constructor provides a named alternative to `Default::default()`. Having both allows code to be more expressive about intent—calling `MmapChoice::never()` makes it clear that the author actively chose to disable memory maps, while `Default::default()` might just mean "I don't care, give me the default."

See: Companion Code Section 4

---

## Section 5: The Open Method and Graceful Degradation

The `open` method is where the actual memory mapping happens, and it demonstrates several important patterns for robust systems programming. First, notice it returns `Option<Mmap>` rather than `Result<Mmap, Error>`. This is intentional: a failure to create a memory map isn't necessarily an error that the caller needs to handle specially. The searcher can always fall back to regular read-based I/O.

The method implements multiple layers of checking before attempting the memory map. First, it checks whether memory maps are enabled at all via `is_enabled()`. Then it applies platform-specific heuristics—in this case, unconditionally disabling memory maps on macOS with a candid comment acknowledging this deserves re-evaluation.

When the actual memory map creation fails, the method logs the error at debug level rather than propagating it. This is exactly the right approach for an optimization that has a reliable fallback:

you want developers to be able to see what happened if they're debugging performance, but you don't want to burden every caller with handling errors for an optional optimization.

See: Companion Code Section 5

---

## Section 6: Propagating Safety Assertions

Inside `open`, there's an `unsafe` block around the `Mmap::map` call. The accompanying safety comment explains the reasoning: since the only way to have `MmapChoiceImpl::Auto` is through the unsafe `auto()` constructor, reaching this code means the caller already asserted that memory mapping is safe for their use case.

This is a pattern worth understanding deeply. The unsafety "flows" through the type system: someone called `unsafe { MmapChoice::auto() }`, and that single unsafe block justifies all subsequent safe usage of that `MmapChoice` value. The `open` method doesn't require an `unsafe` block to call because the contract was already established at construction time.

This design localizes the unsafe decision to the point where the caller actually has the information needed to make that decision. By the time `open` is called, deep in the search logic, there's no meaningful safety decision left to make—the file handle came from somewhere, and if memory mapping is enabled, we proceed with the operation.

See: Companion Code Section 6

---

## Section 7: Platform-Specific Behavior with cfg

The macOS check uses `cfg!(target_os = "macos")`, which is a compile-time conditional that evaluates to a boolean. Unlike `#[cfg(...)]` attributes that conditionally include code, `cfg!()` includes both branches but lets one become dead code that the compiler eliminates.

The choice to use `cfg!()` here rather than `#[cfg()]` is subtle but reasonable. With `cfg!()`, you get the same compiled behavior (the disabled branch is optimized away), but the code is more readable—you can see all the logic in one place rather than having alternate implementations. The trade-off is that both branches must type-check on all platforms, but since the disabled branch just returns `None`, that's not a concern.

The comment about macOS memory map performance is characteristically honest. Rather than pretending to have precise knowledge, it acknowledges uncertainty while still making a reasonable

default choice. This kind of pragmatic documentation is valuable—it tells future maintainers that this decision should be revisited rather than treated as fundamental.

See: Companion Code Section 7

---

## Section 8: The Builder Pattern Connection

This file connects to a broader pattern we've seen across ripgrep's codebase: configuration types that are built up separately from the components they configure. `MmapChoice` is designed to be a field in a larger configuration struct (the `SearcherBuilder` we saw in the core module), not to be used in isolation.

The pattern enables what you might call "configuration composition." A builder can accept an `MmapChoice` without knowing anything about how it was constructed or what safety assertions were made. The builder just stores it and passes it along when needed. This separation means the safety decision (calling `unsafe { MmapChoice::auto() }`) can happen at the application's entry point, far from the search implementation that actually uses memory maps.

This architectural decision is important for library design. It means that a GUI application, a CLI tool, and an embedded use case can all make different memory mapping decisions without any changes to the search implementation. The configuration flows through the system as data, carrying with it the safety guarantees established at construction.

See: Companion Code Section 8

---

## Key Takeaways

1. Memory-mapped I/O trades safety for performance by letting the OS handle file access, but exposes the program to unrecoverable errors if the underlying file changes during access.
2. Rust's `unsafe` keyword can mark constructors that establish contracts the compiler can't verify, making the caller explicitly acknowledge risks that might manifest much later in the program's execution.
3. The newtype pattern—wrapping an enum in a struct—enables controlled construction, where the only way to create certain states is through specific functions that can enforce invariants or require `unsafe`.
4. Graceful degradation through `Option` return types allows optimizations to fail silently when there's a reliable fallback, with debug logging for developers who need to understand what

happened.

5. Platform-specific behavior can be implemented cleanly with `cfg!()` macros that keep conditional logic together while still generating efficient platform-specific code.
6. Separating configuration types from implementation allows safety decisions to be made at application entry points and carried through the system as data.

---

## What to Read Next

How does the searcher decide between memory-mapped and read-based I/O at runtime? Return to `searcher/core.rs` to see how `MmapChoice::open` integrates with the main search loop.

What other configuration options does the searcher support? Look at `searcher/mod.rs` to see how `MmapChoice` fits into the broader `SearcherBuilder` pattern.

How does ripgrep's command-line interface decide whether to enable memory maps? The top-level application code makes the `unsafe { MmapChoice::auto() }` call based on user flags and platform detection.

# ripgrep crates/printer/src/lib.rs: Printer Architecture

## What This File Does

This file serves as the public interface and organizational hub for ripgrep's printer subsystem. While the file itself is relatively small, it defines the architectural boundaries of how search results flow from the searcher to human or machine-readable output. The printer crate sits at the final stage of ripgrep's data pipeline, transforming raw match data into formatted strings that users actually see.

The architecture follows a familiar Rust pattern: a `lib.rs` file that re-exports types from private submodules, carefully curating what becomes public API. This design allows the implementation to evolve internally while maintaining a stable external contract. The file also reveals important design decisions through its documentation, feature flags, and the types it chooses to expose.

---

## Section 1: The Public API Surface

The file opens with a module-level documentation comment that serves dual purposes: it explains the crate to documentation readers and provides a working example. This pattern is valuable because Rust's documentation tests actually compile and run, ensuring the example stays accurate as the code evolves.

Notice how the documentation describes three distinct printer types: `Standard`, `JSON`, and `Summary`. This isn't arbitrary—it reflects a fundamental design choice about output formats. Rather than creating a single configurable printer, ripgrep separates concerns into specialized types. Each printer type has a distinct purpose and audience: `Standard` for humans reading terminal output, `JSON` for programs consuming structured data, and `Summary` for aggregate reporting.

The example demonstrates the integration pattern between the searcher and printer crates. The `printer.sink(&matcher)` call is particularly important—it shows how printers create sink adapters that implement the traits the searcher expects. This inversion of control means the searcher doesn't need to know anything about output formatting; it just writes to a sink.

See: Companion Code Section 1

## Section 2: The Re-export Strategy

The `pub use` statements reveal the intentional public API. Everything listed here is what users of the crate can directly import and use. Notice the grouping: color-related types come first, then hyperlinks, paths, the standard printer, statistics, and summary printing.

This re-export pattern exemplifies Rust's approach to API design. The actual implementations live in private modules (`color`, `hyperlink`, `standard`, etc.), but users import from the crate root. This gives maintainers freedom to restructure internals without breaking downstream code. If `ColorSpecs` moves from `color.rs` to `color/specs.rs`, users importing `grep_printer::ColorSpecs` notice nothing.

The naming conventions tell a story about the Builder pattern ecosystem you've seen throughout ripgrep. For almost every major type, there's a corresponding builder: `Standard` has `StandardBuilder`, `Summary` has `SummaryBuilder`, `PathPrinter` has `PathPrinterBuilder`. This consistency across the crate (and indeed across all of ripgrep's crates) creates a predictable API that developers can learn once and apply everywhere.

See: Companion Code Section 2

---

## Section 3: Conditional Compilation for Optional Features

The JSON printer gets special treatment through conditional compilation. The `#[cfg(feature = "serde")]` attribute means these types only exist when the `serde` feature is enabled. This is a deliberate dependency management strategy—users who only need text output don't pay for JSON serialization in their binary size or compilation time.

The corresponding `cfg_attr` on the documentation ensures that docs.rs (Rust's documentation hosting service) properly marks these items as feature-gated. Users browsing documentation can immediately see that `JSON`, `JSONBuilder`, and `JSONSink` require opt-in.

This pattern appears throughout the Rust ecosystem but requires careful thought. The types re-exported under the feature gate must have their implementing modules similarly gated, which you can see in the module declarations later in the file. Both `json` and `jsont` modules are conditionally compiled.

See: Companion Code Section 3

---

## Section 4: The Look-Ahead Kludge

The `MAX_LOOK_AHEAD` constant with its extensive comment reveals something important about real-world software engineering: sometimes you need pragmatic workarounds. The comment admits this is "an unfortunate kludge" while explaining exactly why it exists.

PCRE2, one of the regex backends ripgrep can use, supports look-ahead assertions—patterns that match based on what follows without consuming those characters. When searching line-by-line, the searcher provides chunks of input, but look-ahead needs to see beyond the current chunk's boundary. The ideal solution would involve refactoring multiple crate interfaces, but that's a "large change" with its own risks.

Instead, the pragmatic solution adds 128 extra bytes of context for PCRE2 to examine. The comment notes this only affects multi-line mode, limiting the impact. This is instructive for any large codebase: document your workarounds, explain the ideal solution, and make clear where the compromise lies. Future maintainers (or your future self) will thank you.

See: Companion Code Section 4

---

## Section 5: Module Organization and Privacy

The module declarations at the bottom of the file show the internal structure of the crate. The `#[macro_use]` attribute on `macros` is notable—it makes macros defined in that module available throughout the crate without explicit imports. This is an older Rust pattern; modern code often uses explicit `crate::macros::macro_name!()` calls, but `macro_use` remains common for internal utilities.

The module list itself maps to the functionality exposed in the public API: `color` for terminal coloring, `counter` presumably for counting matches, `hyperlink` for clickable terminal links, and so on. Some modules like `counter` and `util` aren't represented in the public exports at all—they're pure implementation details.

The separation between `json` and `jsont` is interesting. Based on naming conventions, `jsont` likely contains the JSON type definitions (schemas for the output format), while `json` contains the printer implementation. This separation allows the type definitions to be shared or tested independently.

See: Companion Code Section 5

---

## Section 6: The Sink Pattern and Trait Integration

The documentation example shows `printer.sink(&matcher)`, which hints at a crucial architectural pattern. The printer doesn't implement the searcher's sink trait directly. Instead, it produces a sink wrapper that does. This allows a single printer to be reused across multiple searches while producing properly separated output.

This design connects directly to what you learned in the Searcher Core lesson. The `Searcher` accepts anything implementing `Sink`, and the printer crates provide `Sink` implementations. The `Standard` type produces `StandardSink`, `Summary` produces `SummarySink`, and `JSON` produces `JSONSink`.

The `&matcher` parameter reveals that sinks need access to the matcher for operations like replacement. When you call `search_slice`, matches flow through the sink, which uses the matcher to reconstruct what was matched and potentially compute replacements. The printer formats this information according to its configuration.

See: Companion Code Section 6

---

## Section 7: The Builder Pattern Ecosystem Continued

This file is a waypoint in understanding ripgrep's pervasive use of the Builder pattern. Having studied `searcher/mod.rs` in earlier lessons, you've seen how `SearcherBuilder` configures search behavior. The printer crate follows identical conventions.

`StandardBuilder`, `SummaryBuilder`, and `JSONBuilder` all let you configure output before creating the printer. This might include settings like whether to show line numbers, how many context lines to include, color configuration, and hyperlink formatting. The builder accumulates these settings, then the `build()` method (or similar) produces an immutable printer.

This consistency isn't accidental—it's architectural discipline. When every major component uses builders, users develop intuition: "I need to configure X, so I probably want XBuilder." The pattern also enables future extensibility. Adding a new option means adding a builder method, not changing constructors throughout the codebase.

See: Companion Code Section 7

## Section 8: Terminal Coloring Strategy

The re-exported `ColorSpecs`, `UserColorSpec`, `ColorError`, and `default_color_specs` types reveal that ripgrep takes color seriously. Unlike tools that hardcode ANSI escape sequences, ripgrep allows users to customize colors and handles different terminal capabilities.

`UserColorSpec` suggests user-provided color specifications, probably from command-line arguments or configuration files. `ColorSpecs` is likely the internal representation after parsing and validation. The presence of `ColorError` indicates that color specifications can be invalid, and the error type is part of the public API so callers can handle these gracefully.

The `default_color_specs` function provides sensible defaults, but the architecture allows overrides at every level. This reflects ripgrep's philosophy of being powerful out of the box while remaining fully customizable. The printer doesn't assume a terminal type—it's configured with color information and produces output accordingly.

See: Companion Code Section 8

---

## Section 9: Hyperlink Infrastructure

The hyperlink types represent relatively modern terminal functionality. Some terminals support OSC 8 escape sequences that make text clickable. When you see a file path in ripgrep's output in a supported terminal, clicking it can open that file in your editor.

The exported types reveal the configuration surface: `HyperlinkConfig` for enabling the feature, `HyperlinkFormat` for the URL template, `HyperlinkEnvironment` for accessing system information, and `HyperlinkAlias` for named format shortcuts. The presence of `HyperlinkFormatError` indicates that format strings are validated—users can specify invalid templates and get helpful error messages.

The `hyperlink_aliases` function likely provides preset formats for common scenarios. This feature demonstrates ripgrep's attention to modern terminal capabilities while maintaining backward compatibility. Terminals that don't understand OSC 8 sequences simply don't render them as links, and the configuration system lets users opt out entirely.

See: Companion Code Section 9

## Section 10: Statistics and Summary Reporting

The `Stats` type and the `Summary` printer serve complementary purposes. `Stats` collects numeric data about searches: how many matches, how many files searched, how many bytes processed. `Summary` formats this data (and similar information) for display.

`SummaryKind` suggests different summary formats—perhaps counts only, files with matches, files without matches, or quiet mode that only sets exit codes. This flexibility lets ripgrep integrate into scripts that need different information levels.

The separation between data collection (`Stats`) and presentation (`Summary`) follows good design principles. The searcher or other components update stats during execution, and the summary printer reads those stats when formatting output. Neither component knows too much about the other's internals.

See: Companion Code Section 10

---

## Key Takeaways

1. A `lib.rs` file defines the public API surface of a crate through selective re-exports, allowing internal restructuring without breaking downstream users.
2. Separating printer types by output format (Standard, JSON, Summary) rather than creating one configurable type leads to cleaner implementations where each type optimizes for its specific use case.
3. Feature-gated re-exports with `#[cfg(feature = "...")]` let crates offer optional functionality without imposing dependencies on users who don't need it.
4. Documenting workarounds like `MAX_LOOK_AHEAD` with clear explanations of why they exist and what the ideal solution would be helps future maintainers understand the codebase.
5. The Builder pattern ecosystem across ripgrep's crates creates predictable, learnable APIs where users can apply the same mental model to configuration regardless of which subsystem they're working with.
6. The sink pattern decouples searchers from output formatting, allowing different printers to provide different sink implementations while the searcher remains unchanged.

## What to Read Next

How does the Standard printer actually format output? Read `crates/printer/src/standard.rs` to see the builder implementation, color integration, and match formatting logic.

What exactly goes into JSON output? Read `crates/printer/src/json.rs` to understand the structured output format and how it handles the complexity of multi-line matches and contextual information.

How do colors work at the terminal level? Read `crates/printer/src/color.rs` to see how color specifications are parsed, validated, and applied to output.

What statistics does ripgrep track? Read `crates/printer/src/stats.rs` to understand the metrics collected during searches and how they're aggregated.

# ripgrep crates/printer/src/color.rs: Terminal Colors

## What This File Does

This file implements ripgrep's color specification system, allowing users to customize how search results appear in their terminal. It provides a domain-specific language for expressing color preferences through simple strings like `"path:fg:magenta"` or `"match:style:bold"`, then transforms these user-facing specifications into the underlying terminal color commands.

The design separates what users want to say about colors from how those colors get applied. Users work with `UserColorSpec` values that they can parse from strings, while the printer works with `ColorSpecs` that aggregate all specifications into a ready-to-use format. This separation allows ripgrep to accept multiple color specifications from command-line arguments, configuration files, or defaults, then merge them together in a predictable way where later specifications override earlier ones.

## Section 1: Platform-Aware Default Colors

The file opens with a function that establishes ripgrep's default color scheme, and it immediately demonstrates a practical use of conditional compilation. Different terminal emulators on Unix and Windows have different default backgrounds and color palettes, so what looks good on one platform might be illegible on another.

Notice how the defaults use magenta for paths on Unix but cyan on Windows. This isn't arbitrary—it reflects the reality that Windows Terminal historically had different default colors than typical Unix terminals. The code uses `#[cfg(unix)]` and `#[cfg(windows)]` attributes to include only the appropriate default at compile time, meaning the resulting binary never contains the "wrong" platform's default.

The defaults also demonstrate how multiple specifications combine. The match output gets both a foreground color (red) and a style (bold) through separate specification strings. This layered approach—rather than a single complex specification—makes the system more composable and easier for users to understand and override.

See: Companion Code Section 1

## Section 2: A Domain-Specific Error Type

Color parsing involves multiple ways things can go wrong: an unrecognized output type, an invalid color name, a malformed specification format. Rather than collapsing these into a generic error, the code defines `ColorError` as an enum that captures exactly what went wrong and where.

Each variant carries the problematic input string, which enables helpful error messages. When you mistype a color name, the error message doesn't just say "invalid input"—it tells you which value was unrecognized and lists the valid alternatives. This attention to error quality is characteristic of well-designed command-line tools.

The `Display` implementation constructs user-facing messages that guide correction. Each message includes the invalid input, explains what was expected, and enumerates valid choices. This pattern—where errors are data structures that know how to explain themselves—scales well as the number of error cases grows.

See: Companion Code Section 2

---

## Section 3: The Two-Layer Specification Model

The file introduces two distinct types for color specifications: `UserColorSpec` and `ColorSpecs`. Understanding why both exist reveals an important design principle about interface boundaries.

`UserColorSpec` represents a single user intention: "I want paths to be magenta" or "I want matches to be bold." It preserves the user's expression in a structured form. Multiple `UserColorSpec` values can be created from command-line arguments, with later ones intended to override earlier ones.

`ColorSpecs` (plural) represents the merged result—a complete configuration ready for the printer to use. It has fields for each output type (path, line, column, match, highlight), each containing a fully-resolved `ColorSpec` from the `termcolor` crate. The printer doesn't need to understand how specifications were combined; it just asks for the color to use for paths and gets a direct answer.

This separation means the parsing and merging logic stays contained in this file, while the printing logic in other files remains simple. It's an application of the principle that data should be transformed into convenient forms at system boundaries rather than forcing every consumer to understand the raw input format.

See: Companion Code Section 3

## Section 4: Internal Type Vocabulary

Between the user-facing types and the `termcolor` library's types, the code defines several internal enums: `OutType`, `SpecType`, `SpecValue`, and `Style`. These types serve as an intermediate representation that captures all valid combinations without requiring string manipulation.

`OutType` enumerates where color applies: paths, line numbers, column numbers, matched text, or highlighted lines. `SpecType` distinguishes between foreground color, background color, style, and the special "none" that clears existing settings. `Style` captures the eight style options (bold, underline, italic, intense, plus their negations).

`SpecValue` is particularly interesting because it represents what a specification does rather than how it was expressed. A foreground color, a background color, and a style instruction are fundamentally different operations on a terminal color state, and `SpecValue` models this reality. The `None` variant exists for clearing specifications, which isn't a color or a style—it's an instruction to reset.

See: Companion Code Section 4

## Section 5: The Merge Strategy

The `ColorSpecs::new` function reveals how multiple specifications combine. It starts with default (empty) specifications and iteratively applies each user specification in order. The key insight is in the `merge_into` method: it modifies an existing `ColorSpec` rather than replacing it entirely.

This means specifications are additive by default. If you specify `"match:fg:red"` and later `"match:style:bold"`, you get red bold matches—the later specification adds to rather than replaces the earlier one. However, the `SpecValue::None` variant calls `clear()`, which resets everything. This gives users an escape hatch: specify `"match:none"` to clear all match styling, then add back just what you want.

The merge pattern also explains why there's a `default_with_color` constructor separate from the `Default` trait implementation. The `Default` trait returns empty specifications (no colors), while `default_with_color` applies the platform-appropriate defaults. This distinction lets code explicitly choose between "no styling" and "sensible defaults."

See: Companion Code Section 5

## Section 6: Parsing with FromStr

The `FromStr` implementation for `UserColorSpec` is where strings become structured data. The format `{type}:{attribute}:{value}` gets split on colons and validated piece by piece. Each piece uses its own `FromStr` implementation, creating a cascade of parsing that produces specific errors for specific failures.

The parsing logic handles the special case where `{attribute}` is "none" and no value is expected. For other specification types (fg, bg, style), it validates that a third component exists before attempting to parse it. This structure means invalid formats get caught early with clear messages, while valid formats flow through to the appropriate type parsers.

The individual `FromStr` implementations for `OutType`, `SpecType`, and `Style` all follow the same pattern: lowercase the input, match against known values, return the appropriate variant or an error carrying the original input. Case-insensitive matching makes the interface forgiving of user input variations.

See: Companion Code Section 6

---

## Section 7: Applying Specifications to Terminal State

The `merge_into` methods show how abstract specifications become concrete terminal commands. `SpecValue::merge_into` takes a mutable reference to a `ColorSpec` and modifies it based on what value the specification holds.

For colors, the code calls `set_fg` or `set_bg` with the parsed color. For styles, it sets the appropriate boolean flag. The style handling is verbose but clear—each style variant maps to exactly one method call with a boolean value. The "no" variants (nobold, nointense, etc.) set the flag to false, explicitly disabling rather than clearing.

This explicitness matters because terminal styling has state. If an earlier specification enabled bold and a later one sets a foreground color, you want the bold to persist. By only modifying the specific attribute each specification mentions, the merge preserves unrelated settings while updating the targeted ones.

See: Companion Code Section 7

## Section 8: Connection to the Builder Pattern Ecosystem

This color specification system connects to the broader builder pattern you've seen throughout ripgrep. Just as `SearcherBuilder` accumulates configuration before constructing a `Searcher`, the color system accumulates `UserColorSpec` values before constructing the final `ColorSpecs`.

The difference is that color specifications use an implicit builder through the `new` function rather than explicit builder methods. The pattern still applies: collect configuration incrementally, validate at boundaries, produce an immutable result for use. The merged `ColorSpecs` struct has only getter methods—once constructed, it doesn't change.

This immutability matters for thread safety. The printer holds a `ColorSpecs` and can access it from any thread without synchronization, because no thread can modify it. The mutability boundary exists during construction, contained within the `new` function's scope.

See: Companion Code Section 8

---

## Key Takeaways

1. **Platform-aware defaults through conditional compilation**: Rather than runtime checks, the code uses `#[cfg]` attributes to select appropriate defaults at compile time, eliminating dead code and runtime overhead.
2. **Domain-specific error types improve user experience**: Each error variant captures context about what went wrong, enabling messages that guide users toward valid inputs rather than just reporting failure.
3. **The two-layer type design separates user intent from printer needs**: `UserColorSpec` captures what users say, while `ColorSpecs` provides what printers need, with transformation logic contained at the boundary.
4. **Merge semantics enable compositional configuration**: Later specifications override earlier ones, but only for the specific attributes they mention, allowing incremental refinement of styling.
5. **FromStr implementations cascade through the type hierarchy**: Each component type knows how to parse itself, and the top-level parser orchestrates them, producing specific errors at the appropriate level.
6. **Terminal styling is fundamentally stateful**: The merge pattern acknowledges that setting a color shouldn't clear a style, reflecting how actual terminal control sequences work.

## What to Read Next

How does the printer actually use these color specifications when writing output? Read `crates/printer/src/standard.rs` to see `ColorSpecs` in action.

What terminal abstraction does ripgrep use under these specifications? The `termcolor` crate documentation explains the cross-platform terminal color handling that `ColorSpec` provides.

How do users specify colors on the command line? The argument parsing in `crates/core/app.rs` shows how `--colors` arguments become `UserColorSpec` values.

# ripgrep crates/printer/src/util.rs: Printer Utilities

## What This File Does

This file contains the shared utility infrastructure that powers ripgrep's printing system. While the main printer types handle the logic of *what* to print, this utility module provides the *how*—the foundational abstractions for text replacement, path handling, performance-optimized formatting, and the careful coordination between the searcher and printer layers.

The utilities here solve several thorny problems that arise when building a high-performance text search tool. How do you efficiently perform regex replacements without allocating memory on every match? How do you handle file paths portably across Unix and Windows while supporting terminal hyperlinks? How do you format numbers without the overhead of Rust's standard formatting machinery? Each utility in this file represents a carefully considered answer to these questions, and together they form the infrastructure that makes ripgrep's output both fast and flexible.

---

## Section 1: The Replacer Pattern for Amortized Allocation

One of the most expensive operations in text processing is memory allocation. If you're searching through millions of lines and performing replacements on each match, allocating new buffers for every replacement would create significant overhead. The `Replacer` struct solves this problem through a pattern you'll see throughout high-performance Rust code: amortized allocation with lazy initialization.

The design philosophy here is to delay allocation until absolutely necessary, then reuse that allocation across multiple operations. The `Replacer` wraps an `Option<Space<M>>`, where `Space` contains three reusable buffers: one for capture locations, one for the replacement output, and one for tracking match positions. When you create a new `Replacer` with `Replacer::new()`, it allocates nothing—the `space` field is simply `None`.

Only when you call `replace_all` does the `allocate` method check whether space exists and create it if needed. Subsequent calls to `replace_all` reuse the same buffers, clearing them but not deallocating them. This pattern is particularly effective when processing many files, as the buffers grow to accommodate the largest replacement and then stabilize.

The generic parameter `M: Matcher` is essential here. Different regex engines have different capture group representations, so the `Replacer` must be parameterized over the matcher type. The `Space` struct stores `M::Captures`, which means it can hold capture data for any matcher implementation without runtime type checking.

See: Companion Code Section 1

---

## Section 2: Multi-Line Mode and the Look-Ahead Problem

The `replace_all` method contains some of the most subtle code in the entire printer system, dealing with a fundamental tension between how regex engines work and how ripgrep presents search results. The extensive comments in this function reveal a hard-won understanding of edge cases that only emerge in production use.

The core issue is look-ahead in regular expressions. When a regex contains patterns like `(?=...)` (positive look-ahead) or `$` (end of line), the match depends not just on the matching text but on what comes *after* it. When ripgrep reports a match to the printer, it typically provides just the matching lines. But if the regex required look-ahead beyond those lines, re-running the match during replacement could fail even though the original search succeeded.

The solution involves carefully managing how much context the replacement engine can see. In multi-line mode, the code extends the search buffer by `MAX_LOOK_AHEAD` bytes beyond the reported match, giving the regex engine enough context to reproduce its original matches. In single-line mode, the opposite problem arises: the regex might *fail* to match because it can see the line terminator. So the code trims the line terminator before attempting the replacement, then adds it back afterward.

This dance between the searcher and printer represents what the comments describe as a potential abstraction boundary violation. Ideally, the searcher would provide all the match information the printer needs, but that would impose costs even when matches aren't needed for counting. The current design accepts some complexity in the printer to avoid penalizing the common case.

See: Companion Code Section 2

## Section 3: The Sunk Abstraction for Unified Match Handling

The `Sunk` struct provides a clean abstraction over two different types of search results: actual matches (`SinkMatch`) and contextual lines (`SinkContext`). When you use ripgrep with context options like `-A` (after context) or `-B` (before context), the printer receives both types of results but often wants to handle them uniformly.

The name "Sunk" cleverly references the "Sink" pattern used by the searcher—results flow into a sink, and `Sunk` represents something that has "sunk" into that sink. This abstraction carries all the metadata a printer might need: the actual bytes, the absolute byte offset in the file, optional line numbers, the kind of context (before, after, or passthrough), and both the current matches and the original matches before any replacement.

The distinction between `matches` and `original_matches` is crucial for replacement scenarios. When you perform a replacement, the byte positions of matches shift because the replacement text has different length than the original. The `matches` field contains positions in the *replaced* text, while `original_matches` preserves the positions in the original. This lets the printer correctly highlight both the replacement output and preserve information about what was originally matched.

The factory methods `from_sink_match` and `from_sink_context` handle the conditional logic of using either replacement data or original data, encapsulating this complexity so that printer code can work with `Sunk` values uniformly.

See: Companion Code Section 3

---

## Section 4: Cross-Platform Path Handling

The `PrinterPath` struct addresses one of the genuinely difficult problems in cross-platform Rust: file path representation. On Unix, paths are just byte sequences with no encoding requirement. On Windows, paths are UTF-16 encoded and can contain characters that don't have valid UTF-8 representations. The standard library's `Path` type abstracts over these differences, but when you need to *output* a path as bytes—for piping to another program or displaying in a terminal—the complexity resurfaces.

The design uses conditional compilation (`#[cfg(unix)]` and `#[cfg(not(unix))]`) to optimize for the common case while maintaining correctness on all platforms. On Unix, the struct only stores the byte representation, since you can reconstruct a `Path` from bytes with zero cost. On Windows, it stores both the original `Path` reference and a byte representation, because you cannot safely convert bytes back to a Windows path.

The `OnceCell<Option<HyperlinkPath>>` field demonstrates lazy computation. Terminal hyperlinks (clickable file paths in modern terminals) require path canonicalization, which involves filesystem operations that can fail. Rather than computing this upfront, the `OnceCell` ensures it's computed only once, on first access, and the `Option` handles the case where canonicalization fails.

The `with_separator` method reveals another practical concern: users sometimes want different path separators, particularly in Windows environments running under Cygwin or WSL where `/` might be preferred over `\`. The method uses the builder pattern to create a modified path with replaced separators.

See: Companion Code Section 4

---

## Section 5: Performance-Optimized Number Formatting

The `DecimalFormatter` struct might seem like overkill—after all, Rust's standard library can format numbers just fine with `format!("{}", n)`. But ripgrep's performance requirements justify this optimization. When displaying line numbers for millions of matches, the overhead of the standard formatting machinery becomes measurable.

The implementation uses a fixed-size buffer of 20 bytes (the maximum length of a `u64` in decimal) and fills it from right to left using basic arithmetic. The algorithm is simple: repeatedly divide by 10 to extract digits, convert each digit to its ASCII representation by adding `b'0'`, and track where the number starts in the buffer. The `as_bytes` method then returns a slice from that start position.

The comment acknowledges that the `itoa` crate is faster, but the team chose to implement this directly to avoid the dependency while still getting meaningful performance improvement. This decision reflects ripgrep's general philosophy of being conservative about dependencies while accepting well-motivated complexity in the codebase itself.

The `try_from` call for converting the remainder to `u8` will always succeed (since `n % 10` is always 0-9), but using `try_from` with `unwrap` is both more explicit about the programmer's intent and slightly cleaner than a direct cast.

See: Companion Code Section 5

## Section 6: Duration Formatting for Human and Machine Readers

The `NiceDuration` wrapper demonstrates how to make Rust's standard types more ergonomic for specific use cases. The standard `Duration` type is excellent for computation but its `Debug` and `Display` implementations aren't optimized for user-facing output. `NiceDuration` provides a display format that shows seconds with six decimal places of precision—human-readable while preserving accuracy.

More interestingly, the optional Serde serialization shows thoughtful API design for JSON output. When serialized, `NiceDuration` produces three fields: `secs` (the integer seconds), `nanos` (the sub-second nanoseconds), and `human` (the formatted string). This serves both machines (which can reconstruct the exact duration from `secs` and `nanos`) and humans (who can read the `human` field directly).

The `#[cfg(feature = "serde")]` conditional compilation means this serialization code only exists when the `serde` feature is enabled. This keeps the base compilation fast and the binary small for users who don't need JSON output.

See: Companion Code Section 6

---

## Section 7: Match Discovery in Context

The `find_iter_at_in_context` function encapsulates the match-finding logic needed by printers, handling the same multi-line complexity discussed earlier in the replacement code. The lengthy comment block is worth reading carefully because it documents not just what the code does, but the fundamental architectural tension it addresses.

The problem, restated: when the searcher reports a match, it gives the printer the matching bytes. But to highlight individual matches within those bytes, the printer needs to re-run the regex. In multi-line mode, the regex might have used look-ahead that extended beyond the bytes provided. In single-line mode, the regex might be sensitive to line terminators that need to be hidden.

The function signature uses a callback pattern (`matched: F`) rather than returning a collection. This is more efficient because it avoids allocating a vector of matches and allows the caller to short-circuit iteration by returning `false`. The callback receives each `Match` as it's found, and the printer can use these positions to apply syntax highlighting or other formatting.

The `.map_err(io::Error::error_message)` at the end converts matcher errors to IO errors, maintaining a consistent error type throughout the printer layer despite the underlying diversity of regex engine error types.

See: Companion Code Section 7

## Section 8: Line Terminator Trimming

The `trim_line_terminator` function handles an edge case that seems simple but has subtle implications: removing line terminators from the end of match data. The function must handle both Unix-style (`\n`) and Windows-style (`\r\n`) line endings, and it must do so in a way that works with the searcher's configured line terminator.

The design returns a slice of the trimmed bytes rather than just indicating how many bytes were trimmed. This is important for replacement operations, where the line terminator needs to be added back after processing. By returning the actual trimmed bytes, the caller can easily append them to the result.

The `is_crlf()` check and the subsequent byte inspection show careful handling of Windows line endings. When the line terminator is CRLF, simply trimming one byte would leave a dangling carriage return, so the code checks for and removes both characters.

See: Companion Code Section 8

---

## Section 9: Bounded Replacement with Captures

The private `replace_with_captures_in_context` function extends the matcher's standard replacement API with end-bound awareness. The standard `replace_with_captures_at` method takes a start position but no end position, which creates problems when you only want to replace matches within a specific range.

The implementation manually tracks the "last match end" position and builds the result incrementally. For each capture, it copies the bytes between the last match and the current match start, then calls the `append` callback to generate the replacement. The callback pattern allows the caller to control exactly how captures are interpolated into the replacement string.

The end-of-buffer handling is particularly careful. If a match extends beyond the intended range (possible with multi-line patterns), the code uses the full buffer length. Otherwise, it respects the range boundary. Finally, the line terminator is appended—this is where the value returned by `trim_line_terminator` gets used.

See: Companion Code Section 9

---

## Section 10: Whitespace Trimming for Clean Output

The `trim_ascii_prefix` function provides a utility for cleaning up match output by removing leading whitespace. This is useful when ripgrep needs to display matches without excessive indentation, particularly in code search scenarios where lines might have deep nesting.

The function demonstrates Rust's iterator adaptor pattern. Rather than writing a manual loop, it uses `take_while` with a predicate that checks whether each byte is both a whitespace character and not a line terminator. The count of matching bytes becomes the amount to trim from the range.

The line terminator check is crucial: without it, the function might trim across line boundaries in multi-line scenarios, fundamentally changing what gets displayed. By stopping at line terminators, the function ensures it only trims horizontal whitespace within a single line.

The `is_space` helper function defines whitespace in terms of ASCII bytes rather than using a standard library function. This explicit definition ensures consistent behavior across platforms and makes the whitespace definition visible and auditable.

See: Companion Code Section 10

---

## Key Takeaways

First, amortized allocation through lazy initialization is a powerful pattern for performance-critical code. The `Replacer` struct demonstrates how to defer allocation until needed and then reuse buffers across operations.

Second, abstraction boundaries between search and print layers create genuine complexity. The look-ahead handling in `replace_all` and `find_iter_at_in_context` shows how practical concerns can complicate otherwise clean designs.

Third, the `Sunk` type exemplifies using abstraction to unify related but distinct types. By providing a common interface over matches and context lines, it simplifies printer implementations.

Fourth, cross-platform path handling requires conditional compilation and careful consideration of platform-specific byte representations. The `PrinterPath` design shows how to optimize for the common case while maintaining correctness everywhere.

Fifth, sometimes bypassing standard library facilities for custom implementations is justified. The `DecimalFormatter` trades generality for performance in a measurable way.

Sixth, designing for both human and machine consumption simultaneously requires thoughtful serialization. The `NiceDuration` Serde implementation shows how to serve both audiences from a single type.

---

## What to Read Next

How do these utilities get used in actual printing? Read `crates/printer/src/standard.rs` to see the `Replacer` and `Sunk` types in action within the standard printer implementation.

How does the searcher layer that feeds these printers work? Read `crates/searcher/src/searcher/core.rs` to understand the `SinkMatch` and `SinkContext` types that `Sunk` abstracts over.

What is the `HyperlinkPath` type that `PrinterPath` references? Read `crates/printer/src/hyperlink.rs` to understand how terminal hyperlinks are constructed from file paths.

How do the color specifications from the previous lesson interact with these utilities? Read `crates/printer/src/color.rs` to see how `ColorSpecs` work alongside match positions to produce highlighted output.

# ripgrep crates/printer/src/json.rs: JSON Output

## What This File Does

This file implements ripgrep's JSON Lines printer, which transforms search results into a structured, machine-readable format. While the standard printer produces human-readable terminal output with colors and formatting, the JSON printer creates output designed for programmatic consumption—perfect for piping into other tools, building editor integrations, or constructing search UIs.

The implementation demonstrates several sophisticated patterns: a builder for configuration, a sink that adapts to the searcher's streaming interface, careful handling of non-UTF-8 data in a format that demands UTF-8, and memory-efficient representation of match data. The extensive documentation in this file also serves as the specification for ripgrep's JSON wire format, making it both an implementation and a contract.

## Section 1: The Configuration and Builder Pattern

The JSON printer follows the same builder pattern we've seen throughout ripgrep, but with notably fewer configuration options than the standard printer. This isn't a limitation—it's a deliberate design choice. The `Config` struct holds just three fields: whether to pretty-print, whether to always emit begin/end messages, and an optional replacement pattern.

The builder pattern here serves the same purpose it does elsewhere in ripgrep: separating configuration from operation, enabling method chaining, and ensuring that once a printer is built, its configuration is frozen. Notice how `Config` uses `Arc<Option<Vec<u8>>>` for the replacement bytes. The `Arc` allows the configuration to be cloned cheaply when building the printer, while the `Option<Vec<u8>>` pattern lets users specify "no replacement" distinctly from "empty replacement."

The `JSONBuilder` methods return `&mut JSONBuilder` to enable fluent configuration chains. Each method simply sets a field in the internal `Config` and returns `self`. When `build` is called, the configuration is cloned into the new `JSON` printer, severing the relationship between builder and printer.

See: Companion Code Section 1

## Section 2: The JSON Printer Structure

The `JSON` struct itself is surprisingly simple: it holds the frozen configuration, a `CounterWriter` wrapping the output destination, and a reusable vector for storing matches. The `CounterWriter` is a wrapper type that tracks how many bytes have been written, which becomes important for statistics and for determining whether any output has been produced.

The `matches` vector deserves attention. Rather than allocating a new vector for every match line, the printer reuses this single vector, clearing it between uses. This amortization pattern appears throughout performance-critical Rust code—pay the allocation cost once, then reuse the buffer.

The `new` constructor delegates to `JSONBuilder::new().build(wtr)`, demonstrating that the builder is the canonical way to construct this type. Users who want defaults can call `new` directly; users who need customization use the builder.

See: Companion Code Section 2

---

## Section 3: The Sink Abstraction

The `JSONSink` type implements the `Sink` trait from the searcher crate, acting as the bridge between the search engine's streaming results and the JSON output format. This is a callback-based architecture: the searcher calls methods on the sink as it discovers matches, context lines, and other events.

The sink carries substantial state: the matcher (for finding individual match positions within lines), a replacer (for optional text substitution), a reference back to the parent `JSON` printer, an optional file path, timing information, match counts, and accumulated statistics. The lifetime parameters `'p` and `'s` carefully track the borrowed path and the borrowed printer reference.

This design separates concerns cleanly. The `JSON` type owns the output infrastructure and configuration. The `JSONSink` type handles the search-specific state that lives only for the duration of a single search operation. When you call `sink()` or `sink_with_path()` on a `JSON` printer, you get back a `JSONSink` that's ready to receive search results.

See: Companion Code Section 3

## Section 4: The Wire Format and Message Types

The JSON Lines format emits one JSON object per line, with each object wrapped in an envelope containing a `type` field and a `data` field. The four message types—`begin`, `end`, `match`, and `context`—form a complete protocol for describing search results.

The `write_message` method handles the serialization, choosing between compact and pretty-printed output based on configuration. After serializing, it always appends a newline, maintaining the JSON Lines format invariant. The method signature takes a reference to a `jsont::Message`, where `jsont` is an internal module defining the serializable message structures.

The message types mirror the sink callbacks. When a search begins, the sink may emit a `begin` message. When matches are found, it emits `match` messages. Context lines become `context` messages. When the search completes, it emits an `end` message with summary statistics. The `always_begin_end` configuration controls whether begin/end are emitted even when no matches are found—useful for tools that need to know a file was searched even if nothing matched.

See: Companion Code Section 4

---

## Section 5: Handling Non-UTF-8 Data

One of the most interesting challenges in this file is representing arbitrary bytes in JSON, which is inherently a UTF-8 format. Search results might contain invalid UTF-8—binary files, files with mixed encodings, or paths on systems where filenames aren't required to be valid UTF-8.

The solution is an "arbitrary data object" that uses either a `text` key (for valid UTF-8) or a `bytes` key (for base64-encoded data). This design ensures lossless representation: consumers can always recover the original bytes, either by using the string directly or by base64 decoding. The printer guarantees that `text` is used whenever possible, so most output remains human-readable.

This pattern appears throughout the wire format: paths, match content, and replacement text all use this dual representation. The documentation extensively describes this encoding, which is important because the JSON output serves as a stable interface for external tools.

See: Companion Code Section 5

## Section 6: Recording Matches Within Lines

The `record_matches` method populates the printer's `matches` vector with the precise positions of each match within a line. This is necessary because a single line might contain multiple matches,

and the JSON format reports each one with its start and end offsets.

The method calls `find_iter_at_in_context`, a utility function that handles the complexity of finding matches while respecting context settings. The closure passed to this function adjusts the match offsets to be relative to the start of the line rather than the start of the buffer—a subtle but important normalization.

There's also special handling for empty matches at the end of the input. Empty matches (from patterns like `.*` matching zero characters) at the end of a line are discarded. This prevents confusing output where a match appears to exist past the end of the content.

See: Companion Code Section 6

---

## Section 7: The Match and Context Sink Callbacks

The `matched` and `context` callbacks are where search results become JSON output. When `matched` is called, the sink increments its match count, ensures the begin message has been printed, records match positions, optionally computes replacements, and then constructs and writes the match message.

The `SubMatches` type provides an efficient way to represent the match ranges within a line. Rather than always allocating a vector, it uses an enum that handles the common case of exactly one match per line with a fixed-size array. This small-vector optimization avoids allocation in the vast majority of cases.

The `context` callback is similar but handles contextual lines—lines adjacent to matches that provide surrounding context. Interestingly, context lines can contain submatches when the search is inverted (showing lines that don't match). This might seem counterintuitive, but it allows tools to highlight what the pattern would have matched in the non-matching context.

See: Companion Code Section 7

---

## Section 8: The Begin and Finish Callbacks

The `begin` callback runs at the start of each search, resetting per-search state: clearing the byte counter, recording the start time, zeroing the match count, and clearing the binary offset. If `always_begin_end` is configured, it also writes the begin message immediately; otherwise, the begin message is deferred until the first match or context line.

The `finish` callback runs after the search completes, regardless of whether matches were found. It records the binary detection offset (if binary data caused the search to stop), accumulates timing statistics, and updates search counts. If a begin message was printed, it writes the corresponding end message with comprehensive statistics.

The conditional logic around `begin_printed` is subtle but important. If `always_begin_end` is false and no matches were found, neither begin nor end messages are written. This keeps the output clean when searching many files—only files with matches produce any output.

See: Companion Code Section 8

---

## Section 9: The SubMatches Optimization

The `SubMatches` enum is a beautiful example of optimizing for the common case. Most lines contain exactly one match, so allocating a `Vec` for every line would be wasteful. Instead, `SubMatches` uses three variants: `Empty` for no matches, `Small` for exactly one match (stored in a fixed-size array), and `Big` for multiple matches (requiring heap allocation).

The `new` constructor checks the number of matches and chooses the appropriate variant. The `as_slice` method then returns a uniform `&[jsont::SubMatch]` regardless of which variant is in use, hiding the optimization from calling code.

This pattern—small vector optimization—appears frequently in performance-sensitive Rust code. The standard library's `SmallVec` crate generalizes this pattern, but here a custom enum provides exactly the behavior needed with minimal overhead.

See: Companion Code Section 9

---

## Section 10: Binary Data Handling

The `binary_data` callback is called when the searcher detects binary content. The JSON printer logs this detection at debug level but doesn't prevent the search from continuing. The actual handling depends on the searcher's configuration—specifically, whether a quit byte was set.

This callback stores the offset where binary data was found, which later appears in the end message. Consumers of the JSON output can use this to understand why a search might have produced partial results or no results for a particular file.

The interaction between the printer and searcher here demonstrates the separation of concerns: the searcher decides policy (whether to quit on binary data), while the printer merely reports what happened.

See: Companion Code Section 10

---

## Section 11: Statistics Accumulation

The `Stats` type accumulates comprehensive statistics across searches: elapsed time, bytes searched, bytes printed, match counts, and line counts. These statistics appear in every `end` message and can be used by consumers to understand search performance and coverage.

The statistics accumulation happens in `finish`: elapsed time is computed from `start_time`, search counts increment, and byte counts are recorded. The `stats` field on the sink persists across calls, allowing statistics to accumulate if the same sink is reused (though in practice, sinks are typically created fresh for each search).

The `bytes_printed` statistic uses the `CounterWriter`'s count, demonstrating why that wrapper exists. Without it, the printer would need to manually track output size, duplicating bookkeeping that the writer can handle transparently.

See: Companion Code Section 11

---

## Key Takeaways

1. The JSON printer uses the same builder pattern as other ripgrep components, but with deliberately fewer options because structured output doesn't need the formatting flexibility of terminal output.
2. The sink abstraction provides a clean callback-based interface between the searcher (which finds matches) and the printer (which formats output), allowing different printers to share the same search infrastructure.
3. Non-UTF-8 data in a UTF-8 format requires careful design—the arbitrary data object pattern with `text` and `bytes` keys ensures lossless representation while keeping output readable when possible.
4. Small vector optimizations like `SubMatches` avoid heap allocation in the common case of one match per line, demonstrating how Rust's enums enable efficient representation of variant

data.

5. The conditional begin/end message emission keeps output clean when searching many files, printing nothing for files without matches unless explicitly configured otherwise.
6. Comprehensive statistics in every `end` message enable consumers to understand both search results and search performance without needing separate tooling.

---

## What to Read Next

How does the JSON message format get serialized? Read `crates/printer/src/jsont.rs` for the serde-derived structures that define the wire format.

What about human-readable output? Read `crates/printer/src/standard.rs` for the standard printer, which has far more configuration options and formatting complexity.

How does the searcher produce the events that drive this printer? Revisit `crates/searcher/src/searcher/core.rs` to see how matches are found and reported to sinks.

What statistics does the printer track? Read `crates/printer/src/stats.rs` to understand the `Stats` type and how metrics are accumulated.

# ripgrep crates/printer/src/summary.rs: Summary Output

## What This File Does

This file implements ripgrep's summary printer, which produces aggregate output about search results rather than showing individual matches. When you run `rg --count`, `rg --files-with-matches`, or `rg --files-without-match`, you're using the summary printer. Instead of displaying each matching line, these modes condense results into counts or simple file listings—output that answers questions like "how many matches?" or "which files matched?" rather than "what matched?"

The summary printer represents a fascinating counterpoint to the standard printer we studied earlier. Where the standard printer must handle complex formatting of individual matches with context lines and highlighting, the summary printer focuses on efficient aggregation and early termination. This file demonstrates how the same Sink trait interface can serve radically different output requirements, and how the builder pattern creates a consistent configuration experience across the printer ecosystem.

---

## Section 1: The Six Summary Modes

The `SummaryKind` enum defines six distinct output modes, and understanding these modes is essential to understanding the rest of the file. Each mode answers a different question a user might ask about their search results. `Count` tells you how many lines matched, while `CountMatches` tells you how many individual matches occurred (potentially multiple per line). `PathWithMatch` lists files that contain matches, and its inverse `PathWithoutMatch` lists files without matches. The `QuietWithMatch` and `QuietWithoutMatch` modes produce no output at all but affect ripgrep's exit code.

What makes this enum particularly interesting is how each variant determines the printer's behavior through three key methods. The `requires_path` method indicates whether a mode makes sense without a file path—clearly, `PathWithMatch` needs a path to print, while `Count` can simply output a number. The `requires_stats` method signals whether a mode needs detailed match tracking even if the user didn't request statistics. The `quit_early` method determines

whether the search can stop after finding the first match. These three properties create a matrix of behaviors that the rest of the code consults repeatedly.

See: Companion Code Section 1

---

## Section 2: Configuration Through the Builder Pattern

The `Config` struct and `SummaryBuilder` follow the same builder pattern we've seen throughout ripgrep's codebase. The builder holds a `Config` instance that accumulates settings, then clones that configuration into the final `Summary` printer. This pattern appears in the searcher, the standard printer, and the ignore walker—a consistent approach that makes ripgrep's API predictable for users who work with multiple components.

The configuration options reveal the flexibility hidden behind simple command-line flags. The `separator_field` setting controls what character appears between a file path and its match count—defaulting to a colon but configurable for machine parsing. The `separator_path` option allows replacing path separators, addressing the common need for Unix-style paths on Windows systems. The `exclude_zero` setting determines whether files with zero matches appear in count output. Each option solves a real user need without complicating the common case.

Notice how the builder methods follow a consistent pattern: each takes `&mut self`, sets a field on `self.config`, and returns `&mut SummaryBuilder` for chaining. This mirrors the pattern in `SearcherBuilder` and `StandardBuilder`, creating a uniform experience across the codebase. The `build` method consumes the configuration to create the printer, while `build_no_color` provides a convenience wrapper that removes the need to explicitly import `termcolor::NoColor`.

See: Companion Code Section 2

---

## Section 3: The Summary Printer and RefCell

The `Summary` struct itself is remarkably simple—just a configuration and a writer wrapped in `RefCell<CounterWriter<W>>`. But that `RefCell` deserves attention because it represents a deliberate architectural choice. The printer needs interior mutability because the `Sink` trait methods receive `&mut self`, but the internal write methods only need `&self`. Using `RefCell` allows the write methods to borrow the writer mutably even when they only have shared access to the printer.

The `CounterWriter` wrapper tracks how many bytes have been written, enabling the `has_written` method to report whether any output occurred. This matters for modes like `PathWithMatch` where the presence or absence of output conveys information. The counter also feeds into statistics gathering, letting ripgrep report how many bytes it printed during a search.

The printer provides two ways to create a sink: `sink` without a path and `sink_with_path` with one. This split addresses the fact that some output modes require a file path while others don't. When you call `sink` in `PathWithMatch` mode, the search will immediately fail with an error—better to fail fast than produce confusing output. The `sink_with_path` method creates a `PrinterPath` that handles path separator normalization, ensuring consistent output regardless of the operating system.

See: Companion Code Section 3

---

## Section 4: The SummarySink State Machine

The `SummarySink` struct implements the `Sink` trait, receiving callbacks from the searcher as it processes input. Unlike a simple accumulator, `SummarySink` must track multiple pieces of state that interact in subtle ways. The `match_count` field accumulates matches, but its meaning differs between single-line and multi-line modes. The `binary_byte_offset` field records whether binary data was detected, which can cause the entire search result to be discarded. The optional `stats` field gathers detailed statistics when requested.

The type parameters on `SummarySink` tell a story about lifetimes and ownership. The `'p` lifetime refers to the file path, which might be borrowed from the caller or might be `'static` if no path was provided. The `'s` lifetime refers to the `Summary` printer being borrowed. The `M` generic is the matcher type, and `W` is the writer type. This proliferation of generics allows the sink to work with any combination of matchers and writers without runtime dispatch.

The `start_time` field enables performance tracking, measuring how long each search takes. Combined with the byte counters, this gives ripgrep the data it needs for its `--stats` flag. The hyperlink interpolator prepares clickable links when terminal support is available, threading the feature through even this minimal output mode.

See: Companion Code Section 4

---

## Section 5: Match Counting Subtleties

The `matched` callback handles a deceptively complex task: counting matches accurately across different search modes. In single-line mode with no statistics, the count is simple—each callback represents one matching line. But in multi-line mode or when statistics are enabled, a single callback might represent a line with multiple matches, and the printer needs to count each one.

The code uses `find_iter_at_in_context` to re-scan the matched region and count individual matches. This might seem redundant—the searcher already found these matches—but the separation of concerns in ripgrep's architecture means the searcher reports match locations, not counts. The comment acknowledges that `find_iter_at_in_context` is a "giant kludge internally" and might miss matches in edge cases involving look-around assertions, so the code ensures at least one match is recorded.

The early termination logic appears here too. When `quit_early` returns true (for `PathWithMatch` or `QuietWithMatch` modes) and statistics aren't needed, the callback returns `Ok(false)` to signal that the search should stop. This optimization matters because users often search large codebases just to find which files contain matches—there's no need to find every match in every file.

See: Companion Code Section 5

## Section 6: Binary Data Handling

The `binary_data` callback addresses a thorny problem: what happens when a search encounters binary content? The method logs the encounter but returns `Ok(true)` to continue searching. The actual decision about whether to report results happens later, in the `finish` method.

This deferred handling exists because binary data might appear partway through a file. If the printer already found matches before hitting binary content, should it report those matches? The answer depends on the binary detection mode. When using "quit on binary" detection, finding binary data invalidates the entire file's results—even matches found before the binary content. This prevents misleading output where a file appears to have fewer matches than it actually contains.

The comment in the code acknowledges an inconsistency: in `PathWithMatch` mode, the printer might have already quit after finding the first match, before ever encountering binary data. So the same file might be listed by `--files-with-matches` but excluded by `--count`. The authors explicitly accept this inconsistency as a tradeoff for performance—fixing it would require scanning entire files even when only checking for match existence.

See: Companion Code Section 6

## Section 7: Search Lifecycle Management

The `begin` and `finish` methods bracket each search, handling setup and output generation. The `begin` method validates preconditions—ensuring a path exists when required—and resets per-search state. The `finish` method is where actual output occurs, dispatching on `SummaryKind` to produce the appropriate format.

The timing of output is significant. Unlike the standard printer, which emits output incrementally as matches arrive, the summary printer waits until the search completes. This isn't just about gathering totals—it's about knowing whether to output anything at all. In `PathWithoutMatch` mode, output only occurs if no matches were found. In `Count` mode with `exclude_zero` enabled, files with no matches produce no output.

The `finish` method also updates statistics if enabled, recording elapsed time, bytes searched, bytes printed, and match counts. These statistics aggregate across all searches performed with a given sink, enabling ripgrep's summary output that shows totals across all searched files.

See: Companion Code Section 7

---

## Section 8: Output Formatting Details

The private write methods handle the mechanics of producing formatted output. The `write_path_line` and `write_path_field` methods share logic for writing paths but differ in what follows—a line terminator versus a field separator. Both respect the `path_terminator` configuration, which allows null-terminated output for machine parsing (the `-0` flag).

The `write_path` method applies styling when colors are configured, using the same `ColorSpecs` system as the standard printer. It also handles hyperlinks, wrapping the path in escape sequences that terminals can interpret as clickable links. This feature appears even in summary output, where clicking a file path in a match count listing might open that file.

The `write_spec` method demonstrates the `WriteColor` protocol: set the color specification, write the content, then reset to default. This pattern ensures color state doesn't leak between output elements. The underlying `CounterWriter` tracks all bytes written through these methods, feeding the statistics system.

See: Companion Code Section 8

## Section 9: Testing the Summary Printer

The test module validates the printer's behavior across its various modes and configuration options. The tests construct a matcher with `grep_regex::RegexMatcher`, configure a printer with `SummaryBuilder`, then run a search against a sample text. The `printer_contents` helper extracts the output for comparison.

Several tests verify error conditions. Attempting to use `PathWithMatch` mode without providing a path produces an error—the test confirms this by checking that the result is an error rather than asserting on output. This validates the precondition checking in `begin`.

The statistics interaction tests are particularly instructive. The "quiet" test shows that `QuietWithMatch` stops after finding one match, but "quiet_with_stats" shows that enabling statistics forces the search to continue. The match count jumps from 1 to 3, demonstrating how statistics gathering affects the search's early termination behavior.

See: Companion Code Section 9

---

## Key Takeaways

First, the `SummaryKind` enum's methods—`requires_path`, `requires_stats`, and `quit_early`—create a behavior matrix that drives decisions throughout the code, showing how enumerated types can encapsulate policy decisions.

Second, the `RefCell<CounterWriter<W>>` pattern enables interior mutability for the writer while maintaining `&self` signatures on internal methods, a common Rust idiom when shared references need to modify state.

Third, deferred output generation in `finish` rather than incremental output in `matched` enables inverse operations like `PathWithoutMatch`, where you can only know what to print after seeing all inputs.

Fourth, the binary data handling demonstrates intentional inconsistency—accepting that `PathWithMatch` and `Count` may produce different results for the same file when binary content exists, trading correctness for performance.

Fifth, the builder pattern's consistent application across `SummaryBuilder`, `StandardBuilder`, and `SearcherBuilder` creates a unified API experience that reduces cognitive load when working with multiple ripgrep components.

Sixth, the statistics system's integration shows how optional features can influence core behavior—enabling statistics prevents early termination, a side effect users must understand when benchmarking.

---

## What to Read Next

How does the standard printer handle individual match output? Read `crates/printer/src/standard.rs` to see the more complex output formatting for line-by-line results.

What statistics does ripgrep track? Read `crates/printer/src/stats.rs` to understand the `Stats` struct that accumulates timing and count information.

How does the searcher invoke these sink callbacks? Read `crates/searcher/src/searcher/mod.rs` to see the search loop that drives the `Sink` trait.

How do hyperlinks work in terminal output? Read `crates/printer/src/hyperlink.rs` to understand the OSC 8 escape sequence formatting.

# ripgrep crates/globset/src/lib.rs: Glob Pattern Matching

## What This File Does

This file implements a high-performance glob pattern matching system that forms the foundation of ripgrep's file filtering capabilities. When you tell ripgrep to search only `*.rs` files or ignore `target/**`, this is the code that makes those patterns work. The `globset` crate provides two levels of abstraction: individual `Glob` matchers for single patterns and `GlobSet` for matching multiple patterns simultaneously against a single path.

What makes this implementation particularly interesting is its **strategy-based optimization**. Rather than treating all glob patterns the same way, the code analyzes each pattern and routes it to the most efficient matching algorithm. A simple extension match like `*.rs` doesn't need the full power of regular expressions—it can be handled with a hash table lookup. This tiered approach is a masterclass in practical performance optimization, where the common cases are made fast while still supporting the full generality of glob syntax.

See: Companion Code Section 1

---

## Section 1: The Builder Pattern in Context

The `globset` crate continues ripgrep's consistent use of the Builder pattern, connecting directly to patterns we've seen in the searcher, printer, and ignore modules. Here, we see two builders: `GlobBuilder` for configuring individual patterns and `GlobSetBuilder` for assembling collections of patterns.

The `GlobSetBuilder` follows the familiar accumulate-then-build pattern. You create a builder, add patterns incrementally, and then call `build()` to produce an immutable, optimized matcher. This separation is particularly important for `GlobSet` because the build phase performs significant optimization work—analyzing patterns, categorizing them by type, and constructing specialized data structures. You wouldn't want to repeat that work every time you add a pattern.

Notice how `GlobSet::builder()` provides an alternative constructor that returns a `GlobSetBuilder`. This is a convenience that ripgrep uses throughout its codebase, allowing both

`GlobSetBuilder::new()` and `GlobSet::builder()` depending on what reads more naturally at the call site. The `GlobSet::new()` method provides yet another option for when you already have a collection of globs and want to build a set in one step.

See: Companion Code Section 1

---

## Section 2: Error Design Philosophy

The error handling in this module demonstrates thoughtful API design. The `Error` type carries both the original glob pattern (when available) and a categorized `ErrorKind`. This dual information allows error messages to be both specific and actionable—users see exactly which pattern failed and why.

The `ErrorKind` enum uses `#[non_exhaustive]`, which is worth pausing on. This attribute tells downstream code that the enum might gain new variants in future versions. Code matching on `ErrorKind` must include a wildcard arm, which enables the library to evolve without breaking compatibility. Two variants are marked as deprecated (`InvalidRecursive` and `NestedAlternates`), showing how the glob syntax has become more permissive over time while maintaining backward compatibility.

The error messages themselves are crafted for helpfulness. Look at `UnopenedAlternates`: the message not only explains the problem but suggests a solution—"maybe escape '}' with '[}]'?" This kind of guidance transforms error handling from frustration into education.

See: Companion Code Section 2

---

## Section 3: The Candidate Abstraction

The `Candidate` struct represents a path prepared for matching. This is an **amortization pattern**—doing expensive work once and reusing the result across multiple operations. When matching a single path against many patterns, you construct one `Candidate` and pass it to each matcher.

A `Candidate` precomputes three views of the path: the full normalized path, the basename (filename without directory), and the file extension. These correspond directly to the optimization strategies we'll see later. A glob like `*.rs` only needs the extension, while `Makefile` (matching a literal basename) only needs the basename. By computing these once, the matching code can jump straight to the relevant portion.

The `Cow<'a, [u8]>` type (Copy-on-Write) enables efficient handling of both borrowed and owned data. When normalizing a path that's already in the correct form, the `Candidate` can borrow directly from the input. Only paths requiring normalization (like those with backslashes on Windows) allocate new storage. This is a performance pattern you'll see throughout ripgrep—avoiding allocation in the common case while handling edge cases correctly.

See: Companion Code Section 3

---

## Section 4: Strategy-Based Matching Architecture

The heart of this module is its strategy system. Rather than compiling every glob to a regular expression and running a regex engine, `GlobSet` categorizes patterns and routes each to its optimal matching strategy. The `GlobSetMatchStrategy` enum captures seven distinct strategies:

**Literal** matches exact paths like `src/main.rs`. A hash table lookup tells you instantly if the path matches—O(1) regardless of how many literal patterns exist.

**BasenameLiteral** handles patterns like `Makefile` that should match any file with that exact name, regardless of directory. Again, a hash table lookup on just the basename.

**Extension** covers the extremely common `*.ext` pattern. Another hash table, keyed by extension.

**Prefix** and **Suffix** handle patterns like `src/**` (everything under src) and `**/*.rs` (all Rust files). These use the Aho-Corasick algorithm for efficient multi-pattern matching.

**RequiredExtension** is a hybrid: patterns that require a specific extension but need regex for the rest. The extension lookup narrows candidates before the more expensive regex runs.

**Regex** is the fallback for patterns too complex for specialized handling.

This stratification means that adding a hundred `*.ext` patterns to a glob set doesn't slow down matching—they all go into one hash table. Only genuinely complex patterns pay the regex cost.

See: Companion Code Section 4

---

## Section 5: The Multi-Pattern Matching Engine

The `GlobSet` matching methods reveal a carefully designed API surface. There are two axes of variation: what you want to know (is there any match? which patterns match?) and whether you're

providing a raw path or a pre-built `Candidate`.

The `is_match` family short-circuits on the first match—if you only need a boolean answer, there's no reason to check all patterns. The `matches` family returns all matching pattern indices, maintaining the original insertion order. The `matches_into` variant lets callers provide a pre-allocated vector, avoiding allocation when matching many paths.

The `matches_candidate_into` method shows the sorting and deduplication step. Because different strategies might report the same pattern index (a pattern might qualify for multiple strategies), the results need deduplication. The sort ensures consistent, ascending order regardless of which strategies fired.

The `matches_all` and `matches_all_candidate` methods check if every pattern matches—useful for implementing "must match all" semantics. Notice the edge case handling: an empty set matches all patterns because vacuously all zero patterns match.

See: Companion Code Section 5

---

## Section 6: Hash-Based Strategies

The `LiteralStrategy`, `BasenameLiteralStrategy`, and `ExtensionStrategy` all follow the same pattern: a hash map from byte sequences to vectors of pattern indices. The FNV (Fowler-Noll-Vo) hash function, implemented in the `fnv` module, is chosen for its speed on small keys—file extensions and basenames are typically short.

These strategies demonstrate **trading space for time**. Each strategy maintains its own copy of the relevant data, and patterns might appear in multiple strategies. The memory overhead is minimal compared to the matching speed gained.

The `#[inline(never)]` annotations on `matches_into` methods are intriguing. This is a hint to the compiler about code generation strategy. By preventing inlining, the function maintains a single copy that can be better optimized and keeps the instruction cache footprint smaller. It's a performance tuning detail that shows the level of care in this implementation.

See: Companion Code Section 6

## Section 7: Aho-Corasick for Prefix and Suffix Matching

The `PrefixStrategy` and `SuffixStrategy` use the Aho-Corasick algorithm, a classic string matching algorithm that finds all occurrences of multiple patterns in a single pass through the text. It's the same algorithm that powers tools like `grep -F` when searching for multiple literal strings.

For prefix matching, the strategy extracts the beginning of the path (up to the length of the longest pattern) and searches for any pattern that starts at position 0. For suffix matching, it extracts the end of the path and searches for any pattern that ends at the final position.

The `longest` field enables an optimization: if no prefix pattern is longer than 20 bytes, there's no need to examine more than 20 bytes of the path. The `path_prefix` and `path_suffix` methods on `Candidate` use this bound to create efficient slices.

The `MultiStrategyBuilder` abstracts the common construction logic. It collects literals, maintains the mapping from internal pattern indices to global indices, and tracks the longest pattern. The builder then transforms into either a `PrefixStrategy` or `SuffixStrategy` depending on context.

See: Companion Code Section 7

---

## Section 8: The Required Extension Hybrid Strategy

The `RequiredExtensionStrategy` represents an elegant middle ground. Consider a pattern like `src/**/*.rs`—it requires the `.rs` extension but needs regex power for the `src/**/` part. Compiling this to pure regex would work, but we can do better.

This strategy uses a two-phase approach: first, check if the candidate's extension exists in our hash map. If not, the pattern can't possibly match—no regex needed. Only if the extension matches do we run the regex against the full path.

The builder stores regex patterns as strings, deferring compilation until `build()` is called. This allows the `RequiredExtensionStrategyBuilder` to be `Clone` (strings are cheap to clone, compiled regexes are not) and batches the expensive compilation work.

See: Companion Code Section 8

---

## Section 9: Regex Set Strategy and Memory Pooling

When patterns resist all optimizations, they fall through to `RegexSetStrategy`. This uses `regex-automata`'s multi-pattern matching capability to check all remaining patterns simultaneously.

The `PatternSet` pool deserves special attention. A `PatternSet` is a bit vector tracking which patterns matched—it must have exactly as many bits as there are patterns in the regex set. Creating a fresh `PatternSet` for each match operation would mean frequent allocations.

The pool provides a thread-safe cache of pre-allocated `PatternSet` instances. A thread borrows one from the pool, uses it, clears it, and returns it. The `Arc<Pool<...>>` enables `RegexSetStrategy` to be `Clone` (multiple strategy instances share the same pool) without sacrificing thread safety.

The comment about the "next semver breaking release" shows thoughtful API evolution planning. The current pooling is internal and automatic; a future version might expose this to callers for even better amortization control.

See: Companion Code Section 9

---

## Section 10: Regex Configuration and Safety Limits

The `new_regex` and `new_regex_set` functions configure the regex engine with careful attention to both correctness and resource limits. The `utf8(false)` setting is crucial—paths on Unix can contain arbitrary bytes, not just valid UTF-8. The regex engine must handle this reality.

The size limits (`nfa_size_limit`, `hybrid_cache_capacity`) protect against malicious or pathological patterns. Without limits, a carefully crafted glob could consume unbounded memory during compilation or matching. The 10 MiB limits are generous enough for any reasonable use case while providing protection.

The `match_kind(MatchKind::All)` setting for regex sets ensures all matching patterns are reported, not just the first or leftmost-longest. This is essential for the glob set semantics where we want every matching pattern's index.

See: Companion Code Section 10

---

## Section 11: The Escape Function

The `escape` function provides a way to treat arbitrary strings as literal glob patterns. Metacharacters like `*`, `?`, and brackets are wrapped in character classes: `*` becomes `[*]`, which matches a literal asterisk.

This is essential for programmatic glob construction. If you're building a glob from user input that should match literally (like searching for a file named `important*.txt`), escaping ensures the `*` doesn't become a wildcard.

The implementation is straightforward but handles a subtle edge case: `!` doesn't need escaping because it's only special as the first character inside brackets (`[!abc]` means "not a, b, or c"). Outside that context, `!` is just a literal exclamation mark.

See: Companion Code Section 11

---

## Key Takeaways

First, strategy-based optimization transforms potentially slow operations into fast ones by recognizing common patterns. The same glob set that might compile to expensive regex patterns instead becomes a collection of hash lookups and Aho-Corasick searches.

Second, the `Candidate` abstraction demonstrates the amortization principle—pay a small cost once to avoid paying repeatedly. When matching many patterns against one path, this preprocessing pays dividends.

Third, the Builder pattern enables both flexible construction and aggressive optimization. The separation between building and using allows the implementation to analyze all patterns together and construct optimal data structures.

Fourth, thoughtful error design includes not just what went wrong but helpful guidance for fixing it. The `#[non_exhaustive]` attribute enables safe API evolution.

Fifth, resource limits on regex compilation protect against denial-of-service through pathological patterns. Production code must consider adversarial inputs.

Sixth, the pool pattern for `PatternSet` shows how to amortize allocation costs in hot paths while maintaining thread safety through careful use of `Arc` and synchronization primitives.

## What to Read Next

How do individual glob patterns get parsed and compiled? Read `crates/globset/src/glob.rs` to see the `Glob`, `GlobBuilder`, and `MatchStrategy` types.

How does ripgrep use glob sets for file filtering? Read `crates/ignore/src/gitignore.rs` to see how gitignore files are parsed into glob sets.

What path utilities support this module? Read `crates/globset/src/pathutil.rs` for the `normalize_path`, `file_name`, and `file_name_ext` functions.

How does the FNV hash function work? Read `crates/globset/src/fnv.rs` for a compact, performance-oriented hash implementation.

# ripgrep crates/globset/src/glob.rs: Glob Implementation

## What This File Does

This file implements the core glob pattern parser and compiler that transforms shell-style patterns like `*.rs` or `**/src/**/*.c` into executable matchers. While the previous lesson covered `lib.rs` and how multiple globs get combined into efficient sets, this file focuses on the mechanics of individual patterns: how they're parsed, what optimizations can be extracted from them, and how they become regular expressions.

The architecture here is deceptively sophisticated. A simple pattern like `**/*.rs` doesn't just become a regex—the code analyzes its structure to determine the fastest matching strategy. Can we match by extension alone? By prefix? By suffix? These optimizations matter enormously when scanning millions of files, and they're all computed by examining the parsed token stream that this file produces.

## Section 1: The Match Strategy Optimization

The most interesting design decision in this file appears right at the top: the `MatchStrategy` enum. Rather than treating every glob pattern as a regex that needs full pattern matching, the code identifies patterns that can be matched with simpler, faster operations.

Consider what happens when ripgrep encounters a `.gitignore` containing `*.txt`. That pattern could be handled by compiling it to a regex and running the regex engine on every file path. But there's a much faster approach: extract the extension `.txt` and perform a simple byte comparison against each file's extension. No regex engine needed, no backtracking, just a quick equality check.

The `MatchStrategy` enum captures seven different optimization categories. A `Literal` strategy means the entire path must match exactly—no wildcards at all. `BasenameLiteral` means we only need to check the filename portion. `Extension` is the `.txt` case. `Prefix` handles patterns that start with a literal path. `Suffix` handles patterns that end with a literal string. `RequiredExtension` is a partial optimization—we can quickly reject non-matching extensions before falling back to a full regex match.

This tiered approach reflects a key insight about real-world usage. Most ignore patterns in actual `.gitignore` files fall into simple categories. Optimizing for the common case while falling back to full regex power for complex patterns gives ripgrep both speed and correctness.

See: Companion Code Section 1

---

## Section 2: The Glob Type and Builder Pattern

The `Glob` struct represents a successfully parsed pattern, and it follows a pattern we've seen throughout ripgrep: it stores both the original representation and the compiled form. The struct holds the original glob string, the generated regex string, the parsing options, and the internal token representation.

What's interesting here is the trait implementations. The `PartialEq` implementation only compares the glob string and options—not the regex or tokens. This is semantically correct because two globs with the same pattern and options should be equal regardless of their internal representation. The `Hash` implementation follows the same logic, which is essential for using globs as hash map keys.

The `Debug` implementation shows thoughtful attention to ergonomics. In normal debug mode, it shows just the glob pattern wrapped in a tuple struct format. But with alternate formatting via `{:#?}`, it expands to show all internal fields including the generated regex and tokens. This is useful during development when you need to see what the parser actually produced.

The `GlobBuilder` type follows the builder pattern we've examined in previous lessons on `searcher/mod.rs` and `printer/standard.rs`. It holds a reference to the pattern string and accumulates options before parsing. The lifetime `'a` is tied to the input pattern, which means the builder borrows rather than owns the pattern text. This is efficient for the common case where you parse a pattern immediately.

See: Companion Code Section 2

## Section 3: The Token Representation

Before any matching can happen, a glob pattern must be parsed into a structured representation. The `Token` enum captures all the possible elements that can appear in a glob pattern.

The basic tokens are straightforward: `Literal` holds a single character, `Any` represents `?` (matches one character), and `ZeroOrMore` represents `*` (matches zero or more characters). But the recursive tokens are where things get interesting. `RecursivePrefix` represents `**/` at the start of a pattern—it can match any number of directory components. `RecursiveSuffix` is `/**` at the end. `RecursiveZeroOrMore` is `/**/` in the middle, which must match at least one slash.

The distinction between these recursive variants matters for optimization. A pattern starting with `**/` only needs to match the basename of files, while `/**/` requires at least one path separator. Getting these semantics wrong would cause incorrect matching behavior.

Character classes become `Class` tokens containing a negation flag and a vector of character ranges. The parser normalizes syntax like `[a-z0-9]` into a vector of tuples representing ranges. Alternates like `{foo,bar}` become nested `Tokens` structures, allowing for arbitrary nesting depth.

The `Tokens` wrapper type implements `Deref` and `DerefMut` to `Vec<Token>`, making it easy to work with as a vector while maintaining a distinct type for clarity.

See: Companion Code Section 3

---

## Section 4: Strategy Extraction Methods

The `Glob` implementation includes a series of private methods that attempt to extract optimized matching strategies from the token stream. Each method returns `Option<T>`, returning `None` if the pattern is too complex for that optimization.

The `literal` method checks if the entire pattern consists only of literal characters. If case-insensitive matching is enabled, this optimization isn't available—case folding requires regex machinery. The method iterates through tokens, bailing out if it finds anything other than literals.

The `ext` method is more complex. It looks for patterns of the form `*.ext` or `**/*.ext`. The logic must account for several edge cases: the pattern might start with `**` (recursive prefix), the `*` must be able to match path separators (otherwise `*.c` wouldn't match `foo/bar.c`), and the extension itself can't contain dots or slashes.

The `prefix` and `suffix` methods extract literal strings that must appear at the start or end of matching paths. These are particularly useful for patterns like `build/**` or `**/*.test.js`. The suffix method returns a tuple containing both the suffix string and a boolean indicating whether the suffix must match the entire path or can appear after a separator.

The `basename_tokens` method is crucial for the common `**/filename` pattern. When a pattern starts with `**/` and the rest doesn't contain path separators, we only need to match against the file's basename. This method returns a slice of the original tokens rather than allocating a new vector—a small but meaningful efficiency.

See: Companion Code Section 4

---

## Section 5: The Parser Structure

The `Parser` struct manages state during glob parsing. Its design reveals the complexity hidden behind seemingly simple glob syntax.

The parser tracks alternation nesting through two fields: `alternates_stack` marks indices where alternation groups begin, and `branches` holds the token sequences for each branch. When the parser encounters `{`, it pushes the current branch count onto the stack and starts a new branch. When it sees `,`, it starts another branch at the same nesting level. When it hits `}`, it pops the stack and combines all branches since that point into an `Alternates` token.

The `prev` and `cur` fields track the previous and current characters. This two-character lookahead is essential for parsing `**`—the parser needs to know if a `*` is followed by another `*` to determine whether it's a recursive wildcard.

The `found_unclosed_class` field handles a subtle compatibility concern. Some glob implementations treat an unclosed `[` as a literal character rather than an error. When `allow_unclosed_class` is enabled and we fail to find a closing `]`, we set this flag and never attempt to parse character classes again. Without this optimization, a pattern like `[[[[[[[...` would cause quadratic parsing time as we repeatedly scan forward looking for a `]` that doesn't exist.

See: Companion Code Section 5

---

## Section 6: Parsing Special Sequences

The `parse_star` method handles the most complex parsing in the file. A single `*` is straightforward, but `**` has special semantics depending on context.

The method first checks if the next character is also `*`. If not, it simply pushes a `ZeroOrMore` token. But when we have `**`, the interpretation depends on position. At the start of a pattern (or

after a separator), `**` followed by a separator becomes `RecursivePrefix`. At the end of a pattern, `**/` becomes `RecursiveSuffix`. In the middle with separators on both sides, it becomes `RecursiveZeroOrMore`.

The edge cases are subtle. What about `**` at the start followed by a literal? It becomes two `ZeroOrMore` tokens, not a recursive pattern. What about `a**b`? Also two `ZeroOrMore` tokens—the recursive patterns only apply with proper separator context.

The `parse_class` method handles character classes with their own complexity. It tracks whether we're building a range (`a-z`) or collecting individual characters. The `]` and `-` characters have special meaning at the start of a class—`[]]` matches a literal `]`, and `[-a]` includes a literal `-`. The method also implements the recovery logic for unclosed classes when that option is enabled.

See: Companion Code Section 6

---

## Section 7: Regex Generation

Once parsing completes, the token stream must become a regular expression. The `to_regex_with` method on `Tokens` performs this translation.

The regex always starts with `(?-u)` to disable Unicode mode, since file paths are arbitrary bytes that may not be valid UTF-8. If case-insensitive matching is requested, `(?i)` follows. Then comes `^` to anchor at the start.

Each token type has a direct regex translation. `Literal` characters get escaped and inserted. `Any` becomes `.` (or `[^/]` if literal_separator is enabled). `ZeroOrMore` becomes `.*` (or `[^/]*`). The recursive patterns use non-capturing groups with alternation: `RecursivePrefix` becomes `(?:/?|.*/)` to match either nothing, a slash, or any path prefix.

The `char_to_escaped_literal` function handles the byte-level complexity of non-ASCII characters. Since the regex operates on bytes, multi-byte UTF-8 characters must be encoded as their individual bytes with proper escaping. A snowman character ☃ becomes `\xe2\x98\x83` in the generated regex.

Character classes translate fairly directly, with each range becoming a regex character class. Alternates become non-capturing groups with `|` separators.

See: Companion Code Section 7

---

## Section 8: The GlobBuilder Options

The builder pattern for `Glob` exposes five configuration options, each affecting parsing and matching semantics in specific ways.

The `case_insensitive` option adds `(?i)` to the generated regex, enabling case-insensitive matching. But it also disables all the extraction optimizations—when matching is case-insensitive, we can't do simple byte comparisons for extensions or literals.

The `literal_separator` option determines whether `*` and `?` can match path separators. By default they can, which matches traditional glob behavior. But for strict path matching, enabling this option makes `*.c` only match files in the current directory, not `foo/bar.c`. This affects both regex generation (using `[^/]` instead of `.`) and which optimizations are available.

The `backslash_escape` option controls whether `\` escapes special characters. On Unix, this is enabled by default—`\*` matches a literal asterisk. On Windows, where `\` is a path separator, it's disabled by default. The implementation checks `is_separator('\\')` to determine the platform-appropriate default.

The `empty_alternates` option allows patterns like `{,a}` to match both empty string and `a`. By default, empty branches in alternates are filtered out during regex generation.

The `allow_unclosed_class` option enables compatibility mode where `[` without a matching `]` is treated as a literal rather than an error. This is useful for consuming patterns from sources that may have different parsing rules.

See: Companion Code Section 8

---

## Section 9: The Matcher Types

The `GlobMatcher` struct wraps a compiled pattern for actual matching. It holds both the original `Glob` and a compiled `Regex` from the `regex_automata` crate.

The matching interface is simple: `is_match` takes anything that can be converted to a `Path`, and `is_match_candidate` takes a pre-computed `Candidate`. The `Candidate` type from `lib.rs` caches the path's bytes, basename, and extension, avoiding repeated computation when matching against multiple patterns.

For testing purposes, there's also `GlobStrategic`, which implements the optimized matching strategies. It's gated behind `#[cfg(test)]` because the public API always uses the regex matcher—the strategic matching is used in `GlobSet` from `lib.rs`, not in single-pattern matching. The test implementation ensures that strategic matching produces the same results as regex matching for all test cases.

The strategic matcher's `is_match_candidate` method shows the optimization logic in action. For `Literal` strategy, it's a simple byte comparison. For `Extension`, it compares the cached extension bytes. For `Prefix` and `Suffix`, it checks the appropriate ends of the path. For `RequiredExtension`, it checks the extension first as a quick rejection, then falls back to the regex.

See: Companion Code Section 9

---

## Section 10: The Test Suite Philosophy

The test suite at the end of the file demonstrates a philosophy of exhaustive edge-case coverage. The macros define DSLs for different types of tests: syntax tests verify token parsing, error tests verify parse failures, regex tests verify generated patterns, and match tests verify actual matching behavior.

The syntax tests verify that patterns parse to expected token sequences. This is valuable for catching parser regressions and documenting expected behavior for edge cases like `[-a-z-]` (a dash at the start and end of a character class).

The match tests use both the standard regex matcher and the strategic matcher, ensuring they agree. They also test through a `GlobSet` to verify the optimization logic in the set builder.

The extraction tests verify the strategy detection methods. These are particularly important because incorrect strategy extraction leads to subtle matching bugs—patterns might match correctly in tests but fail on edge cases in the wild.

The options constants (`CASEI`, `SLASHLIT`, `BSESC`, etc.) allow tests to easily specify non-default configurations while keeping test definitions concise.

See: Companion Code Section 10

## Key Takeaways

First, glob patterns benefit enormously from preprocessing optimizations. By analyzing pattern structure at compile time, ripgrep can often avoid regex matching entirely, using simple string operations instead.

Second, the builder pattern with method chaining provides a clean API for optional configuration while keeping the common case—default options—simple and efficient.

Third, cross-platform compatibility requires careful attention to path separator semantics. The `backslash_escape` option with its platform-dependent default shows how to handle behavioral differences between Unix and Windows.

Fourth, error recovery options like `allow_unclosed_class` should have conservative defaults but be available when compatibility matters more than strict parsing.

Fifth, testing strategy matters as much as implementation. Running both the optimized path and the fallback path in tests ensures they stay synchronized as the code evolves.

Sixth, the separation between parsing, optimization analysis, and regex generation keeps each concern focused and testable. The token representation serves as a clean intermediate form that all three phases can work with.

---

## What to Read Next

How does the glob set combine multiple patterns efficiently? Return to `crates/globset/src/lib.rs` to see how `MatchStrategy` feeds into the multi-pattern optimization.

How does the ignore crate use these glob patterns? Read `crates/ignore/src/gitignore.rs` to see how `.gitignore` semantics layer on top of glob matching.

What other builder patterns exist in ripgrep? Compare with `crates/searcher/src/searcher/mod.rs` to see how the same pattern applies to search configuration.

# ripgrep crates/ignore/src/lib.rs: Ignore Crate Overview

## What This File Does

The `ignore` crate serves as ripgrep's intelligent directory traversal engine, combining fast recursive file walking with sophisticated filtering based on `.gitignore` files, custom ignore patterns, and file type definitions. This `lib.rs` file establishes the crate's public interface, defines its comprehensive error handling system, and introduces the `Match` abstraction that powers pattern matching decisions throughout the crate.

Think of this crate as the gatekeeper that decides which files ripgrep should examine. Before any searching happens, the ignore crate must traverse directories, read ignore files, apply glob patterns, and determine whether each file should be processed or skipped. This file lays the foundation for all those decisions by exporting the key types and defining how errors and match results are represented.

## Section 1: The Crate's Public Interface

The module documentation at the top of this file serves as both a user guide and an architectural overview. Notice how it immediately establishes the crate's dual purpose: providing a fast recursive directory iterator while also exposing lower-level components like gitignore and file type matchers for fine-grained control.

The two code examples in the documentation demonstrate the builder pattern that you've seen throughout ripgrep. The simple `Walk::new("./")` approach provides sensible defaults, while `WalkBuilder::new("./").hidden(false).build()` shows how to customize behavior. This pattern connects directly to the builder ecosystem you've studied in files like `searcher/mod.rs` and `printer/standard.rs`. The ignore crate follows the same philosophy: make the common case easy while allowing complete customization.

The `pub use` statement re-exports key types from the internal `walk` module, making `Walk`, `WalkBuilder`, `WalkParallel`, and related types available at the crate root. This is a common Rust pattern for organizing code internally while presenting a clean external interface. Users can simply write `ignore::Walk` rather than `ignore::walk::Walk`.

See: Companion Code Section 1

---

## Section 2: Module Organization

The module declarations reveal the crate's internal architecture. Each module handles a specific responsibility, and the visibility modifiers tell you which are considered public API versus internal implementation details.

The `pub mod` declarations for `gitignore`, `overrides`, and `types` indicate that these are stable public interfaces that users can import directly. These modules provide the fine-grained control mentioned in the crate documentation. In contrast, modules like `default_types`, `dir`, `pathutil`, and `walk` are private—they're implementation details that might change between versions.

This organization reflects a key Rust library design principle: expose the minimum necessary surface area while keeping implementation flexibility. The `walk` module is private even though its key types are re-exported at the crate level. This means the crate authors can reorganize the internal structure without breaking downstream code, as long as the re-exported types maintain their behavior.

See: Companion Code Section 2

## Section 3: The Error Enum Architecture

The `Error` enum demonstrates a sophisticated approach to error handling that goes beyond simple error messages. Each variant captures context about where and why an error occurred, allowing errors to be wrapped with additional information as they propagate up the call stack.

The design uses wrapper variants like `WithLineNumber`, `WithPath`, and `WithDepth` to add context without losing the original error. This creates a nested structure where an error might be an `Error::WithPath` containing an `Error::WithLineNumber` containing an `Error::Glob`. When you display such an error, you get rich context like "path/to/.gitignore: line 42: invalid glob pattern".

The `Partial` variant deserves special attention. It holds a `Vec<Error>`, representing situations where some operations succeeded while others failed. This is particularly important when parsing gitignore files—you don't want a single malformed pattern to prevent all the valid patterns from being used. The crate can report these "soft" errors while continuing to function.

See: Companion Code Section 3

## Section 4: Implementing Clone for Error

The manual `Clone` implementation for `Error` reveals an interesting challenge: `std::io::Error` doesn't implement `Clone`. This is a common pain point in Rust—IO errors contain internal state that the standard library authors decided shouldn't be trivially copyable.

The solution here is pragmatic. For IO errors with a raw OS error code (like ENOENT or EACCES), the code extracts that code and creates a new error from it. For other IO errors, it creates a new error with the same kind and a string representation of the original. This loses some information—the original error's source chain, for example—but it's good enough for the error reporting use cases in this crate.

This pattern demonstrates a broader Rust theme: sometimes you need to work around limitations in the standard library or in types you don't control. The key is to understand what information you actually need to preserve and what can be safely approximated.

See: Companion Code Section 4

---

## Section 5: Error Inspection Methods

The `Error` type provides several methods for inspecting and classifying errors. The `is_partial()` and `is_io()` methods are particularly important because they allow callers to make decisions based on error severity.

Notice how these methods recursively unwrap the context layers. An `Error::WithPath` containing an `Error::Io` should still return `true` for `is_io()`. This design means the context wrappers are truly transparent—adding path or line number information doesn't change the fundamental nature of the error.

The `io_error()` and `into_io_error()` methods provide access to the underlying `std::io::Error` when present. The pair of methods—one borrowing, one consuming—follows the Rust pattern of providing both options so callers can choose based on their needs. If you just need to check the error kind, use `io_error()`. If you want to return the IO error to a caller, use `into_io_error()` to avoid cloning.

See: Companion Code Section 5

---

## Section 6: Error Context Builders

The `with_path()`, `with_depth()`, and `tagged()` methods are internal helpers for constructing richly-contextualized errors. These methods take `self` by value and return a wrapped version, enabling a fluent style of error construction.

The `tagged()` method is particularly clever. It adds both line number and path information, but it handles the edge case where the path might be empty. In that case, it returns just the line-numbered error without a path wrapper. This small detail prevents confusing error messages that would otherwise show an empty path.

The `from_walkdir()` method converts errors from the `walkdir` crate into the ignore crate's error type. This is a common pattern when wrapping external dependencies—you translate their error types into your own to maintain a consistent error interface. Notice how it extracts loop information specially, recognizing that filesystem loops are a distinct kind of error that shouldn't be wrapped as generic IO failures.

See: Companion Code Section 6

---

## Section 7: Standard Trait Implementations

The implementations of `std::error::Error` and `std::fmt::Display` make the error type work seamlessly with Rust's error handling ecosystem. The `Display` implementation is particularly important because it's what users see when errors are printed.

The display formatting handles each variant appropriately. Partial errors join their messages with newlines. Line-numbered errors prepend the line number. Path errors prepend the path. This recursive formatting means a deeply nested error like `WithPath { path, WithLineNumber { line, Glob { err } } }` automatically produces output like `"path: line 42: glob error"`.

The `From<std::io::Error>` implementation enables using the `?` operator to automatically convert IO errors into this crate's error type. This is a standard Rust ergonomics pattern that makes error propagation more pleasant to write.

See: Companion Code Section 7

## Section 8: The PartialErrorBuilder

The `PartialErrorBuilder` is an internal helper for accumulating errors that might occur during operations that can partially succeed. It's a simple but effective pattern for collecting errors without immediately failing.

The `push_ignore_io()` and `maybe_push_ignore_io()` methods reveal an important policy decision: sometimes IO errors should be silently ignored. This typically happens when reading optional ignore files—if a `.gitignore` file is unreadable due to permissions, the directory walk can continue with other ignore sources.

The `into_error_option()` method shows thoughtful API design. If no errors occurred, return `None`. If exactly one error occurred, return it directly rather than wrapping it in a `Partial` variant. Only create a `Partial` when there are genuinely multiple errors. This means callers get the simplest possible error representation.

See: Companion Code Section 8

---

## Section 9: The Match Enum

The `Match<T>` enum is a fundamental abstraction that appears throughout the ignore crate. It represents the result of checking a path against a set of glob patterns, with three possible outcomes: no match, ignore, or whitelist.

The generic parameter `T` allows different matchers to attach different metadata to their results. A gitignore matcher might attach information about which gitignore file and which line caused the match. A file type matcher might attach the type name. This flexibility means the same `Match` type works across all the different filtering systems in the crate.

The distinction between `Ignore` and `Whitelist` is crucial for implementing gitignore semantics correctly. In gitignore files, patterns starting with `!` negate previous ignore rules. The `Match::Whitelist` variant represents these negation patterns, allowing the matching logic to correctly implement precedence rules where later patterns override earlier ones.

See: Companion Code Section 9

## Section 10: Match Operations

The methods on `Match<T>` provide a combinator-style interface for working with match results. The `invert()` method swaps ignore and whitelist states, which is useful when applying negation

patterns. The `map()` method allows transforming the attached metadata without changing the match outcome.

The `or()` method implements a fallback pattern: if this match is `None`, try the other one. This is essential for implementing the precedence rules in gitignore matching, where you might check patterns from most-specific to least-specific and take the first match you find.

Notice how these methods mirror patterns from `Option` and `Result`. Rust programmers will find them immediately familiar. This isn't accidental—designing types to follow established patterns reduces cognitive load and makes code more predictable.

See: Companion Code Section 10

---

## Section 11: Testing Infrastructure

The test module demonstrates pragmatic testing infrastructure. Rather than pulling in a heavyweight testing dependency like `tempfile`, the code includes a minimal `TempDir` implementation that creates and cleans up temporary directories.

The `TempDir` implementation uses an atomic counter to generate unique directory names, with a retry loop to handle potential race conditions. The `Drop` implementation ensures cleanup even if tests panic. This pattern of building minimal infrastructure rather than adding dependencies is characteristic of performance-sensitive projects like ripgrep.

The `err!` macro provides a convenient way to create boxed error types for test assertions. It uses the `Box<dyn std::error::Error + Send + Sync>` pattern that's common in Rust testing, allowing any error type to be returned from test functions.

See: Companion Code Section 11

---

## Key Takeaways

First, the ignore crate follows ripgrep's builder pattern philosophy, providing simple defaults via `Walk::new()` while enabling deep customization through `WalkBuilder`. This connects to the broader builder ecosystem you've studied in other crates.

Second, the `Error` enum demonstrates sophisticated error handling where context can be layered onto errors as they propagate. The wrapper variants like `WithPath` and `WithLineNumber` add information without losing the original error.

Third, the `Partial` error variant enables graceful degradation—operations can partially succeed and report their failures without completely aborting. This is essential for robust gitignore processing.

Fourth, implementing `Clone` for error types that contain `std::io::Error` requires workarounds since IO errors aren't cloneable. The solution here reconstructs errors from their OS error codes when possible.

Fifth, the `Match<T>` enum provides a generic, combinable abstraction for pattern matching results that works across different filtering systems. The ignore/whitelist distinction correctly models gitignore negation semantics.

Sixth, minimal testing infrastructure like the custom `TempDir` reflects ripgrep's philosophy of controlling dependencies, even in test code.

---

## What to Read Next

How does the directory walker actually traverse directories? Read `crates/ignore/src/walk.rs` to see the `Walk` and `WalkBuilder` implementations in detail.

How are gitignore files parsed and matched? Read `crates/ignore/src/gitignore.rs` to understand the gitignore pattern matching system.

How do file type definitions work? Read `crates/ignore/src/types.rs` to see how ripgrep associates globs with named file types.

How do override patterns work? Read `crates/ignore/src/overrides.rs` to understand user-specified patterns that take precedence over gitignore rules.

# ripgrep crates/ignore/src/gitignore.rs: Gitignore Parsing

## What This File Does

This file implements a complete gitignore parser and matcher from scratch, without shelling out to the `git` command line tool. It transforms the somewhat quirky syntax defined in the `gitignore` man page into the glob matching infrastructure we explored in the previous lesson. The result is a system that can read `.gitignore` files, parse their patterns according to git's exact semantics, and efficiently determine whether any given file path should be ignored.

The implementation follows the builder pattern we've seen throughout ripgrep's codebase. A `GitignoreBuilder` accumulates glob patterns from one or more gitignore files, then constructs an immutable `Gitignore` matcher. This separation allows for flexible configuration—you can combine patterns from multiple sources, control case sensitivity, and handle parse errors gracefully without abandoning valid patterns.

## Section 1: The Glob Data Structure

The heart of gitignore matching lies in understanding what a pattern actually means. Git's gitignore syntax carries more semantic information than a simple glob string—a pattern might be a whitelist entry (prefixed with `!`), might only match directories (suffixed with `/`), or might have come from a specific file path that affects how errors are reported. The `Glob` struct captures all of this metadata alongside both the original pattern text and the transformed pattern that will actually be used for matching.

The distinction between `original` and `actual` is subtle but important. When you write `foo/` in a gitignore file, you're saying "ignore directories named foo." But the trailing slash isn't part of what gets matched—it's a semantic modifier. Similarly, patterns without path separators need a `**/` prefix to match anywhere in the directory tree, not just at the root. The `Glob` struct preserves both forms so you can see exactly what the user wrote while also having the correct matching pattern.

This design reflects a broader principle in ripgrep: maintain fidelity to user input while transforming it internally for correctness. When ripgrep reports which pattern matched a file, it shows the

original pattern from the gitignore file, not the internally-transformed version. This makes error messages and diagnostics far more useful.

See: Companion Code Section 1

---

## Section 2: The Gitignore Matcher

The `Gitignore` struct is the compiled, immutable matcher that results from building patterns. It wraps a `GlobSet` from the globset crate—the same infrastructure we studied in the previous lesson—along with metadata about the patterns it contains. This layered design means the gitignore module doesn't need to implement its own matching engine; it transforms gitignore semantics into glob semantics and delegates the heavy lifting.

Notice the `matches` field, which holds an `Option<Arc<Pool<Vec<usize>>>>`. This is the object pool pattern we've seen in ripgrep's searcher and printer infrastructure. When matching, the `GlobSet` needs a scratch vector to store which patterns matched. Rather than allocating a new vector for every match operation, the pool provides thread-safe reuse of these scratch buffers. The `Arc` wrapper allows multiple threads to share the same pool.

The distinction between `num_ignores` and `num_whitelists` matters for optimization. If a gitignore file contains only whitelist patterns (only `!` prefixed patterns), it can never cause a file to be ignored—it can only un-ignore files that were ignored by other patterns. This metadata enables short-circuit logic in the broader ignore infrastructure.

See: Companion Code Section 2

---

## Section 3: Path Stripping and Relative Matching

One of the trickiest aspects of gitignore matching is handling the relationship between the gitignore file's location and the paths being matched. A pattern in `/home/user/project/.gitignore` applies to paths relative to `/home/user/project/`, not absolute paths. The `strip` method handles this normalization.

The implementation handles several edge cases that might not be obvious. A leading `./` is completely superfluous and gets stripped from both the root path and candidate paths. If the candidate path shares a common prefix with the gitignore's root, that prefix gets stripped so matching happens relative to the gitignore location. But there's a subtle case: if the path is just a

filename with no directory components, stripping could incorrectly remove part of the filename itself if it happens to match the root.

The special case for root being `.` is particularly interesting. When the gitignore's root is the current directory represented as `.`, we shouldn't try to strip it from paths that happen to start with a dot (like `.gitignore` itself). This kind of edge case handling is why the file implements gitignore semantics from scratch rather than trying to adapt a simpler glob matcher.

See: Companion Code Section 3

---

## Section 4: The Matching Algorithm

The `matched_stripped` method reveals how precedence works in gitignore files. When multiple patterns could match a path, the last one wins. This is why the method iterates through matches in reverse order—it's looking for the highest-precedence (last-defined) pattern that applies.

The `is_only_dir` check is crucial here. A pattern like `foo/` should only match directories, but the `GlobSet` doesn't know whether we're matching a directory or a file—that's semantic information from the filesystem, not from the pattern. So the matcher checks this constraint after the glob matching succeeds, allowing the pattern to match only if the path is actually a directory.

The return type uses the `Match` enum from the ignore crate's public interface. This tri-state result—`Ignore`, `Whitelist`, or `None`—lets callers distinguish between "this should be ignored," "this was explicitly un-ignored," and "no pattern matched." The whitelist case is important because it affects how patterns from multiple gitignore files interact: a whitelist in a child directory can un-ignore something that was ignored by a parent directory's patterns.

See: Companion Code Section 4

---

## Section 5: The Builder Pattern

The `GitignoreBuilder` follows the same builder pattern we've seen throughout ripgrep. It accumulates configuration and pattern data, then produces an immutable `Gitignore` through its `build` method. The builder owns a `GlobSetBuilder` internally, maintaining the layered abstraction—gitignore semantics on top of glob semantics.

Two configuration options deserve attention. The `case_insensitive` flag affects how globs are matched, which is useful on case-insensitive filesystems like Windows or macOS with default settings. The `allow_unclosed_class` option defaults to `true` to match git's actual behavior—if

you write `[abc` without a closing bracket, git treats it as a literal string rather than an error. This kind of compatibility concern is why implementing gitignore correctly requires careful attention to the specification's edge cases.

The builder also strips the `./` prefix from the root path during construction. This normalization happens once at builder creation rather than on every match operation, which is more efficient and ensures consistent behavior regardless of how the path was originally specified.

See: Companion Code Section 5

---

## Section 6: Parsing Individual Lines

The `add_line` method is where gitignore syntax gets transformed into glob patterns. This is one of the more complex methods in the file because gitignore syntax has many special cases that need to be handled in the right order.

The method first handles comments (lines starting with `#`) and blank lines, which are skipped. Trailing whitespace is trimmed unless it's escaped with a backslash. Then comes the escape handling for patterns that actually start with `!` or `#`—you can write `\!foo` to match a file literally named `!foo`.

The whitelist prefix `!` and the absolute prefix `/` are processed next. A leading slash means the pattern should only match at the root of the gitignore's scope, not anywhere in the tree. This is implemented by *not* adding the `**/` prefix that would otherwise allow matching anywhere.

The trailing slash for directory-only matching is handled, with special attention to escaped slashes. Then comes the key transformation: patterns without a slash anywhere in them get a `**/` prefix so they match at any depth. Finally, patterns ending with `/**` get an extra `/*` appended so they match contents inside the directory rather than the directory itself.

See: Companion Code Section 6

---

## Section 7: Reading Gitignore Files

The `add` method handles reading patterns from a gitignore file on disk. It returns an `Option<Error>` rather than a `Result` because partial errors are possible—one invalid pattern shouldn't prevent other valid patterns from being added.

The line-by-line reading handles a subtle Unicode case: git supports files that begin with the UTF-8 byte order mark (BOM). This invisible character at the start of a file is stripped if present on the first line. This kind of detail shows the commitment to matching git's actual behavior, not just a simplified version of it.

Error handling is tagged with both the file path and line number, which enables useful error messages that tell users exactly where a problem pattern was defined. The `PartialErrorBuilder` collects these errors without stopping processing, allowing ripgrep to use as many valid patterns as possible even when some patterns are malformed.

See: Companion Code Section 7

---

## Section 8: Global Gitignore Discovery

Git supports a global gitignore file configured through `core.excludesFile` in the git configuration. The `build_global` method and its helper functions implement this discovery process, checking multiple locations in the right precedence order.

The search order follows git's conventions: first `$HOME/.gitconfig`, then `$XDG_CONFIG_HOME/git/config` (or `$HOME/.config/git/config` if the environment variable isn't set). If neither configuration file specifies an excludes file, the default location `$XDG_CONFIG_HOME/git/ignore` is used.

The `parse_excludes_file` function extracts the path from git's INI-style configuration format. It uses a regex pattern with case-insensitive matching and handles both quoted and unquoted values. The comment acknowledges this isn't a full INI parser, but it works for the common cases. Tilde expansion is also supported, so `~/foo/bar` in the config file correctly expands to the user's home directory.

See: Companion Code Section 8

---

## Section 9: Parent Directory Matching

The `matched_path_or_any_parents` method provides an interesting optimization for certain use cases. Sometimes you have a full path and need to know if it or any of its parent directories match a gitignore pattern. Rather than walking the directory tree yourself, this method handles it internally.

The method is documented as being more expensive than walking the tree top-to-bottom, which is the normal case during directory traversal. But when you have a list of paths without hierarchy information—perhaps from a different source like a file list—this method provides a convenient API.

The assertion that the path not have a root (not be absolute) protects against incorrect usage. After stripping the gitignore's root prefix, the remaining path should be relative. This is a debugging aid that catches programming errors during development.

See: Companion Code Section 9

---

## Section 10: The Constructor Methods

The `Gitignore::new` constructor provides a convenient entry point for the common case of loading a single gitignore file. It creates a builder, adds the file, handles errors gracefully, and returns both the matcher and any errors that occurred. The design always returns a valid matcher—even if it's empty due to errors—so callers can proceed with what worked.

The `Gitignore::global` constructor wraps the global gitignore discovery process. It handles the case where even getting the current working directory might fail, returning an empty matcher with the error rather than panicking.

The `Gitignore::empty` constructor creates a matcher that never matches anything. This is useful as a default value or as a fallback when gitignore functionality is disabled. The pattern of providing an `empty` constructor alongside `new` appears throughout ripgrep's codebase for types that might reasonably have a "do nothing" instance.

See: Companion Code Section 10

---

## Section 11: Test Macros and Verification

The test suite uses macros to define concise test cases for both positive and negative matching. The `ignored!` macro creates a test that asserts a pattern should match, while `not_ignored!` asserts the opposite. This table-driven style makes it easy to add new test cases as edge cases are discovered.

The test cases cover a remarkable range of gitignore behaviors: basic filename matching, wildcards, the `**` pattern for matching across directories, whitelist entries, directory-only patterns,

escaped special characters, and interaction with different root paths. Many test cases reference specific GitHub issues, indicating they were added to prevent regressions.

The configuration file parsing also has its own test cases, verifying that `core.excludesFile` is correctly extracted from git config content, including tilde expansion and quoted values.

See: Companion Code Section 11

---

## Key Takeaways

First, implementing gitignore correctly requires attention to many edge cases that aren't obvious from casual reading of the specification—escaped characters, trailing slashes, whitelist precedence, and directory-only matching all need special handling.

Second, the layered architecture separates concerns effectively: the globset crate handles pattern matching, while this module handles gitignore-specific syntax transformation and semantics.

Third, the builder pattern enables flexible construction from multiple sources while producing an immutable, thread-safe matcher optimized for repeated use.

Fourth, preserving both original and transformed pattern text enables better error messages and debugging without sacrificing matching correctness.

Fifth, object pooling through `Pool<Vec<usize>>` extends into the gitignore layer, maintaining ripgrep's commitment to minimizing allocations during hot paths.

Sixth, graceful error handling that collects partial errors rather than failing on the first problem lets ripgrep continue working with valid patterns even when some patterns are malformed.

---

## What to Read Next

How does the gitignore matcher integrate with directory traversal? Read `crates/ignore/src/walk.rs` to see how multiple gitignore files at different directory levels combine during a recursive search.

How are multiple ignore sources prioritized? Read `crates/ignore/src/dir.rs` to understand how `.gitignore`, `.ignore`, and command-line patterns interact.

Want to see the glob matching infrastructure this builds on? Return to `crates/globset/src/lib.rs` from the previous lesson for the underlying pattern compilation and

matching engine.

# ripgrep crates/ignore/src/types.rs: File Type Definitions

## What This File Does

This module implements ripgrep's file type filtering system—the mechanism that lets users say "search only Rust files" with `--type rust` or "exclude C files" with `--type-not c`. It transforms human-friendly type names like "rust" or "python" into efficient glob-based matchers that can quickly categorize thousands of files during a search.

The design is sophisticated: it allows multiple globs per file type (Rust files are `*.rs`, but C files include both `*.c` and `*.h`), supports type composition (defining a "web" type that includes HTML, CSS, and JavaScript), and handles the interaction between selections and negations. All of this builds on the `globset` crate you studied earlier, demonstrating how ripgrep's architecture creates specialized tools on top of general-purpose primitives.

## Section 1: The Builder Pattern for Type Matching

This file continues ripgrep's consistent use of the builder pattern, this time through `TypesBuilder` and `Types`. The builder accumulates file type definitions and user selections, then produces an immutable, optimized matcher. This pattern appears throughout the codebase: in `Searcher`, in the printers, and in the directory walker. Each instance adapts the pattern to domain-specific needs.

What makes `TypesBuilder` particularly interesting is its two-phase accumulation. First, users add type definitions—mappings from names like "rust" to globs like `*.rs`. Second, users make selections—which types to include or exclude. Only during `build()` do these two phases merge. This separation matters because users might select types before all definitions are added, or might add the same type multiple times with different globs.

The builder stores definitions in a `HashMap` for O(1) lookup and selections in a `Vec` that preserves ordering. The ordering matters for precedence: later selections override earlier ones. When you run `rg --type rust --type-not rust`, the negation wins because it comes last.

See: Companion Code Section 1

## Section 2: The Selection Enum and Generic Design

The `Selection` enum encapsulates whether a file type should be included (`Select`) or excluded (`Negate`). This is a common pattern in Rust: using an enum to represent mutually exclusive states rather than, say, a boolean flag plus optional data.

What's clever here is the generic parameter `T`. During the building phase, selections carry no associated data—they're `Selection<()>`. After building, they carry the actual file type definition—they're `Selection<FileTypeDef>`. The `map` method enables this transformation, converting `Selection<()>` to `Selection<FileTypeDef>` by looking up each type name and attaching its definition.

This design avoids a common anti-pattern: storing resolved and unresolved data in the same structure with optional fields. Instead, the type system distinguishes between the two states. The builder works with `Selection<()>` because it doesn't yet know if all selected types are valid. Only during `build()` do we resolve names to definitions and fail if any are missing.

See: Companion Code Section 2

---

## Section 3: The Types Matcher Structure

The `Types` struct is the compiled, ready-to-match result. Understanding its fields reveals the data structure design underlying efficient file type matching.

The `defs` field stores all file type definitions, sorted for consistent output. The `selections` field stores user choices in order. The `has_selected` flag is a precomputed optimization—if the user selected at least one type, then any non-matching file should be ignored. Without this flag, we'd have to scan all selections every time we matched a file that didn't match any glob.

The `glob_to_selection` vector is the key data structure. When a glob matches, we get its index in the `GlobSet`. But we need to know which selection that glob came from and which glob within that selection matched. This two-level mapping enables the matcher to report not just "this file matched" but "this file matched because of the rust type's *.rs glob."

The `matches` field is a thread-safe object pool for temporary storage during matching. Rather than allocating a new vector for every file matched, we reuse vectors from the pool. The `Arc<Pool<...>>` type makes this safe across threads, which matters for parallel directory walking.

See: Companion Code Section 3

## Section 4: The Glob Wrapper Type

The `Glob` struct wraps match results with additional context about why a match occurred. Its design using an inner enum (`GlobInner`) demonstrates a Rust pattern for hiding implementation details while providing a clean public API.

The `UnmatchedIgnore` variant handles a subtle case: when the user selects types and a file doesn't match any of them. This isn't "no match"—it's an active decision to ignore the file. The distinction matters for ripgrep's filtering logic downstream.

The `Matched` variant carries a reference to the file type definition that matched. The lifetime `'a` ties this reference to the `Types` matcher, ensuring the definition remains valid as long as the `Glob` result is used. This is borrowing in action: the match result doesn't own the definition, it borrows it from the matcher.

The public API exposes only `file_type_def()`, which returns an `Option`. Callers don't need to know about the inner enum—they just ask "what type matched?" and handle the case where no specific type did. This is information hiding through Rust's type system.

See: Companion Code Section 4

## Section 5: The Matching Logic

The `matched` method is where theory meets practice. It takes a path and returns whether that path should be whitelisted, ignored, or neither. The logic has several layers that handle edge cases while maintaining performance.

First, directories are always skipped—file types apply to files, not directories. Second, if there are no selections at all, we return early. Third, we extract just the file name because file type globs match names, not full paths.

The actual matching uses the compiled `GlobSet`, which was built from all globs across all selections. When multiple globs match, precedence goes to the last match—this is why we call `matches.last()`. The `glob_to_selection` mapping then tells us which selection and which glob within that selection matched.

The return logic handles the interaction between selections and negations. A negated match returns `Match::Ignore`. A selected match returns `Match::Whitelist`. No match with active

selections returns `Match::Ignore`—this is the "if you asked for specific types, everything else is excluded" behavior. No match with no active selections returns `Match::None`—no filtering occurs.

See: Companion Code Section 5

---

## Section 6: Building Type Definitions

The `add` and `add_def` methods show two different APIs for adding type definitions. The `add` method is programmatic: you pass a name and a glob directly. The `add_def` method parses a string format that's suitable for command-line input.

The string format supports two styles. The basic format `name:glob` creates a simple mapping. The include format `name:include:type1,type2` creates a composite type by copying globs from existing types. This composition is evaluated immediately—the globs are copied, not referenced. If you later add more globs to the included types, the composite doesn't see them.

The validation rules prevent subtle bugs. Type names must be alphanumeric to avoid conflicts with glob syntax. The special name "all" is reserved for selecting or negating all types at once. Invalid definitions return errors rather than silently failing, giving users clear feedback.

Notice the use of `entry().or_insert_with()` for the HashMap. This pattern efficiently handles both "this type is new" and "this type already exists and needs another glob" without double lookups.

See: Companion Code Section 6

---

## Section 7: The Build Process

The `build` method transforms the builder's accumulated state into a compiled matcher. This is where validation happens and where the efficient data structures are constructed.

The method iterates through selections in order, resolving each type name to its definition. Unknown types produce errors immediately—we fail fast rather than building a partial matcher. For each selection, we add all its globs to a `GlobSetBuilder`, recording the mapping from glob index to selection index.

The `literal_separator(true)` setting on each glob is crucial. It means that `*` won't match path separators, so `*.rs` matches `foo.rs` but not `foo/bar.rs`. This matches user expectations for file type matching.

The final `Types` struct contains everything needed for matching without referencing the builder. This separation is important: the builder might add more definitions later (it's mutable), but the built matcher is frozen. Users can build multiple matchers from the same builder at different stages of configuration.

See: Companion Code Section 7

---

## Section 8: Default Types and Extensibility

The `add_defaults` method populates the builder with ripgrep's built-in file type definitions. These come from a separate constant `DEFAULT_TYPES`, which is a static array defined elsewhere in the crate. The method simply iterates through this array, calling `add` for each name-glob pair.

The loop structure handles types with multiple names and multiple extensions. For example, Python files might be recognized by both "py" and "python" names, and both `*.py` and `*.pyw` extensions. The nested loop adds all combinations.

The `expect` call is safe here because default types are known to be valid at compile time. This is one of the few places where `expect` is appropriate—truly impossible failures in well-tested code paths. The static message makes test failures clear if this assumption ever breaks.

This extensibility model—built-in defaults plus user additions—appears throughout ripgrep. Users get sensible behavior out of the box but can customize everything. The builder pattern makes this layering natural: start empty, add defaults, add customs, build.

See: Companion Code Section 8

---

## Section 9: The Select and Negate API

The `select` and `negate` methods are deceptively simple: they just push `Selection` values onto a vector. The complexity lies in what these selections mean and how they interact.

Both methods handle the special name "all" by iterating through all currently defined types. This happens at call time, not build time—if you add more types after calling `select("all")`, those new types won't be selected. This is intuitive behavior but worth understanding.

The methods return `&mut TypesBuilder` for method chaining. This fluent API style is common in Rust builders and enables readable configuration code like

`builder.add_defaults().select("rust").negate("test")`.

The ordering of selections matters for precedence. If you select "rust" then negate "rust", files matching `*.rs` will be ignored. If you negate "rust" then select "rust", they'll be whitelisted. This last-writer-wins semantic is simple to understand and implement.

See: Companion Code Section 9

---

## Section 10: Integration with the Ignore System

This file is part of the `ignore` crate, which handles all of ripgrep's file filtering. The `Types` matcher integrates with the broader ignore system through the common `Match` enum and through the directory walker that uses it.

The `Match` enum (defined elsewhere in the crate) provides a tri-state result: `Whitelist`, `Ignore`, or `None`. File type matching uses all three states. `Whitelist` means "this file matches a selected type." `Ignore` means "this file matches a negated type or doesn't match any selected type." `None` means "file types don't apply to this decision."

The directory walker calls `matched` for each file and combines the result with other ignore rules (gitignore, hidden files, etc.). The `Match` algebra—how multiple match results combine—is defined at the crate level. Types participate in this algebra but don't define it.

This separation of concerns keeps the file type code focused on one task: determining if a file matches a type. The higher-level policy of what to do with that determination lives elsewhere.

See: Companion Code Section 10

---

## Key Takeaways

First, the builder pattern in this file shows how to handle two-phase configuration—definitions first, then selections—with validation deferred until build time.

Second, generic enums like `Selection<T>` let you represent the same concept at different stages of processing, with the type parameter indicating what data is available.

Third, the `glob_to_selection` mapping demonstrates a common pattern: building auxiliary data structures during compilation to enable efficient lookups during runtime.

Fourth, object pools (`Pool<Vec<usize>>`) amortize allocation costs in hot paths, a technique that becomes important when matching millions of files.

Fifth, the three-state `Match` enum enables composition of multiple filtering rules, each of which can contribute "yes," "no," or "I don't care" to the final decision.

Sixth, separating the matcher (`Types`) from the builder (`TypesBuilder`) ensures that the hot path—matching files—works with immutable, optimized data structures.

---

## What to Read Next

How do default file types get defined? Read `crates/ignore/src/default_types.rs` to see the static data that powers `add_defaults`.

How does the directory walker use type matching? Read `crates/ignore/src/walk.rs` to see types integrated with gitignore rules and hidden file handling.

How does the `Match` enum work and how do multiple matchers compose? Read `crates/ignore/src/lib.rs` for the crate-level types that tie everything together.

How does `GlobSet` achieve efficient multi-glob matching? Review `crates/globset/src/lib.rs` which you studied in a previous lesson.

# ripgrep crates/ignore/src/walk.rs: Parallel Directory Walking

## What This File Does

This file implements the core directory traversal machinery that powers ripgrep's file discovery. When you run `rg pattern`, something needs to walk through directories, respect `.gitignore` files, skip hidden files, and ultimately produce a stream of files to search. This file provides two implementations of that traversal: a single-threaded iterator (`Walk`) and a parallel work-stealing implementation (`WalkParallel`).

The parallel implementation is particularly sophisticated. Rather than using a simple thread pool with a shared queue, it implements a work-stealing algorithm where each worker thread has its own local stack of directories to process. When a thread runs out of work, it can steal from other threads. This design minimizes synchronization overhead and keeps all CPU cores busy even when directory sizes vary wildly across the tree.

---

## Section 1: The Directory Entry Abstraction

The file begins by defining `DirEntry`, which is the public type that represents a single file or directory discovered during traversal. This isn't just a thin wrapper around the standard library's `DirEntry`—it's a unification layer that handles three fundamentally different sources of entries.

The design challenge here is that entries can come from the `walkdir` crate during normal traversal, they can be synthesized from raw paths in the parallel iterator, or they can represent special pseudo-entries like stdin. The `DirEntryInner` enum captures these three variants: `Stdin`, `Walkdir`, and `Raw`. Each variant requires different handling for operations like retrieving metadata or determining the file type.

The `DirEntry` type also carries an optional error field. This is a crucial design decision—ignore file parsing errors shouldn't abort the entire traversal. Instead, the error travels with the directory entry so callers can decide how to handle it. This "error attachment" pattern appears throughout ripgrep's codebase, allowing errors to be reported without disrupting the primary operation.

See: Companion Code Section 1

## Section 2: Platform-Specific Raw Entries

The `DirEntryRaw` struct demonstrates a pattern common in cross-platform Rust code: using conditional compilation to handle platform differences at the type level. On Unix, directory entries carry an inode number that's cheap to obtain during directory listing. On Windows, the full metadata comes essentially free during directory enumeration.

This matters for performance. When the parallel iterator reads a directory, it can construct `DirEntryRaw` values directly without immediately requiring expensive `stat` calls. The type stores just enough information to defer those calls until actually needed. On Windows, the metadata is stored because Windows provides it during directory reading. On Unix, just the inode is stored because that's what `readdir` provides.

The `from_entry` and `from_path` constructors show this platform-aware construction. The `#[cfg(unix)]` and `#[cfg(windows)]` blocks compile to completely different code on each platform. There's even a fallback for non-standard platforms that returns an error, ensuring the code at least compiles on exotic targets like WebAssembly.

See: Companion Code Section 2

## Section 3: WalkBuilder and the Builder Pattern

The `WalkBuilder` struct is one of the most complex builders in the ripgrep codebase, with over twenty configuration methods. This connects directly to the Builder Pattern thread we've been following through `searcher/mod.rs` and `printer/standard.rs`. The pattern here follows the same principles: construct a builder, configure it through method chains, then call a terminal method to produce the final product.

What's unique about `WalkBuilder` is that it can produce two fundamentally different types through `build()` and `build_parallel()`. The `Walk` type implements the standard `Iterator` trait and works sequentially. The `WalkParallel` type doesn't implement `Iterator` at all—it uses a callback-based API because parallel iteration doesn't fit the iterator model.

The builder stores all configuration in a flat structure of optional values and flags. Note the use of `OnceLock` for `global_gitignores_relative_to`—this is a lazy initialization pattern where we only query the current working directory from the environment if it's actually needed, and we only do it

once. The `get_or_set_current_dir` method implements this lazy initialization, logging the result so users can debug gitignore-related issues.

See: Companion Code Section 3

---

## Section 4: The Single-Threaded Walk Iterator

The `Walk` struct implements Rust's `Iterator` trait, which means it integrates seamlessly with all of Rust's iterator adapters like `filter`, `map`, and `collect`. The implementation maintains a stack of iterators (one per root path provided) and a current ignore matcher that evolves as directories are entered and exited.

The `next` method reveals the traversal logic. It wraps the `walkdir` crate's iterator with a `WalkEventIter` adapter that transforms flat directory entry streams into a structured sequence of "enter directory," "file," and "exit directory" events. This event-based approach is essential for maintaining the ignore matcher stack. When entering a directory, we push new ignore rules from any `.gitignore` files found there. When exiting, we pop back to the parent's rules.

The `skip_entry` method shows the priority ordering for ignore checking. It emphasizes that "trivial skipping" based on ignore patterns happens before any expensive filesystem operations. A comment specifically mentions a Windows scenario where filesystem operations could trigger remote file downloads—the kind of real-world edge case that production software must handle.

See: Companion Code Section 4

---

## Section 5: WalkEvent and Directory Structure

The `WalkEventIter` wrapper demonstrates how to transform one iterator's interface into another. The `walkdir` crate produces a flat stream of directory entries, but for ignore file handling, we need to know when we're entering versus exiting directories. This adapter tracks depth and synthesizes `WalkEvent::Exit` events when the depth decreases.

The implementation is subtle. When the iterator encounters an entry at a lower depth than the current depth, it means we've finished processing a directory and ascended back to its parent. The adapter stashes the current entry in `self.next`, emits an `Exit` event, decrements the depth, and on the next call returns the stashed entry. This lookahead-and-stash pattern is a common idiom when you need to emit synthetic events in an iterator.

Understanding this transformation is key to understanding how ignore rules are stacked. Each `WalkEvent::Dir` triggers pushing new ignore rules, and each `WalkEvent::Exit` triggers popping. Without this structure, we'd have no way to know when to clean up ignore matchers for directories we've finished processing.

See: Companion Code Section 5

---

## Section 6: WalkState and Parallel Visitor Traits

Before diving into the parallel implementation, the file defines the control flow types that govern parallel traversal. `WalkState` is an enum with three variants: `Continue`, `Skip`, and `Quit`. Unlike an iterator where you just stop calling `next()`, parallel workers need explicit signals to coordinate behavior.

The `ParallelVisitor` and `ParallelVisitorBuilder` traits define the callback interface for parallel traversal. The builder pattern here is different from `WalkBuilder`—it creates per-thread visitors. Each thread gets its own visitor instance, avoiding the need for locking during the hot path of directory traversal.

This design enables a powerful pattern: each thread can accumulate results in thread-local storage without synchronization, then merge results after traversal completes. The trait documentation specifically mentions implementing `Drop` for cleanup, indicating that the trait design anticipates stateful visitors that need finalization.

See: Companion Code Section 6

---

## Section 7: The Work-Stealing Stack

The `Stack` struct implements work-stealing using the `crossbeam-deque` crate. Work-stealing is a scheduling algorithm where each worker has its own local queue. When a worker's queue is empty, it can "steal" work from other workers' queues. This provides excellent load balancing without the contention of a single shared queue.

The stack uses `Deque::new_lifo()`, which is crucial for memory efficiency. A comment explains that depth-first traversal (LIFO) dramatically reduces peak memory usage compared to breadth-first (FIFO). When searching a directory like all of crates.io with many nested gitignores, breadth-first would keep thousands of directory states in memory simultaneously. Depth-first processes each subtree completely before moving to siblings.

The `steal` method shows the stealing strategy: iterate through other workers' stealers starting from the neighbor, wrapping around. The `steal_batch_and_pop` call doesn't just steal one item—it steals a batch for amortized efficiency. This is the kind of detail that separates a textbook work-stealing implementation from a production-quality one.

See: Companion Code Section 7

---

## Section 8: The WalkParallel Structure

The `WalkParallel` struct holds the configuration for parallel traversal but doesn't contain the parallel machinery itself. That machinery—the stacks, the workers, the shared atomic flags—is constructed fresh for each `visit()` call. This design means a single `WalkParallel` could theoretically be visited multiple times, though in practice ripgrep uses it once.

The `visit` method shows the initialization sequence. First, root paths are converted to initial work items. Stdin is handled specially—there's no work to do for stdin, it's just passed directly to the visitor. For real paths, we check the device number if `same_file_system` is enabled, and construct raw directory entries.

The thread spawning uses `std::thread::scope`, a feature stabilized in Rust 1.63 that enables spawning threads that can borrow from the enclosing scope. This is perfect for the work-stealing pattern because we need threads to share the stacks and atomic flags, and the scoped API guarantees all threads complete before `visit` returns.

See: Companion Code Section 8

---

## Section 9: Worker Thread Logic

The `Worker` struct contains both the shared state (work stack, atomic flags) and per-worker state (visitor, configuration). The `run` method is the main loop: get work, process it, repeat until no work remains or someone signals quit.

The `run_one` method handles a single work item. The logic flow is: check if we should visit this entry based on depth, handle symlinks and non-directories immediately, add parent ignore rules if this is a root entry, check if we're still on the same filesystem, read the directory contents, invoke the visitor callback, then generate work items for each child entry.

Notice how the visitor is called before we know if the directory read will succeed. This ensures users always get notified of directories, even if we can't read their contents. The error handling path calls the visitor with the error, giving the callback a chance to log it or take action.

See: Companion Code Section 9

---

## Section 10: Work Generation and Termination

The `generate_work` method transforms filesystem directory entries into work items, applying all the filtering logic along the way. It handles symlinks specially when `follow_links` is enabled, including detection of symlink loops. It checks the ignore matcher, the stdout skip handle, filesize limits, and any custom filter predicate.

The `get_work` method implements the termination protocol. When a worker's local stack is empty and it can't steal work, it decrements the `active_workers` counter. If the counter reaches zero, all workers are idle simultaneously, meaning no more work exists. The worker then sends a quit message that propagates to wake other potentially sleeping workers.

The sleep-and-retry loop in `get_work` handles the non-blocking nature of the work-stealing stack. Rather than busy-waiting, workers sleep for 1ms intervals while waiting for work. This is a tradeoff between latency and CPU usage—at search termination, this brief sleep prevents the CPU from spinning unnecessarily.

See: Companion Code Section 10

---

## Section 11: Helper Functions and Platform Handling

The file concludes with several helper functions that handle cross-cutting concerns. The `check_symlink_loop` function walks up the ignore stack looking for a handle that matches the child—if found, we have a loop and return an error. The `path_equals` function determines if a path corresponds to the same file as a handle, used for skipping stdout when it's redirected to a file in the search path.

The `device_num` function shows three-way platform handling: Unix uses the `dev()` method from `MetadataExt`, Windows uses the `winapi-util` crate to get volume serial numbers, and everything else returns an error. This function enables the "same file system" feature that prevents descending into mounted filesystems.

The `stdout_handle` function is defensive programming at its finest. It handles the case where a user runs something like `rg pattern > results.txt` while `results.txt` is in the search path. Without this check, ripgrep could read from a file it's simultaneously writing to, causing unbounded growth.

See: Companion Code Section 11

---

## Key Takeaways

First, parallel file traversal requires fundamentally different abstractions than sequential traversal—you can't just parallelize an iterator. The `WalkParallel` type uses a visitor pattern with callbacks precisely because parallel work doesn't fit the pull-based iterator model.

Second, work-stealing with LIFO stacks provides both load balancing and memory efficiency. The choice of depth-first versus breadth-first traversal has enormous practical implications for memory usage on large directory trees.

Third, platform-specific code can be cleanly encapsulated using conditional compilation at the type and function level. The `DirEntryRaw` struct stores different fields on Unix versus Windows, matching what each platform provides efficiently.

Fourth, error attachment—carrying errors alongside values rather than halting iteration—enables robust handling of partial failures. Ignore file parse errors travel with directory entries rather than aborting traversal.

Fifth, the builder pattern scales to complex configuration scenarios. `WalkBuilder` has over twenty configuration methods, yet the interface remains discoverable and the implementation remains maintainable.

Sixth, termination detection in parallel systems requires careful coordination. The active worker counting protocol ensures all workers agree that work is exhausted before any exit.

---

## What to Read Next

How does the ignore matcher actually decide what to skip? Read `crates/ignore/src/dir.rs` for the `Ignore` type and its matching logic.

What's the `walkdir` crate and why does ripgrep wrap it? The crate provides the core recursive directory iteration, and ripgrep adds ignore handling on top.

How does the parallel walker get used in the main search loop? Read `crates/core/search.rs` to see how `WalkParallel::visit` integrates with the search machinery.

How do the glob patterns from earlier lessons connect to walking? The `Override` and `Types` matchers built from glob patterns are passed to `WalkBuilder` and checked during traversal.

# ripgrep crates/cli/src/lib.rs: CLI Utilities Overview

## What This File Does

This file serves as the public interface for ripgrep's CLI utilities crate—a collection of reusable tools for building command-line search applications. Rather than implementing a single cohesive abstraction, it functions as a utility library that aggregates helpful functionality: terminal detection, colored output, escape sequence handling, pattern parsing, and process management for decompression.

The crate's philosophy centers on improving failure modes. Every utility is designed not just to work correctly in the happy path, but to produce clear, actionable error messages when something goes wrong. This is a deliberate design choice that reflects ripgrep's commitment to user experience. When a pattern contains invalid UTF-8, the error tells you exactly which byte and line caused the problem. When a decompression subprocess fails, you see its stderr output rather than a cryptic error code.

---

## Section 1: The Crate's Documentation-First Design

The file opens with an extensive module-level documentation comment that serves as both API reference and design document. This is a pattern we've seen throughout ripgrep's codebase—treating documentation as a first-class citizen of the code. The `#![deny(missing_docs)]` attribute at the top enforces this: the code will not compile if any public item lacks documentation.

What's notable about this documentation is how it's organized around use cases rather than types. Instead of listing "here are the structs and functions," it groups functionality by what you're trying to accomplish: "Standard I/O," "Coloring and buffering," "Escaping," and so on. This user-centric organization makes the crate more discoverable for developers who know what they want to do but not what function does it.

The documentation also makes explicit cross-platform guarantees. The statement "to the best extent possible, everything in this crate works on Windows, macOS and Linux" sets expectations upfront. This isn't just a nice-to-have—for a tool like ripgrep that aims to work everywhere, cross-platform behavior is a core requirement that shapes every design decision in the crate.

See: Companion Code Section 1

## Section 2: Module Organization and Re-exports

The file follows a common Rust pattern for crate organization: private modules that implement functionality, with a carefully curated public API exposed through re-exports. The seven private modules—`decompress`, `escape`, `hostname`, `human`, `pattern`, `process`, and `wtr`—each handle a distinct concern. The `pub use` block then selects exactly which types and functions become part of the public API.

This separation serves multiple purposes. First, it allows implementation details to change without breaking downstream code. The internal structure of `decompress.rs` could be completely rewritten, and as long as `DecompressionReader` maintains its interface, users wouldn't notice. Second, it creates a flat namespace for common operations—users write `grep_cli::unescape` rather than `grep_cli::escape::unescape`, which makes the API more ergonomic.

Notice how the re-exports are grouped by module origin. All the decompression-related types come together, then escape functions, then hostname, and so on. This organization in the source code makes it easy to understand what the crate offers at a glance, even before reading any documentation.

See: Companion Code Section 2

---

## Section 3: Heuristic Stdin Detection

The `is_readable_stdin` function tackles a surprisingly subtle problem: determining whether a program was invoked with data piped to its standard input. This matters for search tools because `rg foo` should search the current directory, but `rg foo < file` should search the piped file instead.

The function's approach is explicitly heuristic, and the documentation is refreshingly honest about this. When introspection fails or produces ambiguous results, it defaults to `false`—assuming stdin is not readable. The reasoning is pragmatic: falsely assuming stdin is readable would cause the program to hang waiting for input that will never come. Falsely assuming it's not readable merely ignores piped data, which users can work around with explicit flags.

The implementation uses a two-step check. First, it verifies that stdin is not a terminal using the standard library's `IsTerminal` trait. If stdin is a terminal, the user is typing interactively, so there's

no piped data. If stdin is not a terminal, the function calls into platform-specific code to determine what kind of file descriptor it actually is.

See: Companion Code Section 3

---

## Section 4: Platform-Specific Implementation with Conditional Compilation

The Unix implementation of stdin detection demonstrates Rust's approach to platform-specific code through conditional compilation. The `#[cfg(unix)]` attribute means this code is only compiled on Unix-like systems. There's a parallel `#[cfg(windows)]` block for Windows, and a fallback `#[cfg(not(any(unix, windows)))]` for other platforms.

On Unix, the code clones the stdin file descriptor, retrieves its metadata, and checks the file type. A file, FIFO (named pipe), or socket all indicate readable data. The code carefully handles errors at each step, logging debug information and returning `false` rather than panicking. This defensive approach ensures the function never crashes the program, even in unusual environments.

The logging throughout the function is particularly instructive. Each decision point logs what it found and what conclusion it drew. This transforms debugging from "why did ripgrep ignore my piped input?" into "check the debug logs to see exactly what stdin detection found." For a heuristic that can fail, this observability is essential.

See: Companion Code Section 4

---

## Section 5: The Windows Implementation Contrast

The Windows implementation follows the same logical structure but uses completely different system APIs. Instead of file metadata and file type checks, it uses `winapi_util` to query the handle type. On Windows, "disk" and "pipe" types indicate readable stdin, analogous to Unix's file, FIFO, and socket types.

This parallel structure—same interface, different implementation—is the key to cross-platform Rust code. The caller doesn't need to know whether they're on Unix or Windows; they just call `is_readable_stdin()` and get a boolean. The conditional compilation happens at build time, so there's no runtime overhead for platform detection.

The fallback implementation for non-Unix, non-Windows platforms simply returns `false`. This conservative default means ripgrep will work (though with reduced functionality) even on exotic platforms that might be added to Rust's tier-2 or tier-3 support in the future.

See: Companion Code Section 5

---

## Section 6: Deprecation Patterns and API Evolution

The three `is_tty_*` functions demonstrate how to evolve a public API gracefully. These functions originally provided TTY detection before Rust's standard library offered this capability. Now that `std::io::IsTerminal` exists (stabilized in Rust 1.70), these functions are redundant.

Rather than removing them immediately (which would break existing code), the crate marks them as deprecated with the `#[deprecated]` attribute. The attribute includes `since` to indicate when the deprecation occurred and `note` to guide users toward the replacement. This produces compiler warnings that tell developers exactly how to update their code.

The implementations now simply delegate to the standard library trait, ensuring that deprecated and non-deprecated code paths produce identical results. This pattern—deprecate, delegate, document—is the standard way to sunset functionality in Rust libraries without breaking semver compatibility.

See: Companion Code Section 6

---

## Section 7: Escape Sequence Utilities

The escape and unescape functions solve a practical problem: command-line arguments are typically UTF-8 strings, but sometimes you need to express arbitrary bytes. Typing a literal `0xFF` byte into a terminal is difficult or impossible, but typing `\xFF` is straightforward.

The `unescape` function interprets these escape sequences, transforming the six-byte string `\xFF` into the single byte `0xFF`. The `escape` function does the reverse, taking arbitrary bytes and producing a human-readable UTF-8 representation. This bidirectionality is crucial for error messages—when ripgrep needs to show which bytes matched, it can escape them into something displayable.

The `_os` variants (`unescape_os`, `escape_os`) handle operating system strings (`OsStr` and `OsString`), which on Windows may contain non-UTF-8 data. This ensures the escape utilities work

correctly across platforms, even when dealing with filenames that aren't valid UTF-8.

See: Companion Code Section 7

---

## Section 8: Pattern Parsing with Actionable Errors

Pattern handling demonstrates the crate's commitment to user-friendly error messages. Regular expression engines typically require UTF-8 patterns, but command-line arguments aren't guaranteed to be UTF-8. The standard library's conversion functions produce errors like "invalid utf-8 sequence"—accurate but unhelpful.

The `pattern_from_bytes` and `pattern_from_os` functions instead produce errors that include the exact byte offset where invalid UTF-8 was found. When reading patterns from files, `patterns_from_path` adds line numbers to the error. A user seeing "invalid UTF-8 at byte 15 on line 3" can immediately locate and fix the problem.

This pattern of wrapping standard operations with enhanced error reporting appears throughout ripgrep. The extra development effort pays off every time a user encounters an error and doesn't have to guess what went wrong.

See: Companion Code Section 8

---

## Section 9: Process Output Handling with CommandReader

The `CommandReader` and related types provide infrastructure for running external processes and reading their output. The primary use case is decompression: when ripgrep encounters a `.gz` file, it can spawn `gzip -d` and search the decompressed output.

What distinguishes `CommandReader` from simply calling `Command::new().stdout(Stdio::piped())` is error handling. If the subprocess exits with an error code, `CommandReader` captures its stderr and converts it into a Rust error. Instead of seeing "process exited with code 1," users see the actual error message from the decompression tool.

The builder pattern appears here as `CommandReaderBuilder`, continuing the theme we've seen across ripgrep's crates. Configuration options like asynchronous stderr reading (to prevent deadlocks) can be set before spawning the process. This matches the configuration patterns in the searcher, printer, and ignore crates.

See: Companion Code Section 9

## Section 10: Human-Readable Size Parsing

The `parse_human_readable_size` function handles a common CLI need: parsing strings like "2M" or "500K" into byte counts. This makes command-line interfaces more user-friendly—typing `--max-filesize 2M` is easier than `--max-filesize 2097152`.

Error handling here follows the same philosophy as pattern parsing. Instead of a generic "parse error," invalid input produces messages explaining the expected format. The `ParseSizeError` type likely includes information about what was found versus what was expected, helping users correct their input.

This utility encapsulates a small but frequently needed piece of functionality. By placing it in the CLI crate, it's available to any search tool built on ripgrep's infrastructure, not just ripgrep itself.

See: Companion Code Section 10

---

## Section 11: Standard Output with Intelligent Buffering

The `stdout`, `stdout_buffered_block`, and `stdout_buffered_line` functions construct `StandardStream` instances with different buffering strategies. This addresses a subtle performance and usability tradeoff.

Line buffering writes output after each newline, providing low latency when a user is watching results scroll by. Block buffering collects output into larger chunks before writing, which is faster but introduces latency. The default `stdout` function chooses automatically: line buffering when connected to a terminal (where latency matters), block buffering otherwise (where throughput matters).

The `StandardStream` type comes from the `termcolor` crate and implements `WriteColor`, enabling colored output that works correctly across platforms. On Unix, this means ANSI escape codes; on Windows, it means native console APIs. By returning `StandardStream` rather than a bare `Stdout`, these functions provide color support with appropriate buffering in a single, convenient package.

See: Companion Code Section 11

## Key Takeaways

First, utility crates should be organized around use cases, not implementation structure. The documentation groups functions by what users want to accomplish, making the API discoverable.

Second, heuristic operations should be explicitly documented as such, with clear explanations of failure modes and fallback behaviors. The stdin detection function's documentation is exemplary in this regard.

Third, platform-specific code belongs behind conditional compilation, presenting a unified interface to callers. The same function call works on Unix, Windows, and other platforms without caller awareness.

Fourth, deprecation should guide users toward alternatives, not just warn that something will break. The `#[deprecated]` attribute's `note` field is essential for graceful API evolution.

Fifth, error messages are a feature, not an afterthought. The pattern parsing utilities exist specifically because the standard library's error messages aren't helpful enough for end users.

Sixth, builder patterns provide consistency across a codebase. `CommandReaderBuilder` and `DecompressionReaderBuilder` follow the same patterns as builders in the searcher and printer crates.

## What to Read Next

How does decompression matching work? Read `crates/cli/src/decompress.rs` to see how file extensions are matched to decompression programs.

What does the escape sequence implementation look like? Read `crates/cli/src/escape.rs` for the parsing logic behind `unescape` and `escape`.

How does the writer handle terminal colors? Read `crates/cli/src/wtr.rs` to see how `StandardStream` wraps termcolor with buffering logic.

How do these utilities get used in practice? Read `crates/core/app.rs` to see how ripgrep's main binary configures and uses CLI utilities.

# ripgrep crates/cli/src/decompress.rs: Decompression Support

## What This File Does

This file implements transparent decompression for ripgrep, allowing the tool to search inside compressed files without requiring users to manually decompress them first. When you run `rg --search-zip "pattern" archive.gz`, this module determines that `.gz` files should be piped through `gzip -d -c`, spawns that process, and provides the decompressed output as a standard Rust `io::Read` stream.

The design philosophy here is pragmatic: rather than bundling decompression libraries (which would increase binary size and maintenance burden), ripgrep delegates to standard command-line tools that are typically already installed on users' systems. This approach leverages the Unix philosophy of composing small tools, accepting the overhead of process spawning in exchange for simplicity, portability, and automatic support for whatever decompression tools the user has available.

---

## Section 1: The Two-Layer Builder Architecture

The decompression system follows ripgrep's characteristic two-layer builder pattern. There's a `DecompressionMatcherBuilder` that configures how file extensions map to decompression commands, and a `DecompressionReaderBuilder` that uses a matcher to create actual streaming readers. This separation mirrors patterns we've seen throughout the codebase—the matcher handles the "what command should I use?" question, while the reader builder handles the "how should I run that command?" question.

This layering serves a practical purpose: the matcher is expensive to construct because it compiles glob patterns into an optimized matching structure, but cheap to query. By separating matcher construction from reader construction, code that processes many files can build the matcher once and reuse it. The reader builder even provides a `get_matcher()` method so callers can inspect or share the compiled matcher.

The `DecompressionCommand` struct ties together the three pieces of information needed for each file type: the glob pattern to match, the path to the decompression binary, and the arguments to

pass. Notice that the binary path is stored as a `PathBuf` rather than just a program name—this reflects a deliberate security decision we'll explore shortly.

See: Companion Code Section 1

---

## Section 2: Building the Matcher with Layered Defaults

The `DecompressionMatcherBuilder::build()` method demonstrates a clean pattern for combining default behavior with user customization. First, it conditionally loads the default decompression commands based on the `defaults` flag. Then it chains the defaults with user-provided commands, adding each glob to a `GlobSetBuilder` while simultaneously building a parallel vector of commands.

The order of this chaining matters. User commands come after defaults, and when multiple globs match the same file, "the most recently added glob takes precedence." This means user customizations automatically override built-in rules without requiring explicit removal of defaults. If you want to handle `.gz` files with a custom decompressor, just add your rule—it will shadow the default gzip handling.

The error handling here wraps glob compilation errors into `CommandError` via `io::Error`. This is a pragmatic choice that keeps the error type hierarchy simple—all failures in this module surface as `CommandError`, whether they come from glob syntax errors, missing binaries, or process spawning failures.

See: Companion Code Section 2

---

## Section 3: The Associate Methods and Security Considerations

The builder provides two methods for adding custom glob-to-command mappings: `associate()` and `try_associate()`. This is a common Rust API pattern—one method that panics or silently handles errors (convenient for cases where failure is acceptable), and one that returns a `Result` (necessary when callers need to handle failures explicitly).

The `associate()` method simply calls `try_associate()` and ignores any error with `let _ = ...`. This is intentional—when loading configuration that might specify decompression commands for tools the user doesn't have installed, silent failure is the right behavior. The matcher should still work for the tools that are available.

The critical detail in `try_associate()` is the call to `try_resolve_binary()`. Rather than storing the program name and letting `Command::new()` resolve it at execution time, the code resolves the binary to an absolute path at configuration time. The documentation explains why: on Windows, `CreateProcess` will search the current working directory for executables, which is a security vulnerability when searching untrusted directory trees. By resolving binaries against PATH at configuration time, ripgrep avoids accidentally executing malicious programs that might be lurking in the directories being searched.

See: Companion Code Section 3

---

## Section 4: Command Lookup and the GlobSet Integration

The `DecompressionMatcher` struct holds a compiled `GlobSet` and a parallel vector of commands. The `GlobSet` from the `globset` crate is an optimized data structure for matching a path against many glob patterns simultaneously—much more efficient than checking each pattern sequentially.

The `command()` method shows the matching logic. It calls `self.globs.matches(path)`, which returns an iterator of indices for all globs that matched. The `.next_back()` call takes the last matching index, implementing the "most recently added takes precedence" rule. Once we have an index, we use it to look up the corresponding `DecompressionCommand` and build a `Command` from it.

Notice that the method returns `Option<Command>` rather than `Result`. The absence of a matching command isn't an error—it just means this file doesn't need decompression. The caller can then decide whether to fall back to reading the file directly or report an error.

The `has_command()` method provides a cheaper check when you just need to know whether decompression would be attempted, without actually building the `Command`. This uses `is_match()` rather than `matches()`, which can short-circuit as soon as any glob matches.

See: Companion Code Section 4

---

## Section 5: The DecompressionReaderBuilder and Graceful Degradation

The `DecompressionReaderBuilder` composes a `DecompressionMatcher` with a `CommandReaderBuilder` from the process module we saw earlier. This composition follows the

builder pattern conventions established throughout ripgrep—complex behavior emerges from combining simpler, configurable pieces.

The `build()` method implements graceful degradation, a key design principle for the decompression system. Three paths are possible: if no glob matches, return a passthru reader that reads the file directly; if a glob matches but the command fails to spawn, log a debug message and return a passthru reader; if everything succeeds, return a reader wrapping the decompression process.

This behavior is perfect for ripgrep's use case. When searching a directory tree, some `.gz` files might be encountered on systems without gzip installed. Rather than failing the entire search, ripgrep logs a debug message and moves on, treating the file as if it weren't compressed. Users who want stricter behavior can check the logs or validate their environment separately.

The `async_stderr()` option passes through to the underlying `CommandReaderBuilder`. This controls whether stderr from decompression processes is read asynchronously. As the documentation notes, this prevents deadlocks with noisy processes that write extensively to stderr—a subtle but important detail when spawning external commands.

See: Companion Code Section 5

## Section 6: The DecompressionReader and Unified I/O

The `DecompressionReader` struct encapsulates either a `CommandReader` (when decompression is happening) or a raw `File` (for passthru mode). The type is `Result<CommandReader, File>`—an unconventional use of `Result` where both variants represent valid states. Using `Result` here provides a clean way to store one of two types without defining a custom enum.

The `io::Read` implementation demonstrates why this design works well. Both `CommandReader` and `File` implement `Read`, so the trait implementation just delegates to whichever inner type is present. From the caller's perspective, there's no way to tell whether decompression is happening—they just get bytes from a reader.

The `new()` constructor provides the simplest API: pass a path, get a reader. This builds a fresh `DecompressionReaderBuilder` internally, which means it constructs and compiles the default matcher for every call. The documentation explicitly warns about this: "When creating readers for many paths, it is better to use the builder since it will amortize the cost of constructing the matcher."

See: Companion Code Section 6

## Section 7: Resource Management and the Close Method

The `close()` method addresses an important resource management concern with process-based decompression. When you spawn a child process and pipe its output, several resources need cleanup: the process itself, its stdout and stderr handles, and potentially a background thread reading stderr asynchronously.

For a `CommandReader`, `close()` waits for the process to finish and checks its exit status. If the process failed, the error includes stderr output, which often contains the reason for failure. For a passthru `File`, `close()` is a no-op—there's nothing special to clean up beyond what `Drop` handles.

The documentation explains when explicit `close()` calls are necessary: "This method should be called after partially reading a file to prevent resource leakage." If you read to EOF, the `Read` implementation handles cleanup automatically. But if you stop reading early (perhaps because you found what you were searching for), calling `close()` ensures the child process is properly terminated rather than left running.

The `close()` method is idempotent—safe to call multiple times. The first call does the actual cleanup, and subsequent calls are no-ops. This pattern makes cleanup logic in `Drop` straightforward: just call `close()` and print any errors as warnings.

See: Companion Code Section 7

## Section 8: Binary Resolution and Platform Security

The `resolve_binary()` and `try_resolve_binary()` functions implement the security-conscious binary lookup discussed earlier. The public `resolve_binary()` is a no-op on non-Windows platforms—the security issue it addresses is Windows-specific. The private `try_resolve_binary()` does the actual work and is used internally regardless of platform.

The implementation walks through the system PATH, joining each path component with the program name and checking if the result is an executable file. On Windows, it also tries adding `.com` and `.exe` extensions, matching the platform's behavior for finding executables.

The `is_exe()` helper function is deliberately simple—it just checks that the path exists and isn't a directory. This is intentionally permissive because the actual execution will do proper permission checks. The goal here is finding candidates, not fully validating them.

One subtle point: the function can return a relative path if PATH contains relative components. The code accepts this because "it is assumed that the user has set this explicitly." This is a reasonable trust boundary—we protect against implicit current-directory searches, but we trust explicit PATH configuration.

See: Companion Code Section 8

---

## Section 9: Default Decompression Commands

The `default_decompression_commands()` function defines the built-in associations between file extensions and decompression tools. Each command follows a consistent pattern: the program name followed by flags that make it decompress (`-d`) and write to stdout (`-c`).

The implementation uses const arrays for each command's arguments, which helps with readability and ensures the strings are stored in the binary's read-only data section. The `add()` helper function attempts to resolve each binary and silently skips commands whose binaries aren't found. This means the default matcher only includes entries for tools that are actually installed.

The supported formats reflect common compression tools across Unix-like systems: gzip, bzip2, xz, lz4, lzma, brotli, zstd, and traditional Unix compress. The glob patterns handle both standalone compressed files (`*.gz`) and compressed tarballs (`*.tgz`), recognizing that tarball compression is a common use case.

Notice that xz handles both `.xz` and `.lzma` files—LZMA is the underlying compression algorithm used by xz, and xz can decompress LZMA-format files with the `--format=lzma` flag. This kind of detail reflects the practical experience baked into these defaults.

See: Companion Code Section 9

---

## Key Takeaways

1. The two-layer builder pattern (matcher builder plus reader builder) separates the expensive glob compilation from the per-file reader construction, enabling efficient processing of many files with a single compiled matcher.
2. Graceful degradation is central to the design—missing decompression tools result in passthru reading with debug logging rather than hard failures, making the feature robust across diverse environments.

3. Security considerations drove the binary resolution design: resolving programs against PATH at configuration time prevents Windows's `CreateProcess` from searching the current directory, avoiding potential code execution vulnerabilities when searching untrusted directories.
4. The `Result<CommandReader, File>` pattern in `DecompressionReader` provides a clean way to unify two different underlying types behind a single `io::Read` interface, demonstrating that `Result` can be useful even when both variants represent success.
5. The default commands use the `-c` flag pattern (write to stdout) consistently, allowing the decompression tools to stream their output through pipes rather than writing to files—essential for integration with ripgrep's streaming architecture.
6. Resource management for process-based I/O requires explicit attention: the `close()` method exists because simply dropping a reader might leave child processes running or fail to capture their error output.

## What to Read Next

How does ripgrep actually execute these decompression commands and capture their output? Read `crates/cli/src/process.rs` to understand the `CommandReader` that powers the process I/O.

How does ripgrep decide which files to search in the first place? Read `crates/ignore/src/walk.rs` to see how the parallel directory walker integrates with file type filtering.

How does the glob matching work under the hood? The `globset` crate documentation at docs.rs/globset explains the pattern syntax and matching algorithm used throughout ripgrep's filtering logic.

# ripgrep crates/cli/src/process.rs: Process Management

## What This File Does

This file provides infrastructure for executing external processes and streaming their output safely. When ripgrep needs to search through compressed files, it spawns decompression tools like `gzip` or `xz` and reads their output as a stream. The challenge is doing this correctly: processes write to both stdout and stderr, and if you don't read from both simultaneously, the process can deadlock when one buffer fills up.

The `CommandReader` abstraction solves this by wrapping a child process in a type that implements Rust's standard `Read` trait. It handles the tricky details of process lifecycle management—spawning, reading, waiting, and cleaning up—while preventing deadlocks through asynchronous stderr reading. This design lets the rest of ripgrep treat decompressed data as just another input stream, without worrying about the underlying process management complexity.

## Section 1: The Custom Error Type Strategy

The file begins by defining `CommandError`, a custom error type that captures the two ways a command can fail: through an I/O error during process management, or by exiting with an error status and writing to stderr. This distinction matters because the error message you want to show the user differs dramatically between these cases.

When a process fails due to an I/O error—perhaps the executable doesn't exist—you want to show that system error directly. But when a process runs successfully from the operating system's perspective but exits with a non-zero status, the meaningful error message is whatever the process wrote to stderr. A decompression tool might write "file is corrupted" or "unsupported format," and that's what the user needs to see.

The enum-based design with `CommandErrorKind` is a common Rust pattern for errors with multiple causes. The outer `CommandError` struct provides a stable public API while the inner enum handles the variant logic. Notice how the `kind` field is private—users of this type interact through methods, not by matching on variants directly. This encapsulation allows the internal representation to change without breaking callers.

See: Companion Code Section 1

---

## Section 2: Error Display Formatting for Human Readability

The `Display` implementation for `CommandError` shows thoughtful attention to user experience. For I/O errors, it delegates directly to the underlying error's display logic. But for stderr content, it applies formatting that makes error messages from external tools visually distinct and readable.

The stderr formatting wraps the content in a box made of dashes, with the message trimmed of leading and trailing whitespace. This visual treatment serves a practical purpose: when ripgrep reports that a decompression tool failed, users can immediately distinguish between ripgrep's own message and the quoted stderr from the external tool. The 79-character line of dashes provides a clear boundary.

The empty stderr case receives special handling with the placeholder `<stderr is empty>`. This acknowledges reality: some tools exit with error codes without writing anything useful to stderr. Rather than showing nothing, which might confuse users, the error explicitly states that stderr was empty. This helps with debugging—users know the process failed but didn't explain why.

See: Companion Code Section 2

---

## Section 3: Bidirectional Error Conversion

The `From` implementations create a bridge between `CommandError` and `io::Error`, allowing seamless conversion in both directions. This is crucial for integrating with code that expects standard I/O errors while preserving the richer error information when available.

Converting from `io::Error` to `CommandError` is straightforward—the I/O error simply gets wrapped. The reverse direction is more nuanced. When converting a `CommandError` back to `io::Error`, I/O errors unwrap cleanly, but stderr-based errors become `io::Error` with `ErrorKind::Other`. The original `CommandError` becomes the source of the new I/O error, preserving the full error chain.

This bidirectional conversion supports a common pattern in Rust: functions that need to work with the standard library's error types while internally using richer domain-specific errors. The `?` operator works naturally in both directions because Rust's error conversion is driven by `From` implementations.

See: Companion Code Section 3

---

## Section 4: The Builder Pattern for Command Configuration

The `CommandReaderBuilder` follows the builder pattern you've seen throughout ripgrep's codebase. Even though there's currently only one configuration option—`async_stderr`—the builder establishes an extensible API that can grow without breaking existing code.

The builder stores configuration state and produces a `CommandReader` through its `build` method. The separation between configuration and construction is valuable: you might want to create a builder once, configure it based on runtime conditions, and then use it to build multiple readers. The `Clone` and `Default` derives support this use case.

The `build` method takes a mutable reference to a `Command` rather than owning it. This design choice reflects that the caller has already configured the command with its arguments, environment variables, and working directory. The builder only overrides the stdout and stderr settings to `Stdio::piped()`, which is necessary for capturing the process output.

See: Companion Code Section 4

---

## Section 5: Understanding the Deadlock Problem

The `async_stderr` option addresses a subtle but serious problem with process I/O. When you spawn a process, the operating system creates buffers for stdout and stderr. These buffers have limited capacity—typically a few kilobytes. If a process writes enough to fill a buffer and nobody reads from it, the process blocks.

Imagine spawning a decompression tool that writes the decompressed data to stdout and progress messages to stderr. If you read only from stdout, stderr's buffer might fill up. The process blocks waiting to write to stderr, but you're blocked waiting to read from stdout. Neither can proceed—a classic deadlock.

The async_stderr option solves this by reading stderr on a separate thread. While the main thread reads stdout, the background thread drains stderr. Neither buffer fills up, so the process runs to completion. The documentation notes this spawns an additional thread, which has some overhead, but the alternative—potential deadlocks—is far worse. This is why async_stderr defaults to enabled.

See: Companion Code Section 5

---

## Section 6: The CommandReader Core Structure

The `CommandReader` struct holds three things: the child process, a stderr reader, and an EOF flag. The child process owns the stdout pipe (as an `Option<ChildStdout>` inside the `Child` struct), the stderr reader handles the stderr pipe through either sync or async reading, and the EOF flag tracks whether we've finished reading stdout.

The EOF flag serves an important purpose in cleanup logic. When you close a reader before consuming all output—perhaps because you found what you were searching for—the child process receives a broken pipe signal when it tries to write more. This is expected behavior, not an error. The EOF flag lets the close logic distinguish between "closed early, broken pipe is fine" and "read to completion, any error is real."

This structure embodies a key principle: the reader owns the child process and is responsible for its complete lifecycle. When the reader is dropped, it must clean up the process. When the reader encounters an error, it must provide useful diagnostics. Ownership in Rust isn't just about memory—it's about resources generally.

See: Companion Code Section 6

---

## Section 7: The Close Protocol

The `close` method implements careful cleanup logic that handles multiple scenarios correctly. First, it checks if stdout has already been taken—if so, close was already called, and we return early. This makes close idempotent, safe to call multiple times.

Closing starts by dropping stdout. This closes the pipe, which signals to a well-behaved child process that no more output will be read. The process should then exit. We call `wait()` to collect the exit status, which also reaps the process and prevents it from becoming a zombie.

If the process exited successfully, we're done. If it failed, we read whatever stderr contains and return it as an error. But there's a special case: if we didn't read to EOF (meaning we closed early) and stderr is empty, we assume success. This handles the broken pipe scenario gracefully—the process "failed" only because we stopped reading, not because of an actual error.

See: Companion Code Section 7

## Section 8: Resource Safety Through Drop

The `Drop` implementation ensures that resources are always cleaned up, even if the caller forgets to call `close`. This is defense-in-depth: proper usage should call close explicitly to handle errors, but Drop provides a safety net.

When Drop calls close and encounters an error, it can't return that error—Drop has no return value. Instead, it logs a warning. This is a pragmatic compromise: dropping silently might hide real problems, but panicking would be too severe for what might be an expected broken pipe.

The documentation explicitly discusses this design choice, advising callers to call close explicitly if they want to handle errors properly. This is honest API design: the Drop implementation prevents resource leaks and process zombies, but it can't fully replace explicit cleanup. Rust's type system can't express "you must call close before dropping," so documentation fills the gap.

See: Companion Code Section 8

---

## Section 9: Implementing the Read Trait

The `Read` implementation is where `CommandReader` becomes useful to the rest of the codebase. By implementing `io::Read`, a `CommandReader` can be used anywhere that expects a reader: passed to functions that read bytes, wrapped in buffered readers, or used with the standard read methods.

The implementation delegates to the child's stdout, but with two additions. First, it handles the case where stdout has been taken (after close), returning zero bytes to indicate EOF. Second, when a read returns zero bytes naturally (the process finished), it sets the EOF flag and calls close, converting any process failure into an I/O error.

This automatic close-on-EOF is convenient: if you read a `CommandReader` to completion using `read_to_end`, cleanup happens automatically. The EOF flag ensures that close knows this was a natural end, not an early termination. This distinction affects whether a broken pipe is considered an error.

See: Companion Code Section 9

## Section 10: The Stderr Reader Abstraction

The `StderrReader` enum encapsulates the choice between synchronous and asynchronous stderr reading. The async variant holds a join handle to a background thread, while the sync variant holds the stderr pipe directly. Both variants provide the same interface through `read_to_end`.

The async variant is constructed by spawning a thread that reads all of stderr into a `CommandError`. The thread runs independently while the main thread reads stdout. When we need the stderr content—at close time—we join the thread and retrieve its result.

The raw identifier syntax `r#async` deserves explanation. `async` became a reserved keyword in Rust 2018 for async/await syntax. But here it's used as a method name, which requires the `r#` prefix to tell Rust "this isn't the keyword, it's an identifier." This is a minor inconvenience from Rust's evolution, preserved for API consistency.

See: Companion Code Section 10

---

## Section 11: Thread Safety Considerations

The `read_to_end` method on `StderrReader` shows careful handling of the thread join. It takes the join handle out of the Option (using `take()`), ensuring that `read_to_end` can only be called once for the async case. The `expect` message documents this constraint: calling it twice is a programmer error.

The second `expect` on `join()` asserts that the thread doesn't panic. The stderr reading thread is simple—it just reads bytes—so panics are unexpected. If one did occur, it would indicate a bug, not a runtime error, so panicking in the caller is appropriate.

For the sync case, `read_to_end` simply reads stderr directly. This works when stderr output is small or when you know the process won't write much to stderr while producing stdout. The sync path avoids thread overhead but requires this guarantee from the caller.

See: Companion Code Section 11

## Section 12: The Helper Function for Stderr Reading

The `stderr_to_command_error` function handles the actual reading of stderr into a `CommandError`. It reads all bytes into a vector, then wraps either the bytes (on success) or the I/O error (on failure) into the appropriate error variant.

This helper is used both by the async thread and the sync reader, avoiding code duplication. It takes a mutable reference to `ChildStderr`, which implements `Read`, and returns a `CommandError` that captures either the stderr content or any error encountered while reading.

The function's simplicity is intentional. Complex error handling logic lives in `CommandError::stderr` and `CommandError::io`. This function just connects the `Read` trait to those constructors, following the single responsibility principle.

See: Companion Code Section 12

---

## Key Takeaways

First, process management in Rust requires explicit attention to resource cleanup—child processes need to be waited on, pipes need to be closed, and the order of these operations matters for correctness.

Second, the deadlock risk with process I/O is real and subtle. Reading stdout and stderr concurrently, whether through async I/O or threads, prevents buffer-full deadlocks that can otherwise freeze your program.

Third, bidirectional `From` implementations create smooth interoperability between custom error types and standard library types, enabling the `?` operator to work naturally across API boundaries.

Fourth, the builder pattern provides extensibility for APIs even when current configuration is minimal. `CommandReaderBuilder` has one option today but can grow without breaking callers.

Fifth, implementing standard traits like `Read` integrates custom types into Rust's ecosystem. A `CommandReader` works with any code expecting a reader, from `BufReader` wrappers to `read_to_end` calls.

Sixth, Drop implementations provide resource safety as a last resort, but explicit cleanup methods give callers control over error handling. Both are needed for robust APIs.

## What to Read Next

How does ripgrep use this process infrastructure? Read `crates/cli/src/decompress.rs` to see `CommandReader` in action, spawning decompression tools and treating their output as searchable streams.

Want to understand more about streaming I/O in ripgrep? The searcher module in `crates/grep/src/searcher/mod.rs` shows how various input sources, including process output, feed into the search machinery.

Curious about error handling patterns across ripgrep? The main binary in `crates/core/main.rs` demonstrates how errors from different subsystems, including command errors, propagate to user-facing messages.

# ripgrep crates/core/main.rs: Application Entry Point

## What This File Does

This is where everything comes together. The `main.rs` file is the application entry point for ripgrep, orchestrating all the sophisticated machinery we've studied across thirty-one previous lessons into a cohesive command-line tool. It handles argument parsing, mode dispatch, single-threaded and multi-threaded search execution, and the careful exit code management that makes ripgrep a good Unix citizen.

What makes this file particularly instructive is how it demonstrates idiomatic Rust application architecture. You'll see conditional compilation for platform-specific allocators, the strategic use of atomics for thread coordination, and a thoughtful approach to error handling that distinguishes between "something went wrong" and "the user piped output somewhere that closed." This file is the capstone that shows how all the builder patterns, the parallel walking infrastructure, and the printer abstractions work in concert.

---

## Section 1: The Allocator Override

Before we even reach executable code, ripgrep makes a significant platform-specific decision: which memory allocator to use. The file begins with a conditional compilation block that swaps in jemalloc for musl-based builds, and the reasoning embedded in the comments is a masterclass in pragmatic systems programming.

Rust's default behavior since version 1.56 is to use the system allocator. On Linux with glibc, this is typically excellent. But ripgrep also supports static compilation via musl libc, which prioritizes small binary size and simplicity over allocation performance. The musl allocator, while perfectly correct, can significantly slow down ripgrep's operation. The solution is to conditionally use jemalloc only when building against musl on 64-bit systems.

This is **conditional compilation at the deepest level**. The resulting binary literally has different allocation code depending on the build target. The `#[global_allocator]` attribute tells Rust's runtime to route all heap allocations through the specified allocator, completely replacing the default. Notice that this isn't runtime configuration—it's a compile-time decision that produces fundamentally different binaries.

The constraints are illuminating: jemalloc only on musl (because glibc's allocator is fine), only on 64-bit systems (because jemalloc doesn't support 32-bit architectures well), and explicitly not unconditional (because jemalloc increases compile times and the system allocator works well in most cases). This is the kind of nuanced decision-making that separates production-quality tools from naive implementations.

See: Companion Code Section 1

---

## Section 2: The Main Function and Exit Code Philosophy

The `main` function itself is remarkably brief, but it encapsulates ripgrep's entire error handling philosophy. It returns `ExitCode`, the modern Rust approach available since version 1.61, which provides a clean way to communicate success or failure to the shell.

The function immediately delegates to `run()` with parsed arguments, then handles the result. The interesting part is what happens on error. Ripgrep walks the error chain looking for a specific condition: a broken pipe. When you pipe ripgrep's output to a command like `head` that closes early, the write fails with `BrokenPipe`. Rather than treating this as an error, ripgrep exits with code 0—success.

This matches traditional Unix convention. In C programs, writing to a closed pipe sends a SIGPIPE signal that terminates the process without ceremony. Rust's runtime doesn't request SIGPIPE signals, so the program receives an I/O error instead. Ripgrep's explicit handling ensures consistent behavior regardless of this implementation difference. The docstring's cryptic quote—"Then, as it was, then again it will be"—playfully references this: behavior that existed in Unix's past should exist again in its Rust-powered future.

For all other errors, ripgrep prints to stderr using a locked macro (we'll explore that shortly) and exits with code 2. This three-way exit code scheme—0 for success, 1 for no matches, 2 for errors—is standard grep behavior and critical for scripting.

See: Companion Code Section 2

---

## Section 3: The Run Function and Mode Dispatch

The `run` function is ripgrep's central dispatcher. It receives a `ParseResult<HiArgs>`—the high-level argument representation that earlier lessons on the flags module prepared—and routes execution to the appropriate handler based on the operating mode.

The pattern matching on `ParseResult` demonstrates a clean separation of concerns. Parse errors become immediate error returns. "Special" modes (help, version) short-circuit to their handler. Only successful parses with full `HiArgs` proceed to the main logic. This ensures that asking for help never fails due to other initialization problems like an inaccessible current directory.

The mode dispatch within successful parsing shows ripgrep's operational versatility. Search mode branches based on thread count—single-threaded search when `threads() == 1`, parallel search otherwise. There's also a subtle optimization: if `matches_possible()` returns false (perhaps because the pattern can never match anything), ripgrep skips searching entirely.

The `files` mode (listing paths without searching) mirrors this single/multi-threaded split. `Types` mode handles `--type-list`. `Generate` mode produces shell completions and man pages. This is **comprehensive mode handling** that turns a single binary into multiple tools depending on invocation.

The exit code calculation at the end is precise: success (0) if matches were found and either we're in quiet mode or no errors occurred; error (2) if any errors were logged; no-match (1) otherwise. The `messages::errored()` function tracks whether any error messages were emitted during execution, providing a global error state without threading explicit error counts through every function.

See: Companion Code Section 3

---

## Section 4: Single-Threaded Search Architecture

The `search` function implements ripgrep's single-threaded search path, and it's a beautiful example of how the builder pattern ecosystem we've studied across many lessons comes together. Every major component is constructed through builder methods on `HiArgs`.

The setup phase constructs the complete search infrastructure: a haystack builder to convert directory walk results into searchable units, a walk builder to traverse the filesystem, an optional sorter to order results, and finally the search worker itself composed of a matcher, searcher, and printer. Each of these came from dedicated builder types we explored in previous lessons—now they're being orchestrated together.

The search loop demonstrates graceful error handling. When a search operation fails with `BrokenPipe`, the loop breaks cleanly rather than propagating the error. Other errors get logged with `err_message!` but don't stop iteration. This resilience is crucial: one unreadable file shouldn't prevent searching thousands of others.

Statistics accumulation shows optional feature handling. The `stats` variable is `Option<Stats>`—if the user requested statistics (via `--stats`), it's `Some`, otherwise `None`. The `if let Some(ref mut stats)` pattern elegantly skips statistics work when not requested. This is idiomatic Rust: making optional features zero-cost when disabled.

The `quit_after_match` check enables `--quiet` mode behavior, where ripgrep stops immediately upon finding any match. Combined with the `has_implicit_path` check that triggers a helpful error message when no files were searched (usually due to overly broad ignore rules), this function handles all the edge cases a command-line tool must consider.

See: Companion Code Section 4

---

## Section 5: Parallel Search and Thread Coordination

The `search_parallel` function is where ripgrep's performance really shines, and it demonstrates sophisticated thread coordination patterns. The parallelism comes from the directory walker—`build_parallel()` spawns worker threads that each process files independently—while this function provides the per-file search logic.

Thread-safe state sharing uses **atomics for simple flags and mutexes for complex data**. The `matched` and `searched` flags are `AtomicBool` because they only need single-bit state transitions. The `stats` aggregation requires a `Mutex<Stats>` because it involves complex arithmetic on multiple fields. This is a common pattern in parallel Rust: use the lightest-weight synchronization primitive that provides the guarantees you need.

The `SeqCst` (sequentially consistent) ordering on atomic operations is the strongest memory ordering guarantee. While `Relaxed` might work here since the flags are simple coordination signals, `SeqCst` eliminates any subtle ordering bugs at minimal performance cost for operations that happen once per file.

The `searcher.clone()` call inside the closure is critical. Each parallel worker gets its own searcher instance, avoiding contention on shared search state. The builders we studied are designed for this exact usage pattern: construct once, clone for each worker.

Output coordination uses the `BufferWriter` pattern from the `termcolor` crate. Each worker writes to its own buffer, then atomically flushes to stdout. This prevents output interleaving that would make results unreadable. The flush happens through `bufwtr.print()`, which handles the locking internally.

The `WalkState` return values control the parallel walker. `Continue` means "keep going," while `Quit` signals all workers to stop. This is how `--quiet` mode achieves early termination across threads—one worker finds a match, stores `true` to the atomic, returns `Quit`, and the walker coordinates shutdown.

See: Companion Code Section 5

---

## Section 6: File Listing Modes

Ripgrep's `--files` flag lists paths that would be searched without actually searching them. The `files` and `files_parallel` functions implement this mode, and they reveal interesting design decisions about I/O and thread coordination.

The single-threaded `files` function is straightforward: iterate paths, print each one. The `PathPrinter` abstraction (constructed via builder, naturally) handles platform-specific path formatting and separators. Early termination via `quit_after_match` works identically to search mode.

The parallel version is more interesting. It uses a **channel-based architecture** with a dedicated printing thread. Worker threads send discovered paths to a channel, and a single printer thread receives and outputs them. The comment admits this design choice wasn't rigorously benchmarked—it's "under the presumption that it's probably faster than using a mutex"—but the architecture makes sense: channel sends are typically faster than mutex acquisitions under contention.

The explicit `drop(tx)` before joining the print thread is essential. The channel has multiple sender clones (one per worker), and the receiver only returns `None` when all senders are dropped. Without the explicit drop, the print thread would wait forever for more paths. This is a common pattern with Rust channels and a frequent source of bugs when forgotten.

Error handling on the print thread join is careful. A broken pipe gets swallowed (it's graceful termination), but other errors propagate. The `join().unwrap()` handles thread panics—unwrap is appropriate here because a panic in the print thread represents a bug, not a recoverable error.

See: Companion Code Section 6

---

## Section 7: Type Listing and Generation Modes

The `types` function implements `--type-list`, displaying ripgrep's known file type definitions and their associated globs. This is simpler than search modes but demonstrates good practices for formatted output.

The function writes directly to stdout using `std::io::Write` trait methods. Each type definition is written with its name, a colon separator, and a comma-separated list of globs. The `first` flag pattern avoids leading commas in the glob list—a common idiom when building delimited strings.

The exit code reflects whether any types were defined: 1 if the list is empty (shouldn't happen in practice), 0 otherwise. This follows grep's convention of exit code 1 meaning "nothing found."

The `generate` function handles ripgrep's self-documentation features: generating man pages and shell completions. These are **meta-features** that help with distribution and integration. The implementations delegate to functions in the flags module that we studied earlier.

The `special` function handles help and version output with intentional short-circuiting. When users request `--help` or `--version`, they want that output regardless of other problems like missing directories or invalid configurations. The comment explains that initialization steps are deliberately skipped to ensure help is always available.

The PCRE2 version check is notable: it returns an error exit code if PCRE2 support isn't compiled in. This lets scripts check for optional feature availability programmatically.

See: Companion Code Section 7

---

## Section 8: Helper Functions and User Feedback

The file concludes with two helper functions that handle user communication. The `eprint_nothing_searched` function addresses a specific failure mode: when ripgrep's filters exclude everything, leaving nothing to search.

This happens more often than you'd expect. A broad `.gitignore` pattern, an overly restrictive `--type` filter, or a directory of only hidden files can result in zero searchable paths. Rather than silently returning "no matches," ripgrep proactively suggests running with `--debug` to understand the filtering. The function only triggers when no explicit path was provided—if the user asked to search a specific (empty) directory, silence is appropriate.

The `print_stats` function generates the statistics output when `--stats` is requested. It demonstrates **format polymorphism**: the same statistics can be rendered as human-readable text

or as JSON, depending on the search mode. JSON mode produces a `summary` message type that complements the JSON printer's existing output format.

The text format uses Rust's formatted write macro with named parameters. The statistics object provides methods like `matches()`, `matched_lines()`, and `bytes_searched()` that we saw when studying the printer module. The elapsed time calculation uses `Instant::now().duration_since(started)` to compute wall-clock search duration.

The function returns a `Result` but callers typically ignore errors. Stats printing failure (probably due to stdout issues) shouldn't change ripgrep's exit code—the search results are what matter.

See: Companion Code Section 8

---

## Section 9: Module Organization and Macro Usage

The top of the file declares ripgrep's internal modules and sets up a critical macro. The `#[macro_use]` attribute on the `messages` module import makes macros from that module available throughout the crate without explicit imports.

The module declarations—`flags`, `haystack`, `logger`, `search`—partition ripgrep's functionality into focused units. The `flags` module handles all argument parsing and configuration. The `haystack` module defines the abstraction over searchable items. The `logger` module provides logging infrastructure. The `search` module contains the search worker logic.

The `messages` module, used with `#[macro_use]`, provides the `err_message!` and `eprintln_locked!` macros we see throughout the file. These macros write to stderr with proper locking to prevent interleaved output in parallel contexts. The locking is crucial: without it, error messages from multiple threads could merge into garbage.

The imports at the top show ripgrep's key dependencies. `ignore::WalkState` controls parallel directory traversal. `anyhow::Result` provides flexible error handling. `ExitCode` enables proper shell integration. The internal `HiArgs` and `SearchMode` types come from the flags module we studied earlier.

This module structure reflects **separation of concerns**: main.rs orchestrates but doesn't implement. The actual work happens in specialized modules, each with its own focused responsibility.

See: Companion Code Section 9

---

## Key Takeaways

1. **First**, conditional compilation can change fundamental runtime behavior. The allocator selection based on `target_env` produces binaries with different memory allocation strategies, demonstrating that Rust's zero-cost abstractions extend to build-time configuration.
2. **Second**, exit code management is essential for Unix tool compatibility. Ripgrep's three-way scheme (0 for matches, 1 for no matches, 2 for errors) with special handling for broken pipes makes it script-friendly and pipeline-compatible.
3. **Third**, the builder pattern ecosystem enables flexible composition. All major components—matchers, searchers, printers, walkers—are constructed through builders, allowing main.rs to assemble them without knowing their internal complexity.
4. **Fourth**, parallel coordination in Rust uses the lightest-weight primitives that provide necessary guarantees. Atomics for flags, mutexes for complex state, channels for streaming data—each has its place depending on the synchronization requirements.
5. **Fifth**, graceful degradation improves user experience. The "nothing searched" warning, broken pipe handling, and always-available help output show attention to edge cases that separate polished tools from rough prototypes.
6. **Sixth**, mode dispatch transforms a single binary into multiple tools. Search, file listing, type listing, generation, and special modes all share infrastructure while providing distinct functionality based on command-line arguments.

---

## What to Read Next

How does the search worker actually find matches? Explore `crates/core/search.rs` to see how matchers and printers coordinate to process individual files.

What makes the error message macros thread-safe? The `crates/core/messages.rs` module defines `err_message!` and `eprintln_locked!` with the locking behavior referenced throughout this file.

How does argument parsing produce the `HiArgs` structure? Review `crates/core/flags/hiargs.rs` to understand the high-level argument representation that drives all mode decisions.

What is a "haystack" in ripgrep's architecture? The `crates/core/haystack.rs` module defines the abstraction over searchable items that bridges directory walking and actual searching.

# ripgrep crates/core/flags/mod.rs: Flag System Architecture

## What This File Does

This module serves as the architectural foundation for ripgrep's entire command-line interface. It defines the abstraction layer that every flag in ripgrep must conform to, establishes the organizational structure for documentation generation, and provides the type system for representing parsed flag values. Rather than containing the implementation of individual flags, this file establishes the contract that all flags must follow—a trait-based design that enables ripgrep to generate shell completions, help text, and man pages from the same flag definitions.

The design here represents a sophisticated approach to CLI architecture: instead of scattering flag parsing logic throughout the codebase, ripgrep centralizes flag definitions in a way that makes them self-documenting. Each flag implementation carries not just its parsing behavior, but also its documentation, its category for organizational purposes, and metadata for shell completion. This centralization is what allows ripgrep to have consistently high-quality help output, man pages, and shell completions without duplicating information.

---

## Section 1: The Module's Public Contract

The module begins by re-exporting a carefully curated set of types from its submodules. This pattern creates a clean public API while hiding implementation details. Notice how the exports are organized by purpose: completion generators for different shells, documentation generators for help text and man pages, the high-level argument type `HiArgs` that the rest of ripgrep actually uses, and the parsing infrastructure.

The `pub(crate)` visibility is significant here. These exports are available throughout the `core` crate but not to external users of ripgrep as a library. This design decision acknowledges that flag parsing is an internal concern—external code should interact with ripgrep through its libraries like `grep-regex` or `grep-searcher`, not by directly manipulating its CLI infrastructure.

The submodule declarations at the bottom reveal the architecture: `complete` handles shell completions, `config` manages configuration file reading, `defs` contains the actual flag definitions, `doc` generates documentation, `hiargs` and `lowargs` represent two levels of

argument abstraction, and `parse` handles the actual command-line parsing. This separation of concerns allows each submodule to focus on a single responsibility while the `mod.rs` file serves as the integration point.

See: Companion Code Section 1

---

## Section 2: The Flag Trait as Architectural Contract

The `Flag` trait is the heart of this module's design. It defines a contract that every flag in ripgrep must satisfy, ensuring consistency across all command-line options while enabling sophisticated documentation and completion generation. The trait bound requirements tell an important story: `Debug` for introspection, `Send + Sync` for potential parallel processing, and `UnwindSafe + RefUnwindSafe` for panic safety—plus `'static` to ensure flag definitions live for the program's lifetime.

The design decision to use **dynamic dispatch** via `&dyn Flag` is deliberate. The `defs` module provides a global slice of trait objects representing all flags. This means flag definitions can be iterated over, filtered, and processed uniformly without knowing their concrete types. The trade-off is a small runtime cost for virtual dispatch, but the benefit is significant architectural flexibility—new flags can be added without changing any of the documentation generation or parsing infrastructure.

Notice how the trait distinguishes between switches (flags that are either on or off, like `--line-number`) and value-taking flags (like `--max-count NUM`). The `is_switch` method determines how the parser treats what follows the flag on the command line. This is a fundamental parsing distinction that must be known before parsing begins, which is why it's a method on the trait rather than something discovered during parsing.

See: Companion Code Section 2

---

## Section 3: Name Resolution and Aliases

The naming methods on `Flag` reveal ripgrep's flexible approach to flag identification. Each flag requires a long name (`--encoding`), may optionally have a short name (`-E`), can have multiple aliases for compatibility, and may have a negation form (`--no-encoding`). This design supports both ergonomic short forms for common operations and explicit long forms for scripts and documentation.

The validation rules embedded in the documentation comments are important: short names must be ASCII alphanumeric or `.`, long names must be at least 2 bytes and use only `-` or alphanumerics. These constraints ensure flags can be unambiguously parsed and displayed across different terminals and shells.

The negation pattern deserves special attention. Rather than treating negations as separate flags, ripgrep considers them part of the same flag definition. This means `--encoding` and `--no-encoding` share documentation, and the negation is automatically mentioned in the flag's help text. This design reduces documentation burden and ensures negation flags stay synchronized with their positive counterparts.

See: Companion Code Section 3

---

## Section 4: Documentation as First-Class Data

The documentation methods on `Flag` demonstrate that ripgrep treats documentation as a first-class concern, not an afterthought. Each flag carries `doc_short` for the condensed `-h` output, `doc_long` for comprehensive `--help` and man page content, `doc_variable` for the placeholder name in usage text, and `doc_choices` for flags that accept a fixed set of values.

The explicit note about `doc_long` being in "mandoc/mdoc format" is significant. This means flag definitions contain actual man page markup, allowing the same flag definition to generate both terminal help output and properly formatted man pages. This eliminates the common problem of documentation drifting out of sync between different output formats.

The design constraint that `doc_short` should fit within 79 columns per line reflects practical usability concerns. Terminal help output needs to be scannable, which means being ruthlessly concise even if it means sacrificing grammatical completeness. The longer `doc_long` can then provide the full context that users need when they want to understand a flag deeply.

See: Companion Code Section 4

## Section 5: The Category System for Organization

The `Category` enum provides the organizational structure for ripgrep's documentation. With seven categories ranging from `Input` to `OtherBehaviors`, this system ensures that ripgrep's 100+ flags are presented in logical groupings rather than as an overwhelming alphabetical list.

The category names themselves reveal ripgrep's conceptual model: input handling, search execution, haystack filtering, output formatting, output mode changes, logging, and miscellaneous behaviors. This taxonomy helps users navigate the flags by thinking about what aspect of ripgrep's behavior they want to modify.

The `as_str` method is interesting for its stated purpose: providing variable names for template interpolation. This suggests that documentation generation uses some form of template system where categories serve as section markers. The snake-case style (using `-` as separator) matches typical documentation conventions while remaining valid as identifiers in most templating systems.

The derive list including `Ord` and `PartialOrd` indicates that categories have a defined ordering, which determines their presentation sequence in documentation. The order in the enum definition—Input first, OtherBehaviors last—reflects a logical flow from "what goes in" to "what comes out" to "everything else."

See: Companion Code Section 5

---

## Section 6: Shell Completion Support

The `CompletionType` enum addresses the practical challenge of generating useful shell completions. Different flags accept different kinds of values, and shells can provide better completion suggestions when they know what type of value is expected.

The variants tell the story of what ripgrep commonly accepts: `Filename` for path arguments, `Executable` for commands to run (like `--hostname-bin`), `Filetype` for ripgrep's type system (like `--type rust`), and `Encoding` for character encoding names. The `Other` fallback handles everything else, where completions fall back to generic behavior based on `is_switch` and `doc_choices`.

This design shows thoughtful attention to user experience. When a user types `rg --type` and presses Tab, the shell can offer `rust`, `python`, `c`, and other type names rather than generic file completions. This context-aware completion dramatically improves the command-line experience.

See: Companion Code Section 6

## Section 7: The FlagValue Type and Parse Results

The `FlagValue` enum represents the outcome of parsing a single flag from the command line. Its two variants—`Switch(bool)` and `Value(OsString)`—capture the fundamental distinction between flags that are on/off toggles and flags that accept arbitrary values.

The use of `OsString` rather than `String` for values is a deliberate platform-awareness decision. On Unix systems, command-line arguments can contain arbitrary bytes, not just valid UTF-8. Using `OsString` preserves this capability, allowing ripgrep to handle file paths with unusual characters correctly. This is the same pattern we've seen throughout ripgrep's I/O handling—preferring OS-native types over Rust's UTF-8 strings when dealing with external data.

The `unwrap_switch` and `unwrap_value` methods are convenience helpers that panic on type mismatch. The documentation explains why this is acceptable: callers "usually know whether a switch or a value is expected." A mismatch indicates a bug in the flag definition itself, not invalid user input, so a panic is the appropriate response. This is internal code, not a public API, so the panic serves as a development-time assertion rather than a user-facing failure mode.

See: Companion Code Section 7

---

## Section 8: The Update Pattern and State Management

The `update` method on `Flag` is where flag definitions connect to application state. Each flag implementation receives a `FlagValue` (the parsed value) and a mutable reference to `LowArgs` (the accumulating state), then updates that state appropriately.

The convention documented here is important: implementations should "validate but not act." For example, `--hostname-bin` should validate that the provided path looks reasonable, but should not actually run the binary to discover the hostname. This deferred execution pattern ensures that argument parsing is fast and predictable, with side effects happening only after all arguments are processed and validated.

This pattern also addresses the "last one wins" semantic common in command-line tools. If a user specifies `--hostname-bin foo --hostname-bin bar`, only `bar` matters. By deferring execution until after parsing, ripgrep naturally gets this behavior—each update overwrites the previous value, and only the final state is acted upon.

The connection to `LowArgs` hints at ripgrep's two-level argument system. `LowArgs` is a relatively direct representation of what was on the command line, while `HiArgs` (mentioned in the module documentation) is a higher-level representation that's ready for use by the search infrastructure. This separation allows validation and transformation to happen in stages.

See: Companion Code Section 8

---

## Section 9: Connection to the Builder Pattern Ecosystem

This flag system represents the final layer in a larger **Builder pattern ecosystem** that spans ripgrep's architecture. We've seen builders in `grep-searcher` for configuring search behavior, in `grep-printer` for configuring output formatting, in `grep-regex` for configuring pattern compilation, and in `ignore` for configuring directory traversal. The flag system is what populates all these builders.

When the module documentation says that `HiArgs` "knows how to create a multi threaded recursive directory traverser," it's describing the connection point. The flags module parses command-line arguments into `LowArgs`, transforms them into `HiArgs`, and `HiArgs` contains methods that construct properly configured builders for searchers, printers, matchers, and walkers. Each flag ultimately influences one or more of these downstream builders.

This architecture inverts the typical dependency direction. Instead of the CLI code depending on searcher/printer/walker internals, those components expose builder APIs, and the CLI code knows how to configure them through those builders. The flag system is the adapter layer that translates user intent (expressed as command-line arguments) into library configuration (expressed as builder method calls).

See: Companion Code Section 9

---

## Section 10: Thread Safety and Static Lifetime Requirements

The trait bounds on `Flag` deserve deeper examination. `Send + Sync` means flag definitions can be safely shared across threads, which matters because ripgrep may parse arguments in one thread while another thread is generating completions or help text. `UnwindSafe + RefUnwindSafe` ensures that if any code panics while processing a flag, it won't leave shared state in an inconsistent state.

The `'static` lifetime bound is perhaps most significant. It means flag definitions must not contain any borrowed data—they must be fully self-contained. This enables the global slice of `&dyn Flag` values mentioned in the trait documentation. These references can live for the entire program lifetime because the flag definitions themselves are static.

These bounds represent a design trade-off. They prevent flag definitions from being dynamically generated or from containing runtime-determined data. But they enable the simple, efficient architecture where all flags are defined as constants at compile time and accessed through a single global slice. For a CLI tool like ripgrep, where flags are known at compile time, this trade-off makes sense.

See: Companion Code Section 10

---

## Key Takeaways

First, the `Flag` trait demonstrates how traits can serve as architectural contracts, defining not just behavior but documentation requirements, enabling uniform processing of diverse flag types through dynamic dispatch.

Second, ripgrep's separation of flag definitions from flag processing creates a clean architecture where adding new flags requires no changes to parsing, documentation generation, or completion generation—just implementing the trait.

Third, the two-level argument system (`LowArgs` → `HiArgs`) separates parsing concerns from semantic concerns, allowing validation and transformation to happen in clearly defined stages.

Fourth, using `OsString` for flag values preserves platform-native semantics, ensuring ripgrep can handle arbitrary file paths even when they contain non-UTF-8 bytes.

Fifth, the category and completion type systems show that good CLI design treats documentation and shell integration as first-class concerns, not afterthoughts to be bolted on later.

Sixth, the "validate but don't act" convention for flag updates ensures parsing remains fast and predictable, with side effects deferred until the full argument picture is known.

---

## What to Read Next

How do individual flags actually implement this trait? Read `crates/core/flags/defs.rs` to see concrete flag definitions and how they update `LowArgs`.

How does the parser use the `Flag` trait to process command lines? Read `crates/core/flags/parse.rs` to understand the parsing loop and how switches and values are distinguished.

How does `LowArgs` transform into `HiArgs`? Read `crates/core/flags/hiargs.rs` to see how low-level parsed values become high-level configuration.

How does ripgrep generate its man pages and help text? Read `crates/core/flags/doc/` to see how the documentation methods on `Flag` are assembled into complete documentation.

# ripgrep crates/core/flags/lowargs.rs: Low-Level Arguments

## What This File Does

This file defines the `LowArgs` struct and its associated types—the complete collection of "low-level" arguments that ripgrep's CLI parser produces. Think of it as the raw output of argument parsing: every flag value, every option, every mode toggle, all collected into a single structure that mirrors the actual command-line interface as closely as possible.

The key design constraint here is intentional simplicity. Low-level arguments contain validated user input but avoid higher-level abstractions. They don't discover hostnames, don't probe filesystem capabilities, don't make network calls. This separation allows ripgrep to guarantee that help and version information are always available, even when the environment is broken in ways that would cause higher-level configuration to fail. The transformation from these low-level arguments into the builder configurations you've seen in previous lessons happens elsewhere—this file is purely about capturing what the user asked for.

---

## Section 1: The Anatomy of Low-Level Arguments

The `LowArgs` struct is surprisingly large—over fifty fields spanning boolean flags, optional values, vectors of configuration, and numerous mode enumerations. This isn't complexity for its own sake; it reflects the genuine richness of ripgrep's command-line interface. Each field here corresponds directly to one or more CLI flags.

Notice how the struct uses `pub(crate)` visibility for all its fields. This is a deliberate design choice: low-level arguments are internal implementation details, not part of ripgrep's public API. Other modules within the `core` crate can read and write these fields directly, but external code cannot. This gives the ripgrep developers freedom to restructure the argument representation without breaking external consumers.

The struct derives `Debug` and `Default`, which is essential for argument parsing. The parser starts with `LowArgs::default()` and then updates individual fields as it processes each flag. This incremental construction pattern means every field must have a sensible default value—either through `Default` for the type or through explicit initialization.

See: Companion Code Section 1

---

## Section 2: Special Modes and Short-Circuiting

The `SpecialMode` enum represents commands that should bypass normal ripgrep operation entirely. When a user asks for `--help` or `--version`, ripgrep shouldn't attempt to build search configurations, probe for PCRE2 support, or discover hostnames. It should just print the requested information and exit.

This design is a **reliability hedge**. Imagine ripgrep needs to read environment variables or check filesystem permissions to construct its full configuration. If something is wrong with the environment, that construction might fail—but the user should still be able to run `rg --help` to understand how to fix their invocation. By separating special modes from regular operation, ripgrep ensures that basic help and version information remain accessible even in degraded environments.

The distinction between "short" and "long" variants for both help and version reflects ripgrep's Unix heritage. The `-h` flag gives condensed output suitable for quick reference, while `--help` provides comprehensive documentation. Similarly, `-V` shows just the version string, while `--version` includes build features and other details useful for bug reports.

See: Companion Code Section 2

---

## Section 3: Operational Modes and Override Semantics

The `Mode` enum defines what ripgrep is actually going to do: search for patterns, list files, show type definitions, or generate shell completions. Unlike special modes that short-circuit everything, these operational modes represent the primary functionality that ripgrep will execute after parsing.

The `update` method on `Mode` implements an interesting override policy. Search modes (the default) can be overridden by anything. But once you're in a non-search mode like `Files`, other non-search modes can override it, but search modes cannot. This means if you type `rg --files -l`, you stay in `Mode::Files`—the `-l` flag (which normally activates `FilesWithMatches` search mode) doesn't override the explicit `--files` request.

This policy exists because flags often come from multiple sources: the command line, config files, environment variables. A user who explicitly requests `--files` probably wants that behavior even

if their config file includes some search-mode flag. The override semantics prevent surprising interactions between configuration sources.

See: Companion Code Section 3

---

## Section 4: Search Mode Variations

The `SearchMode` enum captures the different ways ripgrep can report search results. Standard mode prints matching lines. `FilesWithMatches` prints only filenames. `Count` prints filenames with match counts. Each mode represents a fundamentally different output format.

Notice that there's no explicit flag for `Standard` mode—it's simply the default. The other modes have activation flags (`-l`, `-c`, `--json`), and some have negation flags (`--no-json`) that reset back to standard mode. This asymmetry reflects the common pattern where you want to override a config file setting back to the default behavior.

The distinction between `Count` and `CountMatches` is subtle but important. `Count` reports the number of matching lines, while `CountMatches` reports the total number of matches. A line with three matches of the pattern would contribute 1 to `Count` but 3 to `CountMatches`. Having both as separate modes means the output format can be optimized for each case.

See: Companion Code Section 4

---

## Section 5: Binary Data Handling

The `BinaryMode` enum addresses one of the trickier aspects of text searching: what happens when ripgrep encounters binary files? The three options represent fundamentally different philosophies about how to handle non-text content.

`Auto` mode, the default, makes context-dependent decisions. Files searched explicitly (named on the command line) get searched with binary suppression—ripgrep will search them but warn if it finds binary content near a match. Files found through directory traversal get more aggressive filtering—once ripgrep decides a file is binary, it stops immediately.

The `SearchAndSuppress` mode includes an interesting implementation detail: it replaces `NUL` bytes with line terminators. This sounds strange until you understand the problem it solves. Binary files often contain long runs of `NUL` bytes, and treating those as line content would create astronomically long "lines" that exhaust heap memory. The `NUL` replacement is a heuristic that keeps memory usage bounded while still allowing binary-adjacent searching.

`AsText` mode removes all binary handling—every file is treated as text, period. This is useful when you know your files contain `NUL` bytes but are actually text (some logging formats do this), or when you genuinely want to search binary content and understand the memory implications.

See: Companion Code Section 5

---

## Section 6: Context Mode and Precedence

The `ContextMode` enum and its companion `ContextModeLimited` struct handle the `-A`, `-B`, and `-C` flags that control how many lines around each match ripgrep displays. This seemingly simple feature has surprisingly complex precedence rules.

The `Passthru` variant is special: it means "print all lines, whether they match or not." This transforms ripgrep into something like `grep --color` where you see the entire file with matches highlighted. It's useful when you want context but don't know how much context you'll need.

`Limited` mode is where the complexity lives. Notice that `ContextModeLimited` tracks `before`, `after`, and `both` as separate `Option<usize>` values rather than just two numbers. This is because `-A` (after) and `-B` (before) always override `-C` (both), regardless of the order they appear. If the user specifies `-C 5 -A 2`, they get 5 lines before and 2 lines after—the `-A` partially overrides the `-C`. This order-independence is important for configuration file support, where the user might want to set a default with `-C` but override just one direction on the command line.

The `get` method resolves these precedences into concrete numbers at the point of use, keeping the parsing logic separate from the resolution logic.

See: Companion Code Section 6

---

## Section 7: Separator Types with Escape Handling

The `ContextSeparator`, `FieldContextSeparator`, and `FieldMatchSeparator` types all follow the same pattern: they wrap a `BString` (binary string) and handle the conversion from user-provided arguments. This wrapping serves two purposes: it enforces that escape sequences are properly processed, and it documents the semantic role of each separator.

The `new` constructors on these types require valid UTF-8 input but then immediately convert to binary strings using `Vec::unescape_bytes`. This allows users to specify binary separators using

escape sequences like `\x00` for a null byte, while still providing a clear error message if the input can't even be parsed as UTF-8.

The `ContextSeparator` has an additional `disabled` constructor that creates a `None` variant, representing the explicit absence of a separator. This is different from an empty string separator—disabled means "don't print anything between context groups," while empty string means "print a blank line between context groups."

These types demonstrate a common Rust pattern: creating **newtypes** that add semantic meaning and validation to primitive types. A `BString` could be anything; a `FieldMatchSeparator` clearly communicates its purpose and guarantees its content has been properly processed.

See: Companion Code Section 7

---

## Section 8: Encoding and Engine Choices

The `EncodingMode` and `EngineChoice` enums represent choices that significantly affect ripgrep's behavior but require careful defaulting. Both use `Auto` or `Default` as their default variant, reflecting the philosophy that ripgrep should "just work" for common cases while allowing expert users to override.

`EncodingMode::Auto` tells ripgrep to sniff for byte-order marks (BOMs) that indicate encoding. `EncodingMode::Some` lets users force a specific encoding while still respecting BOMs. `EncodingMode::Disabled` bypasses all encoding logic and searches raw bytes—useful when you know the encoding detection is wrong or when you're searching binary-ish content.

`EngineChoice` controls which regex engine ripgrep uses. The default Rust regex engine is fast and safe but doesn't support all regex features. PCRE2 supports features like backreferences and lookaround but requires an external library. The `Auto` choice implements a clever fallback: try the default engine first, and if the pattern doesn't compile, automatically try PCRE2. This gives users the best of both worlds—fast patterns when possible, powerful patterns when necessary.

See: Companion Code Section 8

---

## Section 9: Sorting and Platform Detection

The `SortMode` and `SortModeKind` types handle the `--sort` and `--sortr` flags, with an interesting twist: sorting by timestamps isn't always supported. The `supported` method performs **runtime capability detection** to determine whether the requested sort mode will work.

Look at how `supported` is implemented: it attempts to get metadata from the current executable and checks whether the relevant timestamp is available. This isn't checking whether the filesystem supports timestamps in general—it's checking whether this particular Rust runtime on this particular platform can retrieve them. Some platforms might support modification times but not creation times. Some embedded or restricted environments might not support any timestamp access.

This approach—probing capabilities at runtime rather than compile time—is pragmatic. It means ripgrep can be compiled once and deployed to varied environments, gracefully degrading when features aren't available. The user gets a clear error message explaining why their sort mode won't work, rather than a cryptic failure later in execution.

See: Companion Code Section 9

---

## Section 10: Pattern Sources and Type Changes

The `PatternSource` enum unifies two ways of providing patterns to ripgrep: the `-e/--regexp` flag for inline patterns and the `-f/--file` flag for pattern files. By collecting both into a single `Vec<PatternSource>`, ripgrep preserves the relative order of these flags.

Order matters because patterns are tried in sequence, and for certain operations (like replacement), the order determines which pattern "wins." If a user specifies `-f patterns.txt -e 'inline'`, they expect the patterns from the file to come before the inline pattern.

The `TypeChange` enum handles the complex world of file type definitions. Users can add new types (`--type-add`), clear existing ones (`--type-clear`), select types for filtering (`--type`), or negate types (`--type-not`). These operations form a mini-language for customizing ripgrep's built-in file type system.

Importantly, these are recorded as changes, not computed results. The actual type definitions are built elsewhere by applying these changes in order to ripgrep's default type database. This separation keeps low-level arguments purely about "what the user said" rather than "what that means."

See: Companion Code Section 10

## Section 11: Color Choices and Terminal Integration

The `ColorChoice` enum demonstrates how command-line tools handle the perennial question of "should this output be colorful?" The four variants cover the full spectrum of possibilities: never, auto-detect, always, and always-with-ANSI.

The `Ansi` variant exists for a specific historical reason: Windows terminal handling. Older Windows console APIs required special escape sequences different from standard ANSI codes. Modern Windows terminals understand ANSI, but ripgrep needs to support both. `ColorChoice::Ansi` forces ANSI codes even when ripgrep thinks it should use legacy Windows APIs.

The `to_termcolor` method bridges ripgrep's internal representation to the `termcolor` crate's `ColorChoice` type. This is a common pattern when integrating with external crates: define your own types that match your semantics, then provide conversion methods to external types. This insulates your codebase from changes in external crate APIs.

See: Companion Code Section 11

---

## Section 12: Connecting to the Builder Ecosystem

This file is the foundation that the builder pattern ecosystem you've studied depends upon. The `LowArgs` struct captures everything needed to configure searchers, printers, and walkers—but it captures them in their raw, unprocessed form.

Consider how fields like `dfa_size_limit`, `regex_size_limit`, and `threads` are all `Option` types. The `None` case doesn't mean "no limit" or "single-threaded"—it means "use the default." The translation from `None` to actual defaults happens when constructing the builders. This separation means the defaults can be computed contextually (perhaps based on available CPU cores or memory) without complicating the argument parsing.

Similarly, boolean flags like `multiline`, `crlf`, and `fixed_strings` will eventually flow into `RegexMatcherBuilder` and `SearcherBuilder` configurations. The `globs` and `iglobs` vectors will configure file filtering. The `heading`, `column`, and `line_number` options will shape printer output. Every field here has a downstream consumer, but the low-level arguments don't need to know about those consumers—they just faithfully record what the user requested.

See: Companion Code Section 12

---

## Key Takeaways

First, low-level arguments intentionally mirror the CLI surface rather than higher-level abstractions, creating a clean separation between "what the user said" and "what that means for the application."

Second, special modes that short-circuit normal operation ensure reliability—help and version information remain accessible even when the environment is broken.

Third, override semantics in enums like `Mode` and `ContextMode` handle the complexity of configuration coming from multiple sources while maintaining user expectations.

Fourth, newtypes like `ContextSeparator` add semantic meaning and validation to primitive types, making code self-documenting and enforcing invariants at construction time.

Fifth, runtime capability detection (as in `SortMode::supported`) allows a single binary to gracefully handle varied deployment environments.

Sixth, this file demonstrates the value of keeping argument parsing separate from argument interpretation—the transformation to builder configurations happens elsewhere, keeping each layer focused on a single responsibility.

## What to Read Next

How do these low-level arguments get transformed into the higher-level configurations that drive searching? Read `crates/core/flags/hiargs.rs` to see the next stage of argument processing.

How does the parser actually populate these fields? Read `crates/core/flags/parse.rs` to understand the parsing machinery that produces `LowArgs`.

How do the separator types and encoding modes affect actual search output? Review `crates/printer/standard.rs` to see how printer configuration consumes these values.

# ripgrep crates/core/flags/hiargs.rs: High-Level Arguments

## What This File Does

This file represents the culmination of ripgrep's entire argument parsing pipeline. While the low-level arguments (`LowArgs`) capture the raw values from the command line—what flags were passed, what values they had—the high-level arguments (`HiArgs`) transform those raw values into fully-configured, ready-to-use components. Think of `LowArgs` as ingredients and `HiArgs` as the finished meal.

The distinction is more than organizational. Many configuration options can only be determined once *all* arguments are known. You can't build a glob matcher until every `-g` flag has been collected. You can't decide whether to use parallelism until you know both the sorting mode and how many files will be searched. This file is where all those interdependent decisions finally get made, producing the configuration objects that the rest of ripgrep will actually use during search.

## Section 1: The HiArgs Structure and Its Philosophy

The `HiArgs` struct contains over 80 fields, which might seem overwhelming at first. But this exhaustive enumeration serves an important purpose: it makes every configuration decision explicit and queryable. Rather than re-computing "should we show line numbers?" throughout the codebase, that decision gets made once during `HiArgs` construction and stored as a simple boolean.

Notice how the fields fall into distinct categories. Some are direct copies from `LowArgs`—simple values that don't need transformation. Others are computed from multiple low-level arguments, like `quit_after_match` which depends on both `stats` and `quiet`. And some are fully-constructed objects from external crates, like the `globs` field which holds a compiled `ignore::overrides::Override` matcher.

This pattern of "resolve everything upfront" eliminates a class of bugs where different parts of the codebase might make inconsistent decisions. If both the printer and the walker need to know whether filenames should be shown, they both read the same `with_filename` field rather than independently re-deriving that value.

See: Companion Code Section 1

---

## Section 2: The Conversion Process and State Management

The `from_low_args` method orchestrates the transformation from low-level to high-level arguments. It begins with an assertion that might seem paranoid: it verifies that no "special mode" is active. Special modes like `--version` or `--help` cause ripgrep to short-circuit before ever reaching this conversion. The assertion documents this invariant and catches programming errors where someone accidentally tries to build search configuration for a non-search invocation.

The `State` struct introduced here is a clever mechanism for sharing context across conversions. It tracks whether stdout is a terminal (affecting color and heading defaults), whether stdin has been consumed (preventing double-reading), and the current working directory. By passing this state through the various conversion functions, they can coordinate their decisions without global variables.

The method also performs some mode adjustments that handle flag interactions. When `-v` (invert match) is combined with `--count-matches`, the behavior silently becomes `--count` instead—because counting individual matches makes no sense when you're matching *non-matching* lines. These adjustments happen early so downstream code sees a consistent, sensible mode.

See: Companion Code Section 2

---

## Section 3: Pattern Collection and Deduplication

The `Patterns` struct and its `from_low_args` method handle one of the trickier aspects of argument parsing: patterns can come from multiple sources. They might be the first positional argument (the common `rg foo` case), from `-e/--regexp` flags, or from files via `-f/--file`. All these sources must be unified into a single pattern list.

The deduplication logic here deserves attention. Using a `HashSet` to track seen patterns adds memory overhead, but it prevents a real performance problem. If a user accidentally passes duplicate patterns—perhaps through a misconfigured shell alias or a generated pattern file—the regex engine would otherwise have to process those duplicates through parsing and compilation. With thousands of duplicate patterns, this can cause severe slowdowns.

The stdin handling demonstrates defensive programming. Before reading patterns from stdin (when `-f -` is given), the code checks `state.stdin_consumed`. If stdin was already used for

something else, it returns a clear error rather than silently producing unexpected behavior. After reading, it sets `stdin_consumed = true` so that later path handling can also verify stdin hasn't been double-read.

See: Companion Code Section 3

---

## Section 4: Path Resolution and Implicit Behavior

The `Paths` struct captures not just what to search, but *how* the search paths were determined. The `has_implicit_path` field is particularly interesting—it tracks whether ripgrep "guessed" that it should search the current directory because no paths were given.

This matters for output formatting. When you run `rg foo`, ripgrep searches the current directory but doesn't prefix results with `./`. But if you explicitly run `rg foo .`, it does include the prefix. This matches GNU grep's behavior and affects how users process ripgrep's output in scripts. A seemingly minor detail, but one that required tracking through the entire argument processing pipeline.

The `is_one_file` field enables optimizations when searching a single file. Notice the careful logic: it checks that the path isn't a directory using `!paths[0].is_dir()` rather than `paths[0].is_file()`. This distinction matters because `is_file()` can return false for things like device files or named pipes that ripgrep should still treat as single-file searches. The comment even references a specific GitHub issue where this subtlety caused real problems.

See: Companion Code Section 4

---

## Section 5: Binary Detection Strategy

The `BinaryDetection` struct reveals a nuanced approach to handling binary files. Rather than a single binary detection mode, ripgrep maintains *two* modes: one for explicitly specified files and one for implicitly discovered files during directory traversal.

The philosophy here is that explicit is different from implicit. When you specifically ask ripgrep to search a file (`rg foo binary.dat`), you've made a deliberate choice. Ripgrep shouldn't silently skip content or quit early—that would be filtering what you explicitly requested. But when ripgrep discovers files during directory traversal, it's more appropriate to skip binary content because you didn't specifically request those files.

The three binary modes (`AsText`, `SearchAndSuppress`, and the default quit behavior) map to different `BinaryDetection` configurations. When `null_data` is set (the `-z` flag), binary detection is completely disabled because the user is explicitly expecting to search data with NUL bytes. These interactions between flags demonstrate how `HiArgs` must consider the entire flag configuration, not just individual options.

See: Companion Code Section 5

---

## Section 6: The Builder Pattern Ecosystem in Action

The numerous builder methods in `HiArgs` showcase how this file serves as the integration point for ripgrep's builder pattern ecosystem. Methods like `matcher()`, `searcher()`, `printer()`, and `walk_builder()` don't just return objects—they configure complex builders using the resolved high-level arguments.

Consider `printer_standard()`. It creates a `StandardBuilder` and configures over fifteen different options: colors, columns, headings, hyperlinks, separators, and more. Each configuration call corresponds to a field in `HiArgs`, and together they produce a printer perfectly tuned to the user's requested output format. The builder pattern allows this extensive configuration without constructors that take fifteen parameters.

The `matcher()` method demonstrates graceful degradation between regex engines. When the default engine is selected, it tries to build a Rust regex matcher first. If that fails—perhaps due to unsupported features like backreferences—it can suggest PCRE2 as an alternative. When auto mode is selected, it actually *tries* PCRE2 if the Rust engine fails, falling back gracefully and only erroring if both engines reject the pattern.

See: Companion Code Section 6

---

## Section 7: Thread Count and Parallelism Decisions

The thread count logic reveals the careful reasoning that goes into parallel search. The computed `threads` value isn't just "use all cores"—it incorporates multiple constraints that might force single-threaded operation.

Sorting requires single-threaded search because parallel workers would produce results in unpredictable order. Searching a single file doesn't benefit from parallelism—you can't usefully parallelize reading a single sequential input. When neither constraint applies, ripgrep defaults to

available parallelism but caps it at 12 threads. This cap exists because spawning hundreds of threads on a high-core-count machine doesn't help—you become I/O bound long before CPU bound.

The logging call `log::debug!("using {threads} thread(s)")` illustrates how `HiArgs` construction is also a good place for diagnostic output. Since this is where all the decision-making happens, it's the natural place to log what decisions were made and why.

See: Companion Code Section 7

---

## Section 8: Memory Mapping Decisions

The memory mapping configuration demonstrates runtime heuristics based on the search target. Memory-mapped I/O can be faster than regular reads for some workloads, but it's not universally better.

When mmap mode is "auto," ripgrep makes a judgment call: if searching ten or fewer paths and all are regular files, use memory mapping. Otherwise, don't. The reasoning is that memory-mapped I/O shines when reading large files multiple times or when the OS can optimize page-in patterns, but directory traversal involves touching many small files where the mmap overhead isn't worthwhile.

The safety comment about `MmapChoice::auto()` being unsafe is refreshingly honest. Memory mapping has inherent risks—if the underlying file is modified or truncated while being read, the program might crash with SIGBUS. The comment acknowledges this isn't perfectly safe but argues it's acceptable for ripgrep's use case: it never mutates mapped memory, doesn't store mapped contents in structures assuming immutability, and the worst case is a crash rather than data corruption.

See: Companion Code Section 8

---

## Section 9: Sorting Implementation

The `sort()` method shows how `HiArgs` can provide sophisticated functionality to callers. It takes an iterator of haystacks and returns either the same iterator (if no sorting needed) or a boxed iterator that yields items in sorted order.

The implementation handles an important optimization for path sorting. When sorting by path in ascending order, the directory walker can produce results in sorted order directly—so the `sort()`

method simply returns the input iterator unchanged. But for descending order or time-based sorting, it must collect all results, sort them, and return a new iterator.

Time-based sorting uses an interesting pattern: it attaches timestamps to haystacks, sorts by timestamp, then strips the timestamps away. Files that fail to retrieve metadata (perhaps due to permissions or race conditions) are sorted to the end, ensuring they appear last rather than causing errors. This defensive handling means sorting works even when some files can't be fully stat'd.

See: Companion Code Section 9

---

## Section 10: Error Message Enhancement

The suggestion functions (`suggest_other_engine`, `suggest_multiline`, `suggest_text`) demonstrate user-experience thinking embedded in the argument processing code. When regex compilation fails, ripgrep doesn't just show the raw error—it inspects the error message and adds helpful suggestions.

If an error mentions "backreferences" or "look-around," ripgrep suggests PCRE2 because those features aren't supported by the default engine but are by PCRE2. If an error complains about literal newlines, ripgrep suggests the `-U/--multiline` flag. If it mentions NUL bytes being forbidden, ripgrep suggests `-a/--text`.

This pattern of error enhancement shows attention to the user experience. Many users encountering these errors wouldn't immediately know the solution. By detecting common error patterns and suggesting specific remedies, ripgrep transforms cryptic failures into learning opportunities.

See: Companion Code Section 10

---

## Section 11: Environment Interaction

The helper functions at the end of the file handle various environment interactions: getting the current directory, hostname resolution, and WSL detection. Each demonstrates defensive programming against environmental uncertainties.

The `current_dir()` function doesn't just call `std::env::current_dir()`—it has a fallback. If the current directory has been deleted (a real scenario in long-running terminals), it checks the `PWD` environment variable. The error message even suggests the diagnosis: "did your CWD get deleted?"

Hostname resolution goes through multiple fallback layers. First, it tries a user-specified binary command if provided. If that fails or isn't specified, it falls back to platform-specific hostname functions. Each failure is logged at debug level so users can understand why hostname resolution behaved a certain way. This layered approach lets users customize behavior while providing sensible defaults.

See: Companion Code Section 11

---

## Key Takeaways

1. **High-level arguments resolve all interdependencies upfront.** By computing every configuration decision during `HiArgs` construction, the rest of the codebase can simply query boolean fields rather than re-deriving complex logic.
2. **State sharing prevents coordination bugs.** The `State` struct allows separate conversion functions to communicate—tracking whether stdin was consumed or whether stdout is a terminal—without global variables.
3. **Explicit versus implicit paths get different treatment.** Ripgrep tracks how search paths were determined because it affects output formatting and binary detection behavior.
4. **The builder pattern scales to complex configuration.** Methods like `printer_standard()` configure fifteen+ options on a builder, demonstrating how ripgrep's builder ecosystem handles extensive customization without unwieldy constructors.
5. **Thread count depends on multiple factors.** Parallelism decisions incorporate sorting requirements, single-file detection, and reasonable caps—not just available cores.
6. **Error messages are opportunities for user education.** By inspecting error strings and adding contextual suggestions, ripgrep transforms failures into guidance.

---

## What to Read Next

How does ripgrep actually execute searches with these configured components? Read `crates/core/search.rs` to see how `SearchWorker` uses the builders and matchers that `HiArgs` produces.

What happens before argument parsing even begins? Review `crates/core/main.rs` to understand how ripgrep bootstraps and decides which code path to take.

How does the directory walker that `walk_builder()` produces actually traverse directories? Read `ignore/walk.rs` to understand the parallel directory traversal implementation that respects all the gitignore and glob configuration.

# ripgrep crates/core/search.rs: Search Orchestration

## What This File Does

This file defines the search worker abstraction—the component that orchestrates how ripgrep actually executes searches against files. While the grep library handles low-level pattern matching and the printer library handles output formatting, the search worker coordinates these components and adds crucial capabilities like preprocessing external commands, automatic decompression of archives, and binary file detection strategies.

Think of the search worker as a conductor bringing together three musicians: the matcher (which regex engine to use), the searcher (how to read and scan data), and the printer (how to format and display results). The conductor doesn't play any instruments but ensures everyone plays together correctly, handles special arrangements like decompressing files before the performance, and makes decisions about when to skip certain pieces entirely.

---

## Section 1: The Configuration Foundation

The `Config` struct captures the user-facing settings that affect how searches are executed. Unlike low-level searcher configuration (line terminator choices, memory map strategies), these settings control higher-level behaviors that happen *before* or *around* the actual search.

The configuration tracks five distinct concerns. First, the preprocessor path and globs determine whether and which files should be piped through an external command before searching. Second, the `search_zip` flag controls automatic decompression. Third and fourth, two separate binary detection strategies handle files discovered during recursive walks versus files explicitly named by the user.

This separation of binary detection into "implicit" and "explicit" categories reveals thoughtful user experience design. When you run `rg pattern`, files discovered during directory walking can be silently skipped if they appear binary—that's the implicit detection. But if you explicitly type `rg pattern somefile.bin`, you probably know what you're doing, so the explicit detection is typically more permissive. The configuration captures this nuance through two separate fields rather than forcing a one-size-fits-all approach.

See: Companion Code Section 1

---

## Section 2: The Builder Pattern in Action

The `SearchWorkerBuilder` demonstrates ripgrep's commitment to the builder pattern that permeates its entire architecture. This pattern serves multiple purposes: it provides a clear separation between configuration and execution, it enables fluent APIs for setting options, and it allows expensive setup work to happen once at construction time rather than per-search.

Notice how the builder's `new` function immediately configures the `CommandReaderBuilder` with `async_stderr(true)`. This isn't arbitrary—when running preprocessor commands, their stderr output needs to be handled asynchronously to prevent deadlocks. The builder captures this operational wisdom in one place, ensuring every search worker inherits correct behavior.

The `build` method shows a deliberate optimization: the decompression builder is only created when `search_zip` is enabled. The comment explains why—"building the reader can sometimes do non-trivial work (like resolving the paths of decompression binaries on Windows)." This conditional construction saves startup time when users don't need decompression, which is likely the common case.

See: Companion Code Section 2

---

## Section 3: Configuring the Preprocessor Pipeline

The `preprocessor` method illustrates careful handling of external program execution. When given a path to a preprocessor, it doesn't simply store that path—it first resolves the binary using `grep::cli::resolve_binary`. This resolution step ensures the command actually exists and handles platform-specific concerns like finding executables on Windows where `.exe` extensions are implicit.

The method signature returns `anyhow::Result<&mut SearchWorkerBuilder>` rather than just `&mut Self`. This is a departure from typical builder patterns where setters are infallible. By returning a Result, the builder fails fast when given an invalid preprocessor path, rather than deferring the error to search time when debugging becomes harder.

The preprocessor globs add another layer of control. By default, when you specify a preprocessor, every file runs through it. But the globs let you say "only run `.pdf` files through the PDF-to-text

converter." The `should_preprocess` method later checks these globs, using the `is_ignore` result inverted—a file should be preprocessed unless the glob match explicitly ignores it.

See: Companion Code Section 3

---

## Section 4: The PatternMatcher and Printer Enums

Ripgrep supports multiple regex engines—Rust's regex crate by default, and optionally PCRE2 for features like backreferences. The `PatternMatcher` enum abstracts over these implementations, allowing the search worker to operate uniformly regardless of which engine is active.

This enum uses conditional compilation with `#[cfg(feature = "pcre2")]` to include the PCRE2 variant only when that feature is enabled. This means release builds without PCRE2 support don't even contain the code paths for it, reducing binary size and eliminating any runtime overhead from unused features.

The `Printer` enum similarly abstracts over output formats. The three variants—Standard, Summary, and JSON—represent fundamentally different output strategies. Standard produces classic grep-like output with matching lines and context. Summary produces aggregate information like match counts or file lists. JSON emits structured data for machine consumption. Each variant wraps the corresponding type from the printer library.

The generic parameter `W: WriteColor` flows through these types, allowing the printer to work with any color-capable output sink. This abstraction lets the same code write to stdout, files, or in-memory buffers for testing.

See: Companion Code Section 4

---

## Section 5: The SearchWorker Structure

The `SearchWorker` struct brings together everything needed to execute searches. It holds the configuration, the command builder for preprocessors, the optional decompression builder, and the three core components: matcher, searcher, and printer.

The struct is generic over `W`, the writer type, allowing flexibility in where output goes. This generic propagates from the `Printer` enum and ultimately from the underlying printer implementations in the grep library.

The comment on `decomp_builder` deserves attention: it explains why this field is `Option` rather than always present. The builder performs work at construction time, and that work is wasted if decompression is never used. Making it optional is a micro-optimization that also serves as documentation—the presence or absence of this builder communicates whether decompression is enabled.

The struct derives `Clone` and `Debug`, which enables copying workers for use across threads and aids debugging. The clone capability is particularly important because each thread in a parallel search typically gets its own worker instance, avoiding contention on shared state.

See: Companion Code Section 5

---

## Section 6: The Main Search Entry Point

The `search` method is the primary interface for executing searches. It takes a `Haystack` (ripgrep's abstraction over a searchable entity) and returns an `io::Result<SearchResult>`. This method embodies the decision tree for how to search any given target.

The first decision is which binary detection strategy to use. The haystack knows whether it was explicitly specified by the user or discovered during walking, and this information selects between `binary_explicit` and `binary_implicit` detection. The searcher is then configured with the appropriate strategy via `set_binary_detection`.

The logging statement `log::trace!` provides visibility into this decision during debugging. This exemplifies a pattern throughout ripgrep: strategic trace logging that helps diagnose configuration issues without cluttering normal output.

After binary detection configuration, the method branches on how to read the haystack: stdin gets special handling, files matching preprocessor criteria go through the preprocessor, decompressible files get decompressed, and everything else takes the direct path. This priority ordering matters—preprocessor takes precedence over decompression, ensuring user-specified processing overrides automatic handling.

See: Companion Code Section 6

## Section 7: Preprocessing External Commands

The `search_preprocessor` method handles files that should be run through an external command. This enables searching files that aren't plain text—PDFs, Word documents, or proprietary formats—by piping them through a converter first.

The implementation builds a command that receives the file path as an argument and the file's contents on stdin. This dual approach lets preprocessors work however they prefer—some might re-read the file themselves, others might process stdin. The command's stdout becomes the search input.

Error handling here is particularly careful. The command might fail to start (missing binary, permission denied), or it might fail during execution (crash, exit non-zero). The code wraps errors with context indicating which command failed and how, making debugging preprocessor issues tractable.

The explicit ordering of `result?` and `close_result?` after assigning both values ensures that both the search result and the command closure are evaluated, and errors from either propagate appropriately. This pattern prevents resource leaks—the command reader is always closed, even if searching fails.

See: Companion Code Section 7

## Section 8: Automatic Decompression

The `search_decompress` method handles automatic decompression of compressed files. The decompression builder knows which file extensions map to which decompression commands (gzip, bzip2, xz, etc.) and can spawn the appropriate decompressor.

The early return pattern `let Some(ref decomp_builder) = self.decomp_builder else { return self.search_path(path); }` elegantly handles the case where decompression isn't enabled—it falls back to direct path searching rather than failing or doing unnecessary work.

The structure mirrors `search_preprocessor`: build a reader from the path, search through that reader, close the reader, and carefully propagate any errors. This consistency in error handling patterns across methods reduces cognitive load and makes the codebase more maintainable.

Notably, this method assumes `should_decompress` has already returned true. The decompression builder's matcher has already confirmed that this file extension has an associated decompression command. The actual decompression happens transparently through the reader abstraction.

See: Companion Code Section 8

## Section 9: Direct Path Searching

The `search_path` method handles the common case: searching a regular file without any preprocessing or decompression. This is the fast path that benefits from optimizations like memory mapping.

The method must dispatch on the pattern matcher type because the searcher's methods are generic over the `Matcher` trait. Rust's type system requires explicit handling of each enum variant—we can't dynamically dispatch here because the matcher types have different concrete types with different performance characteristics.

The `#[cfg(feature = "pcre2")]` attribute on the PCRE2 arm ensures this code only compiles when PCRE2 support is included. This compile-time feature selection is more robust than runtime feature flags because impossible code paths literally don't exist in the binary.

The method passes searcher, printer, and path to the standalone `search_path` function (note: same name, different scope). This function handles the actual interaction with the searcher and printer, constructing appropriate sinks and collecting results.

See: Companion Code Section 9

## Section 10: Reader-Based Searching

The `search_reader` method handles searching through any `io::Read` implementation rather than directly from a file path. This is used when the "file" content comes from a preprocessor command, a decompression pipe, or stdin.

The method's documentation explains an important trade-off: "Generally speaking, this method should only be used when there is no other choice. Searching via `search_path` provides more opportunities for optimizations (such as memory maps)." Memory mapping allows the kernel to efficiently page file contents into memory, potentially avoiding copies. When reading from a pipe or command output, memory mapping isn't possible.

The implementation mirrors `search_path` structurally, dispatching on matcher type and delegating to the standalone `search_reader` function. This parallel structure between path and reader searching makes the code easier to understand and maintain—changes to one path naturally suggest corresponding changes to the other.

See: Companion Code Section 10

---

## Section 11: The Search Execution Functions

The standalone `search_path` and `search_reader` functions perform the actual search mechanics. They're generic over the matcher type `M: Matcher` and the writer type `W: WriteColor`, allowing them to work with any combination of regex engine and output destination.

Each function matches on the printer variant and follows the same pattern: create a sink from the printer, execute the search through that sink, and construct a `SearchResult` from the sink's accumulated state. The sink abstraction (from the printer library) collects matches and statistics during the search, then reports them afterward.

The `sink_with_path` method on each printer type creates a sink that knows the file path for display purposes. This is crucial for output formatting—users expect to see which file each match came from.

The repetition across printer variants might seem like it could be abstracted, but each printer type's sink has a different concrete type. Rust's type system makes this abstraction difficult without boxing (runtime overhead) or complex trait gymnastics (code complexity). The explicit matching is straightforward and the compiler can optimize each path independently.

See: Companion Code Section 11

---

## Section 12: The SearchResult Aggregate

The `SearchResult` struct captures the outcome of a search operation. It's deliberately minimal: a boolean indicating whether any match was found, and optionally detailed statistics.

The `has_match` field enables early termination in some scenarios—if we only care whether a file matches, we can stop searching after the first match without accumulating statistics. The separation of this field from statistics means callers can get a quick answer without paying for detailed metrics.

Statistics are optional because computing them has runtime cost. The JSON printer always collects statistics (hence `Some(sink.stats().clone())` without the `map`), but Standard and Summary printers only collect them when explicitly configured. This design lets users trade performance for information when needed.

The `#[derive(Clone, Debug, Default)]` on `SearchResult` makes it easy to work with: clone for threading scenarios, debug for logging, and default for initializing aggregate collectors that sum results from multiple searches.

See: Companion Code Section 12

---

## Key Takeaways

First, the search worker abstraction separates orchestration from execution—it coordinates the matcher, searcher, and printer without implementing search logic itself. This separation of concerns makes each component independently testable and replaceable.

Second, the builder pattern enables deferred and conditional construction. The decompression builder is only created when needed, and configuration errors surface at build time rather than during search execution.

Third, implicit versus explicit binary detection recognizes that user intent matters. Files discovered during walks deserve different treatment than files explicitly named on the command line.

Fourth, preprocessor and decompression support transform ripgrep from a text-only tool into one that can search structured content and archives, dramatically expanding its utility without complicating the core search logic.

Fifth, the enum-based dispatch for matchers and printers keeps the code explicit and enables compile-time feature selection. The PCRE2 code literally doesn't exist when that feature is disabled.

Sixth, consistent error handling patterns—building readers, executing searches, closing readers, propagating both results—prevent resource leaks and provide clear error context for debugging.

---

## What to Read Next

How does the searcher actually find matches in file content? Read `grep/searcher/mod.rs` to understand memory mapping strategies, line detection, and the sink abstraction.

What determines which files get searched in the first place? Read `ignore/walk.rs` to see how directory walking, gitignore integration, and file filtering work.

How do the different printers format their output? Read `grep/printer/standard.rs` for the classic grep format, or `grep/printer/json.rs` for structured output.

How does the application coordinate multiple search workers in parallel? Look at how `main.rs` and the argument handling code create and distribute workers across threads.

# ripgrep crates/core/haystack.rs: Haystack Discovery

## What This File Does

This file defines the abstraction ripgrep uses to represent "things we want to search." A haystack—the metaphorical place where you look for needles—wraps directory entries with application-level filtering logic that determines what actually gets searched versus what gets skipped. While the `ignore` crate handles gitignore rules and pattern matching, this file answers a different question: given something that passed all the ignore filters, should we actually try to search it?

The file implements a classic builder pattern with just two types: `HaystackBuilder` for configuration and construction, and `Haystack` for the resulting searchable entity. This is one of the simpler builder implementations in ripgrep, making it an excellent case study for understanding how the pattern works when you only have one configuration option. Despite its simplicity, the logic here embodies important decisions about how ripgrep treats user-provided paths differently from discovered paths, and how it handles edge cases like symlinks and special file types.

---

## Section 1: The Builder Pattern at Its Simplest

The `HaystackBuilder` demonstrates the builder pattern stripped to its essence. Unlike the complex builders we've seen in the searcher or printer systems with dozens of configuration options, this builder has exactly one: whether to strip the `./` prefix from paths. This minimalism makes it perfect for understanding the pattern's core mechanics.

The builder pattern in Rust serves a specific purpose: it lets you construct complex objects step by step while keeping the final object immutable and the construction process clear. Here, even though we only have one option, the pattern still pays dividends. The builder owns the configuration state, creates haystacks with that configuration baked in, and can be reused to create many haystacks with the same settings. If ripgrep later needed to add more haystack configuration options, the pattern is already in place.

Notice that the builder is marked `pub(crate)`, making it visible throughout the `core` crate but not to external consumers. This is a common pattern in Rust library design—you expose a clean public API while keeping internal construction details hidden. The search orchestration code can create

and configure haystack builders, but the details of how haystacks get constructed remain an implementation detail.

See: Companion Code Section 1

---

## Section 2: Constructing from Fallible Results

The `build_from_result` method shows a common Rust pattern for handling fallible operations during iteration. When walking a directory tree, the `ignore` crate yields `Result<DirEntry, Error>` because filesystem operations can fail at any time—permissions might be denied, paths might become invalid, or the filesystem state might change during traversal.

Rather than bubbling these errors up and stopping the search, ripgrep takes a pragmatic approach: log the error and move on. The `err_message!` macro emits a message to the user (if error output is configured), and the method returns `None` to signal "nothing to search here." This design reflects ripgrep's philosophy that a search tool should be resilient—one unreadable file shouldn't abort searching thousands of others.

This pattern separates concerns cleanly. The `ignore` crate handles the walking and filtering, potentially producing errors. The haystack builder handles the error logging and conversion to the application's type. The caller—typically the search loop—just needs to filter out the `None` values and search the rest. Each layer does one thing well.

See: Companion Code Section 2

---

## Section 3: The Filtering Decision Tree

The `build` method contains the core logic that determines whether a directory entry becomes a searchable haystack. This decision tree reflects careful thought about user expectations and edge cases. Let's walk through the logic step by step.

First, the method wraps the directory entry in a `Haystack` struct immediately. This might seem premature—why wrap it before deciding if we'll use it?—but it lets us use the `Haystack`'s helper methods (`is_explicit`, `is_file`, `is_dir`) during the decision process. This is a common Rust pattern: create the object early to leverage its methods, then decide whether to return it.

Second, any errors attached to the directory entry get logged via `ignore_message!`. These are partial errors—the entry exists, but something went wrong reading its metadata, for example. Again, we log but don't fail; resilience trumps strictness.

The decision logic then follows a priority order: explicit entries always win, regular files pass through, and everything else gets rejected. This ordering matters. If a user explicitly names a symlink on the command line, ripgrep will search it even though symlinks discovered during traversal get skipped. User intent takes precedence over default safety.

See: Companion Code Section 3

---

## Section 4: Explicit vs Discovered Paths

The distinction between explicit and discovered haystacks is subtle but crucial for understanding ripgrep's behavior. An explicit haystack is something the user directly specified—either stdin or a path they typed on the command line. A discovered haystack is something ripgrep found while walking directories.

The `is_explicit` method uses a clever heuristic: if the directory entry has depth zero, it was provided directly to the directory iterator rather than discovered during traversal. Combined with checking that it's not a directory (we never search directories themselves) and handling stdin as a special case, this correctly identifies user-provided paths.

There's an important caveat documented in the code: shell globbing happens before ripgrep sees the arguments. When you type `rg pattern ./some-dir/*`, your shell expands that glob into individual paths. Ripgrep sees each expanded path as an explicit argument, not as something it discovered. This means shell globs effectively override some of ripgrep's filtering—a reasonable behavior since the user explicitly asked for those paths, even if indirectly through glob expansion.

This explicit/discovered distinction cascades through ripgrep's behavior. Explicit haystacks bypass certain filters, get different treatment in error reporting, and can include things like symlinks that would otherwise be skipped. It's a simple concept with important UX implications.

See: Companion Code Section 4

---

## Section 5: File Type Detection and Symlink Handling

The `is_dir` and `is_file` methods reveal how ripgrep handles the complexity of Unix file types. On Unix systems, a "file" isn't just a file—it could be a regular file, a directory, a symlink, a socket, a named pipe, a block device, or a character device. Ripgrep needs to search regular files and handle symlinks thoughtfully while ignoring everything else.

The `is_file` method is straightforward: check if the file type indicates a regular file. But `is_dir` is more interesting because it explicitly handles symlinks. A symlink might point to a directory, and in some contexts we need to know whether following the symlink would give us a directory. The method first checks if the entry is directly a directory, then checks if it's a symlink pointing to a directory.

This symlink-aware directory checking matters for the filtering logic. When ripgrep is configured to follow symlinks, the `ignore` crate has already resolved them during traversal. But when checking explicit paths, we might encounter symlinks that weren't pre-resolved, and we need to handle them correctly. The method uses `path_is_symlink()` from the directory entry (a cached value) combined with `path().is_dir()` (which follows symlinks) to make this determination efficiently.

See: Companion Code Section 5

---

## Section 6: Path Presentation and User Experience

The `path` method on `Haystack` handles a small but noticeable UX detail: stripping the `./` prefix from paths when searching the current directory. When you run `rg pattern` without arguments, ripgrep searches the current directory. Internally, this means paths start with `./`, but users generally don't want to see `./src/main.rs` in output—they want `src/main.rs`.

The implementation uses `strip_prefix`, one of Rust's standard library methods for path manipulation. The `unwrap()` here is safe because we've already checked that the path starts with `./` using `starts_with`. This is a common Rust pattern: check a condition with a fallible method, then use an infallible method that you know will succeed. Some developers prefer using `expect()` with a message even in these "proven safe" cases, but here the logic is clear enough that a bare `unwrap()` is acceptable.

This prefix stripping is controlled by the builder, and it only applies when the original builder was configured to strip prefixes. This configuration flows from the command-line argument handling—when no explicit paths are given and ripgrep defaults to searching the current directory, it enables prefix stripping. When users explicitly say `rg pattern .`, they've asked for the current directory by name, so the `./` prefix stays.

See: Companion Code Section 6

---

## Section 7: The Haystack Struct Design

The `Haystack` struct itself is remarkably simple: it wraps an `ignore::DirEntry` and carries a boolean configuration flag. This design embodies the thin wrapper pattern—adding just enough logic to adapt an external type to your application's needs without duplicating or obscuring the underlying functionality.

The struct derives `Clone` and `Debug`, both standard for data types in Rust. `Clone` enables haystacks to be duplicated when needed (though in practice they're often consumed), and `Debug` allows them to be printed for logging and debugging. The struct is `pub(crate)`, like the builder, keeping it internal to the core crate.

Storing `strip_dot_prefix` in every haystack might seem wasteful—couldn't we pass this as a parameter to `path()`? But storing it keeps the API simple. Code that holds a haystack can call `path()` without needing to know or track whether prefix stripping was configured. The configuration decision is made once, at construction time, and then honored automatically. This is the builder pattern's benefit: encapsulate configuration decisions so users of the built object don't need to track them.

See: Companion Code Section 7

---

## Section 8: Integration with the Ignore Crate

Understanding `Haystack` requires understanding its relationship with `ignore::DirEntry`. The `ignore` crate—another Burntsushi project—provides the directory walking and ignore-pattern functionality that ripgrep depends on. Its `DirEntry` type is similar to `std::fs::DirEntry` but enhanced with additional information like depth in the directory tree, whether the entry is stdin, and any errors encountered during traversal.

The haystack module doesn't duplicate this functionality; it delegates to it. Methods like `is_stdin()`, `depth()`, `file_type()`, and `path()` all come from the underlying `DirEntry`. The haystack adds application-level concerns: Should this particular entry be searched given ripgrep's rules about explicit paths, file types, and symlinks?

This layered design is characteristic of well-structured Rust projects. The `ignore` crate handles cross-cutting concerns (directory walking, gitignore patterns) that could be useful to many tools. The `haystack` module handles ripgrep-specific concerns (the explicit/discovered distinction, path presentation). Each layer has clear responsibilities, and changes to one don't ripple unnecessarily through the other.

See: Companion Code Section 8

## Section 9: Error Handling Philosophy

This file demonstrates ripgrep's consistent approach to error handling: log and continue rather than fail fast. Both `build_from_result` and `build` log errors but return `None` rather than propagating errors upward. This makes the search loop simpler—it just filters out `None` values—and keeps ripgrep resilient when encountering filesystem issues.

The two macros used for logging, `err_message!` and `ignore_message!`, represent different severity levels. The `err_message!` macro handles outright failures—a directory entry that couldn't be created at all. The `ignore_message!` macro handles partial failures—an entry that exists but has some issue (perhaps unreadable metadata). Both respect ripgrep's output configuration, potentially suppressing messages when quiet mode is enabled.

The debug logging at the end of `build` serves a different purpose: helping developers understand why files are being skipped. This log line only appears in debug builds or when debug logging is enabled, and it only fires for entries that aren't directories (since skipping directories is expected and not worth logging). It outputs the file type and metadata, giving developers the information they need to diagnose unexpected behavior.

See: Companion Code Section 9

---

## Section 10: Connection to the Builder Pattern Ecosystem

This haystack builder is the simplest builder in ripgrep's ecosystem, but it follows the same patterns we've seen in more complex builders. The searcher builder in `grep-searcher` has dozens of options but the same basic structure: a builder struct with configuration fields, methods that modify those fields and return `&mut Self` for chaining, and a final method that produces the configured object.

The `strip_dot_prefix` method shows the standard builder method signature: it takes `&mut self`, modifies internal state, and returns `&mut Self`. This enables method chaining like `builder.strip_dot_prefix(true).build(dent)`, though in this case with only one configuration option, chaining is less useful. What matters is that the pattern is consistent—anyone familiar with ripgrep's other builders immediately understands how to use this one.

The builder pattern also enables future extensibility. If ripgrep needed to add more haystack configuration—perhaps options for handling special file types or additional path transformations—

the builder structure is already in place. New options would just mean new fields and new configuration methods. The existing `build` method would incorporate the new configuration, and all existing code using the builder would continue to work with the default values.

See: Companion Code Section 10

---

## Key Takeaways

First, the builder pattern scales gracefully from simple to complex use cases. Even with just one configuration option, `HaystackBuilder` provides clear construction semantics and room for future growth.

Second, the distinction between explicit and discovered paths is a powerful UX concept. Users expect explicit requests to override default behavior, and ripgrep's depth-based heuristic captures this intent elegantly.

Third, thin wrappers add value by encapsulating application-specific concerns without duplicating underlying functionality. `Haystack` adds ripgrep logic on top of `ignore::DirEntry` without reimplementing directory entry functionality.

Fourth, resilient error handling—log and continue—is often better for command-line tools than fail-fast approaches. One problematic file shouldn't abort searching thousands of others.

Fifth, path presentation details like stripping `./` prefixes are small touches that significantly improve user experience. Configuration at build time means this detail is handled automatically thereafter.

Sixth, symlink handling requires explicit attention in filesystem code. The `is_dir` method's symlink-aware logic shows how to handle these cases correctly without making the code overly complex.

---

## What to Read Next

How does ripgrep walk directories and apply ignore patterns? Read `crates/ignore/walk.rs` to understand the `DirEntry` type that haystacks wrap and how the directory traversal actually works.

How does the search loop consume haystacks? Revisit `crates/core/search.rs` to see how haystacks flow through the search process and connect to the searcher and printer components.

What are those logging macros doing? Read `crates/core/messages.rs` to understand ripgrep's message infrastructure and how `err_message!` and `ignore_message!` handle output configuration.

How do the command-line arguments determine whether to strip the dot prefix? Review `crates/core/flags/hiargs.rs` to trace how path configuration flows from flags to the haystack builder.

# ripgrep crates/grep/src/lib.rs: Code Companion

Reference code for the The Facade Pattern lecture. Sections correspond to the lecture document.

---

## Section 1: The Module Documentation Block

```
/*!
ripgrep, as a library.

This library is intended to provide a high level facade to the crates that
make up ripgrep's core searching routines. However, there is no high level
documentation available yet guiding users on how to fit all of the pieces
together.

Every public API item in the constituent crates is documented, but examples
are sparse.

A cookbook and a guide are planned.
*/
```

The `/*!` syntax creates an "inner" doc comment that documents the enclosing item (the crate itself) rather than the following item. This becomes the crate's root documentation page. Compare with `///` which documents the item that follows it.

---

## Section 2: Public Re-exports as Architecture

```
// Each `pub extern crate` makes an external crate available as a submodule
// The `as` clause renames the crate for cleaner access

pub extern crate grep_cli as cli;       // grep::cli
pub extern crate grep_matcher as matcher; // grep::matcher
pub extern crate grep_printer as printer; // grep::printer
pub extern crate grep_regex as regex;     // grep::regex
pub extern crate grep_searcher as searcher; // grep::searcher
```

The renaming strips the `grep_` prefix, transforming ecosystem-safe crate names into ergonomic module names. Users write `grep::searcher::Searcher` instead of `grep_searcher::Searcher`.

## Section 3: Conditional Compilation with Feature Flags

```
// This re-export only exists when the "pcre2" feature is enabled
// in Cargo.toml: grep = { version = "...", features = ["pcre2"] }
#[cfg(feature = "pcre2")]
pub extern crate grep_pcre2 as pcre2;
```

The `#[cfg(...)]` attribute performs compile-time conditional inclusion. When `pcre2` feature is disabled, this line is completely removed from compilation—no dead code, no unused dependency.

## Section 4: The Crate Dependency Graph

```
// Core abstraction - defines what a "matcher" must do
pub extern crate grep_matcher as matcher;

// Implementations of the Matcher trait
pub extern crate grep_regex as regex;     // Pure Rust regex engine
#[cfg(feature = "pcre2")]
pub extern crate grep_pcre2 as pcre2;     // C library wrapper

// Orchestration and output
pub extern crate grep_searcher as searcher; // File reading + matching
pub extern crate grep_printer as printer;   // Result formatting

// CLI utilities
pub extern crate grep_cli as cli;           // Terminal handling, etc.
```

The dependency flow: `matcher` defines traits → `regex`/`pcre2` implement them → `searcher` uses matchers → `printer` formats results → `cli` handles terminal concerns.

## Section 5: The Builder Pattern Connection

```
// Through the facade, users access builders from various crates:

// From grep::searcher (grep_searcher crate)
// SearcherBuilder::new().line_terminator(b'\n').build()

// From grep::printer (grep_printer crate)
// StandardBuilder::new().color_specs(specs).build(writer)

// From grep::regex (grep_regex crate)
// RegexMatcherBuilder::new().case_insensitive(true).build(pattern)

// The facade makes these discoverable under one namespace
pub extern crate grep_searcher as searcher;
pub extern crate grep_printer as printer;
pub extern crate grep_regex as regex;
```

Each builder follows Rust conventions: `new()` → chainable configuration methods → `build()` to produce the final type. The facade groups these related APIs.

## Section 6: Why Not Just One Big Crate?

```
// Users who only need regex matching can depend on just:
// [dependencies]
// grep_matcher = "..."
// grep_regex = "..."

// Users who want everything use the facade:
// [dependencies]
// grep = "..."

// The facade provides the unified interface:
pub extern crate grep_cli as cli;
pub extern crate grep_matcher as matcher;
#[cfg(feature = "pcre2")]
pub extern crate grep_pcre2 as pcre2;
pub extern crate grep_printer as printer;
pub extern crate grep_regex as regex;
pub extern crate grep_searcher as searcher;
```

Separate crates enable parallel compilation, independent versioning, and selective dependency. The facade provides convenience without sacrificing these benefits.

## Quick Reference

## Re-export Syntax Comparison

| Syntax | Effect | Use Case |
|---|---|---|
| `pub extern crate foo` | Re-exports crate as submodule | Facade patterns |
| `pub extern crate foo as bar` | Re-exports with rename | Cleaner API names |
| `pub use foo::*` | Re-exports all items | Flattening APIs |
| `pub use foo::Item` | Re-exports specific item | Selective exposure |

## Crate Responsibilities

| Module | Crate | Purpose |
|---|---|---|
| `grep::cli` | `grep_cli` | Terminal utilities |
| `grep::matcher` | `grep_matcher` | Core `Matcher` trait |
| `grep::regex` | `grep_regex` | Rust regex implementation |
| `grep::pcre2` | `grep_pcre2` | PCRE2 implementation (optional) |
| `grep::searcher` | `grep_searcher` | File searching coordination |
| `grep::printer` | `grep_printer` | Output formatting |

## Feature Flag Pattern

```
// In Cargo.toml:
[features]
default = []
pcre2 = ["grep_pcre2"]

// In lib.rs:
#[cfg(feature = "pcre2")]
pub extern crate grep_pcre2 as pcre2;
```

# ripgrep crates/grep/src/lib.rs: Code Companion

Reference code for the The Umbrella Crate lecture. Sections correspond to the lecture document.

---

## Section 1: What This File Does

```
// The entire lib.rs file - remarkably minimal for such an important role
/*!
ripgrep, as a library.
...
*/

pub extern crate grep_cli as cli;
pub extern crate grep_matcher as matcher;
#[cfg(feature = "pcre2")]
pub extern crate grep_pcre2 as pcre2;
pub extern crate grep_printer as printer;
pub extern crate grep_regex as regex;
pub extern crate grep_searcher as searcher;
```

The entire file is under 20 lines of actual code. Its sole purpose is organizational—providing a unified namespace for six specialized crates.

---

## Section 2: The Umbrella Crate Pattern

```
// Without the umbrella pattern, users would need:
use grep_cli;
use grep_matcher;
use grep_printer;
use grep_regex;
use grep_searcher;

// With the umbrella pattern, users get a clean hierarchy:
use grep::cli;
use grep::matcher;
use grep::printer;
use grep::regex;
use grep::searcher;
```

The `as` clause in each re-export strips the `grep_` prefix, transforming `grep_matcher` into simply `matcher` within the `grep` namespace.

---

## Section 3: Module Documentation and the Honesty of Incomplete APIs

```
/*!
ripgrep, as a library.

This library is intended to provide a high level facade to the crates that
make up ripgrep's core searching routines. However, there is no high level
documentation available yet guiding users on how to fit all of the pieces
together.

Every public API item in the constituent crates is documented, but examples
are sparse.

A cookbook and a guide are planned.
*/
```

The `/*! ... */` syntax creates an inner doc comment—documentation that describes the containing module rather than the following item. This text appears at the top of the crate's generated documentation.

---

## Section 4: The Six Pillars of ripgrep Search

```
// CLI utilities for search tool development
pub extern crate grep_cli as cli;

// The core Matcher trait - the abstraction all pattern engines implement
pub extern crate grep_matcher as matcher;
```

These two crates handle environment integration (`cli`) and define the central abstraction (`matcher`) that enables ripgrep's pluggable regex engine architecture.

---

## Section 5: Regex Engines: Default and Optional

```
// Always included: Rust's native regex engine
pub extern crate grep_regex as regex;

// Conditionally compiled: only present if pcre2 feature is enabled
#[cfg(feature = "pcre2")]
pub extern crate grep_pcre2 as pcre2;
```

The `#[cfg(feature = "pcre2")]` attribute means this entire line is removed from compilation when the feature is disabled. The resulting binary has no PCRE2 code at all—not disabled code, absent code.

---

## Section 6: The Searcher and Printer: Orchestrating Results

```
// Coordinates reading content and applying matchers
pub extern crate grep_searcher as searcher;

// Handles output formatting (standard, JSON, colorized)
pub extern crate grep_printer as printer;
```

The searcher handles the "how to search" (line iteration, context tracking, memory mapping), while the printer handles "how to display" (formatting, colorization, structured output).

---

## Section 7: The pub extern crate Pattern

```
// Full syntax breakdown:
pub                     // Make visible to downstream crates
extern crate            // Bring in an external crate
grep_matcher            // The crate name from Cargo.toml
as matcher;             // Rename for public API

// This allows downstream users to write:
use grep::matcher::Matcher;  // Clean namespace

// Instead of:
use grep::grep_matcher::Matcher;  // Redundant prefix
```

In Rust 2018+, `extern crate` is rarely needed except for this re-export pattern. The `pub` visibility combined with `as` renaming creates a stable public interface independent of internal crate naming.

# Quick Reference

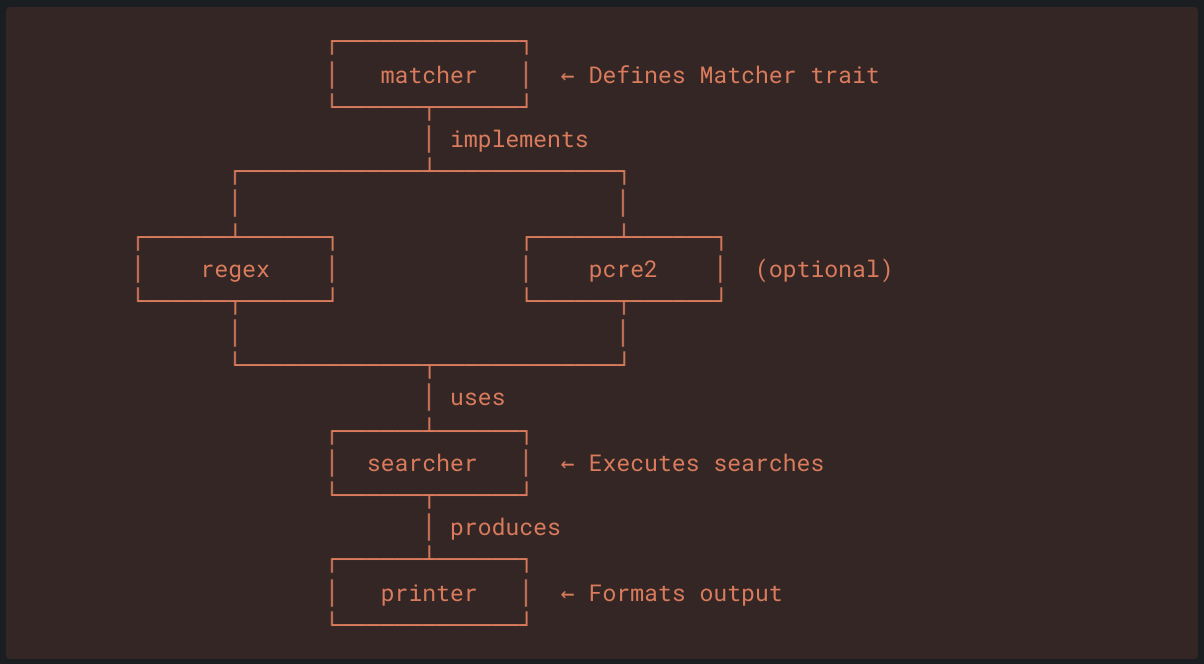
## Crate Mapping

| Internal Crate | Public API | Purpose |
|---|---|---|
| `grep_cli` | `grep::cli` | CLI environment utilities |
| `grep_matcher` | `grep::matcher` | Core `Matcher` trait definition |
| `grep_regex` | `grep::regex` | Rust regex engine adapter |
| `grep_pcre2` | `grep::pcre2` | PCRE2 engine adapter (optional) |
| `grep_searcher` | `grep::searcher` | Search execution logic |
| `grep_printer` | `grep::printer` | Output formatting |

## Feature Flags

```
# In Cargo.toml of a crate depending on grep:
[dependencies]
grep = { version = "x.y", features = ["pcre2"] }  # Enable PCRE2 support
```

## Search Pipeline Architecture


matcher
← Defines Matcher trait
implements
regex
pcre2
(optional)
uses
searcher
← Executes searches
produces
printer
← Formats output

# ripgrep crates/core/main.rs: Code Companion

Reference code for the Application Entry Point lecture. Sections correspond to the lecture document.

---

## Section 1: The Allocator Override

```
// Conditional compilation: only applies when ALL conditions are true
// - target_env = "musl" (building against musl libc)
// - target_pointer_width = "64" (64-bit architecture)
#[cfg(all(target_env = "musl", target_pointer_width = "64"))]
#[global_allocator]  // Replaces default allocator for entire program
static ALLOC: tikv_jemallocator::Jemalloc = tikv_jemallocator::Jemalloc;
```

The `#[global_allocator]` attribute designates this static as the memory allocator for all heap allocations. When the `#[cfg]` conditions aren't met, this code doesn't exist in the compiled binary at all—not disabled, literally absent.

---

## Section 2: The Main Function and Exit Code Philosophy

```rust
use std::process::ExitCode;

/// Then, as it was, then again it will be.
fn main() -> ExitCode {
    match run(flags::parse()) {
        Ok(code) => code,
        Err(err) => {
            // Walk the error chain looking for broken pipe
            for cause in err.chain() {
                if let Some(ioerr) = cause.downcast_ref::<std::io::Error>() {
                    if ioerr.kind() == std::io::ErrorKind::BrokenPipe {
                        // Unix convention: broken pipe = graceful exit
                        return ExitCode::from(0);
                    }
                }
            }
            // eprintln_locked! ensures thread-safe error output
            eprintln_locked!("{:#}", err);  // {:#} = alternate Display format
            ExitCode::from(2)  // Exit code 2 = error condition
        }
    }
}
```

The `err.chain()` method comes from `anyhow::Error` and iterates through the entire error cause chain. The `downcast_ref` attempts to view the error as a specific type. Exit code 2 indicates an error (distinct from exit code 1, which means "no matches found").

---

## Section 3: The Run Function and Mode Dispatch

```rust
fn run(result: crate::flags::ParseResult<HiArgs>) -> anyhow::Result<ExitCode> {
    use crate::flags::{Mode, ParseResult};

    // Three-way split: errors, special modes, normal operation
    let args = match result {
        ParseResult::Err(err) => return Err(err),
        ParseResult::Special(mode) => return special(mode),  // --help, --version
        ParseResult::Ok(args) => args,
    };

    // Mode dispatch with optimization checks
    let matched = match args.mode() {
        // Early exit if pattern can never match
        Mode::Search(_) if !args.matches_possible() => false,
        // Single-threaded vs parallel based on thread count
        Mode::Search(mode) if args.threads() == 1 => search(&args, mode)?,
        Mode::Search(mode) => search_parallel(&args, mode)?,
        // File listing modes (--files flag)
        Mode::Files if args.threads() == 1 => files(&args)?,
        Mode::Files => files_parallel(&args)?,
        // Informational modes
        Mode::Types => return types(&args),
        Mode::Generate(mode) => return generate(mode),
    };

    // Exit code logic: 0 = matches found, 1 = no matches, 2 = error
    Ok(if matched && (args.quiet() || !messages::errored()) {
        ExitCode::from(0)
    } else if messages::errored() {
        ExitCode::from(2)
    } else {
        ExitCode::from(1)
    })
}
```

The `matches_possible()` check handles edge cases like patterns longer than any possible line. The thread count check ensures sorted output uses single-threaded mode (parallelism would destroy sort order).

## Section 4: Single-Threaded Search Architecture

```
fn search(args: &HiArgs, mode: SearchMode) -> anyhow::Result<bool> {
    let started_at = std::time::Instant::now();
    let haystack_builder = args.haystack_builder();

    // Build iterator of searchable items, filtering invalid results
    let unsorted = args
        .walk_builder()?
        .build()
        .filter_map(|result| haystack_builder.build_from_result(result));
    let haystacks = args.sort(unsorted);  // Apply sorting if requested

    let mut matched = false;
    let mut searched = false;
    let mut stats = args.stats();  // Option<Stats> - only if --stats flag

    // Compose the three search components: matcher + searcher + printer
    let mut searcher = args.search_worker(
        args.matcher()?,      // Pattern matching engine (regex or PCRE2)
        args.searcher()?,     // Line-oriented search logic
        args.printer(mode, args.stdout()),  // Output formatter
    )?;

    for haystack in haystacks {
        searched = true;
        let search_result = match searcher.search(&haystack) {
            Ok(search_result) => search_result,
            // Broken pipe = user closed output (e.g., `rg pattern | head`)
            Err(err) if err.kind() == std::io::ErrorKind::BrokenPipe => break,
            Err(err) => {
                // Log error but continue searching other files
                err_message!("{}: {}", haystack.path().display(), err);
                continue;
            }
        };
        matched = matched || search_result.has_match();
        if let Some(ref mut stats) = stats {
            *stats += search_result.stats().unwrap();
        }
        // Support -q/--quiet mode: exit after first match
        if matched && args.quit_after_match() {
            break;
        }
    }

    // Helpful diagnostic when ignore rules filter everything
    if args.has_implicit_path() && !searched {
        eprint_nothing_searched();
    }
    // Print accumulated statistics if requested
    if let Some(ref stats) = stats {
        let wtr = searcher.printer().get_mut();
        let _ = print_stats(mode, stats, started_at, wtr);
    }
    Ok(matched)
```

```
}
```

The "haystack" abstraction represents anything searchable. The `filter_map` chain converts directory walk results into haystacks, filtering out entries that can't be searched (directories, symlinks when not following them, etc.).

---

## Section 5: Parallel Search and Thread Coordination

```rust
fn search_parallel(args: &HiArgs, mode: SearchMode) -> anyhow::Result<bool> {
    use std::sync::atomic::{AtomicBool, Ordering};

    let started_at = std::time::Instant::now();
    let haystack_builder = args.haystack_builder();
    let bufwtr = args.buffer_writer();  // Thread-safe buffered output

    // Mutex for complex state, AtomicBool for simple flags
    let stats = args.stats().map(std::sync::Mutex::new);
    let matched = AtomicBool::new(false);
    let searched = AtomicBool::new(false);

    // Create template searcher that will be cloned per-thread
    let mut searcher = args.search_worker(
        args.matcher()?,
        args.searcher()?,
        args.printer(mode, bufwtr.buffer()),  // Each clone gets own buffer
    )?;

    // Parallel walker handles thread spawning and work distribution
    args.walk_builder()?.build_parallel().run(|| {
        // Capture shared state by reference
        let bufwtr = &bufwtr;
        let stats = &stats;
        let matched = &matched;
        let searched = &searched;
        let haystack_builder = &haystack_builder;
        let mut searcher = searcher.clone();  // Clone for this thread

        // Return boxed closure - called once per discovered file
        Box::new(move |result| {
            let haystack = match haystack_builder.build_from_result(result) {
                Some(haystack) => haystack,
                None => return WalkState::Continue,
            };
            searched.store(true, Ordering::SeqCst);
            searcher.printer().get_mut().clear();  // Reset buffer

            let search_result = match searcher.search(&haystack) {
                Ok(search_result) => search_result,
                Err(err) => {
                    err_message!("{}: {}", haystack.path().display(), err);
                    return WalkState::Continue;
                }
            };

            if search_result.has_match() {
                matched.store(true, Ordering::SeqCst);
            }
            // Accumulate stats under mutex
            if let Some(ref locked_stats) = *stats {
                let mut stats = locked_stats.lock().unwrap();
                *stats += search_result.stats().unwrap();
            }
```

```
            // Print buffer atomically            if let Err(err) =
bufwtr.print(searcher.printer().get_mut()) {                if err.kind() ==
std::io::ErrorKind::BrokenPipe {                    return WalkState::Quit;  //
Stop all threads                }                err_message!("{}: {}",
haystack.path().display(), err);            }            // WalkState controls
traversal: Continue, Skip, or Quit            if matched.load(Ordering::SeqCst)
&& args.quit_after_match() {                WalkState::Quit            } else {
WalkState::Continue            }        })    });
Ok(matched.load(Ordering::SeqCst))}
```

`SeqCst` (sequentially consistent) ordering provides the strongest memory ordering guarantee, ensuring updates are immediately visible across threads. The `bufwtr.print()` call writes the entire buffer atomically to prevent interleaved output from different threads.

## Section 6: The File Listing Modes

```
/// Single-threaded file listing (--files flag)
fn files(args: &HiArgs) -> anyhow::Result<bool> {
    let haystack_builder = args.haystack_builder();
    let unsorted = args
        .walk_builder()?
        .build()
        .filter_map(|result| haystack_builder.build_from_result(result));
    let haystacks = args.sort(unsorted);

    let mut matched = false;
    let mut path_printer = args.path_printer_builder().build(args.stdout());

    for haystack in haystacks {
        matched = true;
        if args.quit_after_match() {
            break;
        }
        if let Err(err) = path_printer.write(haystack.path()) {
            if err.kind() == std::io::ErrorKind::BrokenPipe {
                break;
            }
            return Err(err.into());
        }
    }
    Ok(matched)
}

/// Parallel file listing with channel-based coordination
fn files_parallel(args: &HiArgs) -> anyhow::Result<bool> {
    use std::{
        sync::{atomic::{AtomicBool, Ordering}, mpsc},
        thread,
    };

    let haystack_builder = args.haystack_builder();
    let mut path_printer = args.path_printer_builder().build(args.stdout());
    let matched = AtomicBool::new(false);
    let (tx, rx) = mpsc::channel::<crate::haystack::Haystack>();

    // Dedicated print thread prevents write tearing
    let print_thread = thread::spawn(move || -> std::io::Result<()> {
        for haystack in rx.iter() {
            path_printer.write(haystack.path())?;
        }
        Ok(())
    });

    args.walk_builder()?.build_parallel().run(|| {
        let haystack_builder = &haystack_builder;
        let matched = &matched;
        let tx = tx.clone();  // Clone sender for each worker

        Box::new(move |result| {
            let haystack = match haystack_builder.build_from_result(result) {
```

```
                Some(haystack) => haystack,                None => return
WalkState::Continue,            };            matched.store(true,
Ordering::SeqCst);            if args.quit_after_match() {
WalkState::Quit            } else {                match tx.send(haystack) {
Ok(_) => WalkState::Continue,                    Err(_) => WalkState::Quit,  //
Receiver dropped                }            }        })    });    drop(tx);  //
Close channel so print_thread's iter() terminates    if let Err(err) =
print_thread.join().unwrap() {        if err.kind() !=
std::io::ErrorKind::BrokenPipe {            return Err(err.into());        }    }
Ok(matched.load(Ordering::SeqCst))}
```

The parallel version uses `mpsc::channel` (multi-producer, single-consumer) where worker threads send discovered paths to a dedicated print thread. This ensures writes don't interleave. The `drop(tx)` is crucial—it closes the sending side so the receiver's `iter()` loop terminates.

---

# Quick Reference

## Exit Code Semantics

| Code | Meaning |
|---|---|
| 0 | Matches found (or `--files` found files) |
| 1 | No matches found |
| 2 | Error occurred |

## Mode Dispatch Flow

```
ParseResult::Err      → Return error immediately
ParseResult::Special  → Handle --help/--version
ParseResult::Ok       → Normal mode dispatch:
                          ├─ Search (threads=1)   → search()
                          ├─ Search (threads>1)   → search_parallel()
                          ├─ Files (threads=1)    → files()
                          ├─ Files (threads>1)    → files_parallel()
                          ├─ Types                → types()
                          └─ Generate             → generate()
```

## Key Types

```rust
// Parse result variants
enum ParseResult<T> {
    Err(anyhow::Error),
    Special(SpecialMode),
    Ok(T),
}

// Walk control for parallel traversal
enum WalkState {
    Continue,  // Keep searching
    Skip,      // Skip this directory's children
    Quit,      // Stop all threads
}

// Memory orderings used
Ordering::SeqCst  // Strongest guarantee, used for cross-thread flags
```

## Synchronization Primitives

| Primitive | Used For | Why |
|---|---|---|
| `AtomicBool` | `matched`, `searched` flags | Simple boolean state, no mutex overhead |
| `Mutex` | Statistics accumulation | Complex `+=` operations need exclusivity |
| `mpsc::channel` | Parallel file listing | Prevents write tearing, clean producer/consumer |
| `BufferWriter` | Parallel search output | Atomic buffer printing |

# ripgrep crates/grep/examples/simplegrep.rs: Code Companion

Reference code for the Minimal Working Example lecture. Sections correspond to the lecture document.

---

## Section 1: The Error Handling Entry Point

```rust
use std::{env, error::Error, ffi::OsString, io::IsTerminal, process};

fn main() {
    // Delegate to try_main and handle errors uniformly
    if let Err(err) = try_main() {
        eprintln!("{}", err);      // Print error to stderr
        process::exit(1);          // Exit with failure code
    }
}

fn try_main() -> Result<(), Box<dyn Error>> {
    // ... actual logic here, can use ? operator freely
}
```

The `Box<dyn Error>` return type accepts any error type implementing the `Error` trait. This is called a **trait object**—the `dyn` keyword indicates dynamic dispatch at runtime.

---

## Section 2: Argument Parsing and Platform Safety

```rust
fn try_main() -> Result<(), Box<dyn Error>> {
    // args_os() returns OsString, not String—handles non-UTF-8 filenames
    let mut args: Vec<OsString> = env::args_os().collect();

    // Require at least: program name + pattern
    if args.len() < 2 {
        return Err("Usage: simplegrep <pattern> [<path> ...]".into());
    }

    // Default to current directory if no paths provided
    if args.len() == 2 {
        args.push(OsString::from("./"));
    }

    // args[0] is program name, args[1] is pattern, args[2..] are paths
    search(cli::pattern_from_os(&args[1])?, &args[2..])
}
```

The `.into()` call on the string literal converts `&str` into `Box<dyn Error>` through Rust's `From` trait implementations. The `?` operator on `pattern_from_os` propagates any conversion error.

## Section 3: Pattern Conversion with cli::pattern_from_os

```rust
use grep::cli;

// In try_main:
search(cli::pattern_from_os(&args[1])?, &args[2..])

// cli::pattern_from_os signature (from grep::cli module):
// pub fn pattern_from_os(pattern: &OsStr) -> Result<&str, Error>
```

This function attempts to interpret the `OsString` as UTF-8. If the pattern contains invalid UTF-8 bytes, it returns an error rather than panicking. The `?` propagates this error to `try_main`'s caller.

## Section 4: The Matcher — Your First Builder Pattern

```rust
fn search(pattern: &str, paths: &[OsString]) -> Result<(), Box<dyn Error>> {
    // Create a matcher optimized for line-oriented searching
    // Pattern never matches across line boundaries
    let matcher = RegexMatcher::new_line_matcher(&pattern)?;

    // ... rest of search function
}
```

`RegexMatcher::new_line_matcher` compiles the pattern and configures it to respect line boundaries. The `?` handles regex syntax errors—an invalid pattern like `[unclosed` would return an error here.

---

## Section 5: The Searcher Builder — Configuration Through Method Chaining

```
use grep::searcher::{BinaryDetection, SearcherBuilder};

// Builder pattern: new() → configure → build()
let mut searcher = SearcherBuilder::new()
    // Stop searching if we hit a null byte (indicates binary file)
    .binary_detection(BinaryDetection::quit(b'\x00'))
    // Don't track line numbers (simpler output, slight perf gain)
    .line_number(false)
    // Consume the builder, produce the Searcher
    .build();
```

The `mut` on `searcher` is required because `search_path()` modifies internal buffers. Each builder method returns `&mut Self`, enabling the fluent chaining syntax.

## Section 6: The Printer Builder — Terminal-Aware Output

```
use std::io::IsTerminal;
use grep::{cli, printer::{ColorSpecs, StandardBuilder}};
use termcolor::ColorChoice;

let mut printer = StandardBuilder::new()
    // Set up default colors for matches, filenames, etc.
    .color_specs(ColorSpecs::default_with_color())
    // build() takes a WriteColor implementation
    .build(cli::stdout(if std::io::stdout().is_terminal() {
        ColorChoice::Auto    // Terminal: let printer decide on colors
    } else {
        ColorChoice::Never   // Pipe/file: no ANSI escape codes
    }));
```

`is_terminal()` (Rust 1.70+) detects whether stdout connects to an interactive terminal. The `cli::stdout()` function wraps the standard output handle with buffering and color support.

## Section 7: Directory Walking with WalkDir

```
use walkdir::WalkDir;

for path in paths {
    // WalkDir recursively yields directory entries
    for result in WalkDir::new(path) {
        let dent = match result {
            Ok(dent) => dent,
            Err(err) => {
                // Permission denied, broken symlink, etc.
                eprintln!("{}", err);
                continue;  // Skip this entry, keep searching
            }
        };

        // Skip directories, symlinks, etc.—only search regular files
        if !dent.file_type().is_file() {
            continue;
        }

        // dent.path() returns &Path for use in search
    }
}
```

`WalkDir` handles recursive traversal including symlinks, permissions, and cross-platform path handling. Each `result` is a `Result<DirEntry, walkdir::Error>`.

## Section 8: The Search Execution — Bringing It Together

```
// Inside the directory walking loop:
let result = searcher.search_path(
    &matcher,                                  // Pattern to match
    dent.path(),                               // File to search
    printer.sink_with_path(&matcher, dent.path()), // Where results go
);

// Handle search errors (I/O errors, encoding issues, etc.)
if let Err(err) = result {
    eprintln!("{}: {}", dent.path().display(), err);
}
```

The **Sink** pattern decouples searching from output. `sink_with_path` creates a sink that knows the filename and matcher—it handles formatting each match and writing to the printer's output.

# Quick Reference

## Component Responsibilities

| Component | Type | Purpose |
|---|---|---|
| Matcher | `RegexMatcher` | Compiles and executes regex pattern |
| Searcher | `Searcher` | Reads files, finds matches, manages buffers |
| Printer | `Standard<W>` | Formats and outputs search results |
| Sink | `StandardSink<...>` | Bridges searcher to printer |

## Builder Pattern Summary

```
// General pattern:
let thing = ThingBuilder::new()
    .option_a(value)
    .option_b(value)
    .build();  // Consumes builder, returns Thing
```

## Key Type Signatures

```
// Matcher construction
RegexMatcher::new_line_matcher(pattern: &str) -> Result<RegexMatcher, Error>

// Searcher execution
Searcher::search_path<M, P, S>(
    &mut self,
    matcher: M,
    path: P,
    sink: S
) -> Result<(), S::Error>
where
    M: Matcher,
    P: AsRef<Path>,
    S: Sink

// Printer sink creation
Standard<W>::sink_with_path<M, P>(
    &self,
    matcher: M,
    path: P
) -> StandardSink<'_, M, W>
```

## Import Summary

```
use std::{env, error::Error, ffi::OsString, io::IsTerminal, process};
use grep::{cli, printer::*, regex::RegexMatcher, searcher::*};
use termcolor::ColorChoice;
use walkdir::WalkDir;
```

# ripgrep crates/core/logger.rs: Code Companion

Reference code for the Logging Infrastructure lecture. Sections correspond to the lecture document.

## Section 1: The Philosophy of Minimal Dependencies

```
/*!
Defines a super simple logger that works with the `log` crate.

We don't do anything fancy. We just need basic log levels and the ability to
print to stderr. We therefore avoid bringing in extra dependencies just for
this functionality.
*/

use log::Log;
```

The module documentation comment (`/*!...*/`) appears at the file's start and documents the module itself. The single import—just the `Log` trait—demonstrates the minimal dependency approach.

## Section 2: The Logger Struct and the Singleton Pattern

```
/// The simplest possible logger that logs to stderr.
///
/// This logger does no filtering. Instead, it relies on the `log` crates
/// filtering via its global max_level setting.
#[derive(Debug)]
pub(crate) struct Logger(());  // Unit struct - zero-sized type

/// A singleton used as the target for an implementation of the `Log` trait.
const LOGGER: &'static Logger = &Logger(());  // Static reference, valid for
program lifetime
```

The `Logger(())` syntax creates a tuple struct containing the unit type `()`. This is a zero-sized type (ZST)—it occupies no memory at runtime. The `const` declaration creates a compile-time constant, not a runtime allocation.

## Section 3: Initialization and the Log Crate's Global State

```
impl Logger {
    /// Create a new logger that logs to stderr and initialize it as the
    /// global logger. If there was a problem setting the logger, then an
    /// error is returned.
    pub(crate) fn init() -> Result<(), log::SetLoggerError> {
        log::set_logger(LOGGER)  // Registers LOGGER as the global logger
    }
}
```

The `log::set_logger` function expects `&'static dyn Log`. Our `LOGGER` constant satisfies this because it's a `&'static Logger` and `Logger` implements `Log`. The function can only succeed once per program execution.

## Section 4: Implementing the Log Trait

```
impl Log for Logger {
    fn enabled(&self, _: &log::Metadata<'_>) -> bool {
        // ...
    }

    fn log(&self, record: &log::Record<'_>) {
        // ...
    }

    fn flush(&self) {
        // ...
    }
}
```

The `Log` trait from the `log` crate defines the contract for logging backends. The lifetime `'_` in `Metadata<'_>` and `Record<'_>` uses lifetime elision—the compiler infers appropriate lifetimes. These are borrowed references to data owned by the `log` crate.

## Section 5: The enabled Method and Deferred Filtering

```rust
fn enabled(&self, _: &log::Metadata<'_>) -> bool {
    // We set the log level via log::set_max_level, so we don't need to
    // implement filtering here.
    true  // Always accept messages that pass the global filter
}
```

The underscore prefix on `_: &log::Metadata<'_>` indicates an intentionally unused parameter. The `Metadata` struct contains the log level and target module, which could be used for fine-grained filtering—but ripgrep relies entirely on the global level instead.

---

## Section 6: The log Method and Pattern Matching on Optional Data

```rust
fn log(&self, record: &log::Record<'_>) {
    // Pattern match on tuple of two Option values
    match (record.file(), record.line()) {
        // Both file and line available - most informative output
        (Some(file), Some(line)) => {
            eprintln_locked!(
                "{}|{}|{}:{}: {}",
                record.level(),    // DEBUG, INFO, WARN, ERROR, TRACE
                record.target(),   // Module path like "ripgrep::search"
                file,              // Source file path
                line,              // Line number
                record.args()      // The formatted message
            );
        }
        // Only file available
        (Some(file), None) => {
            eprintln_locked!(
                "{}|{}|{}: {}",
                record.level(),
                record.target(),
                file,
                record.args()
            );
        }
        // Catch-all: no location info or only line (unusual)
        _ => {
            eprintln_locked!(
                "{}|{}: {}",
                record.level(),
                record.target(),
                record.args()
            );
        }
    }
}
```

The `record.args()` method returns `&fmt::Arguments`, which is the pre-formatted result of the format string and arguments passed to the log macro. The pipe-delimited format creates structured, grep-friendly output.

---

## Section 7: The eprintln_locked Macro

```
fn log(&self, record: &log::Record<'_>) {
    // Uses eprintln_locked! instead of standard eprintln!
    eprintln_locked!(
        "{}|{}|{}:{}: {}",
        // ... arguments ...
    );
}

fn flush(&self) {
    // We use eprintln_locked! which is flushed on every call.
}
```

The `flush` method's empty implementation with explanatory comment shows that `eprintln_locked!` handles flushing automatically. This macro is defined elsewhere in the core crate and ensures atomic multi-threaded output to stderr.

---

## Quick Reference

### Log Trait Methods

| Method | Signature | Purpose |
|---|---|---|
| `enabled` | `fn(&self, &Metadata) -> bool` | Check if a message should be logged |
| `log` | `fn(&self, &Record)` | Write the log message |
| `flush` | `fn(&self)` | Ensure buffered output is written |

### Log Record Fields

| Method | Returns | Description |
|---|---|---|
| `level()` | `Level` | TRACE, DEBUG, INFO, WARN, ERROR |
| `target()` | `&str` | Module path (e.g., `ripgrep::search`) |
| `file()` | `Option<&str>` | Source file where log was called |
| `line()` | `Option<u32>` | Line number in source file |
| `args()` | `&Arguments` | Pre-formatted log message |

## Logger Architecture

```
Code calls debug!("msg")
        ↓
log crate checks global max level
        ↓ (if enabled)
Logger::enabled() → true
        ↓
Logger::log() → eprintln_locked! → stderr
```

## Output Format

```
LEVEL|target|file:line: message
DEBUG|ripgrep::search|src/search.rs:42: searching path
```

# The Matcher Trait: Code Companion

Reference code for the The Matcher Trait lecture. Sections correspond to the lecture document.

---

## Section 1: What This File Does

```
/*!
This crate provides an interface for regular expressions, with a focus on line
oriented search. The purpose of this crate is to provide a low level matching
interface that permits any kind of substring or regex implementation to power
the search routines provided by the
[`grep-searcher`](https://docs.rs/grep-searcher)
crate.

The primary thing provided by this crate is the [`Matcher`] trait. The trait
defines an abstract interface for text search. It is robust enough to support
everything from basic substring search all the way to arbitrarily complex
regular expression implementations without sacrificing performance.

A key design decision made in this crate is the use of *internal iteration*,
or otherwise known as the "push" model of searching. In this paradigm,
implementations of the `Matcher` trait will drive search and execute callbacks
provided by the caller when a match is found.
*/
```

The module-level documentation establishes the crate's purpose and explicitly calls out the internal iteration design choice—a crucial architectural decision explained in the lecture.

## Section 2: The Match Type - A Better Range

```rust
/// The type of a match.
///
/// Every `Match` is guaranteed to satisfy the invariant that `start <= end`.
#[derive(Clone, Copy, Debug, Eq, Hash, PartialEq)]
pub struct Match {
    start: usize,
    end: usize,
}

impl Match {
    /// Create a new match.
    ///
    /// # Panics
    /// This function panics if `start > end`.
    #[inline]
    pub fn new(start: usize, end: usize) -> Match {
        assert!(start <= end);  // Invariant enforced at construction
        Match { start, end }
    }

    /// Creates a zero width match at the given offset.
    #[inline]
    pub fn zero(offset: usize) -> Match {
        Match { start: offset, end: offset }
    }

    /// Return a new match with the start offset replaced.
    /// Maintains the invariant by asserting start <= end.
    #[inline]
    pub fn with_start(&self, start: usize) -> Match {
        assert!(start <= self.end, "{} is not <= {}", start, self.end);
        Match { start, ..*self }  // Functional update syntax
    }

    /// Offset this match by the given amount and return a new match.
    #[inline]
    pub fn offset(&self, amount: usize) -> Match {
        Match {
            start: self.start.checked_add(amount).unwrap(),
            end: self.end.checked_add(amount).unwrap(),
        }
    }

    #[inline]
    pub fn len(&self) -> usize {
        self.end - self.start  // Safe because start <= end
    }
}

// Allows using Match directly as a slice index: &bytes[m]
impl std::ops::Index<Match> for [u8] {
    type Output = [u8];

    #[inline]
```

```
    fn index(&self, index: Match) -> &[u8] {        &self[index.start..index.end]
}}impl std::ops::Index<Match> for str {    type Output = str;    #[inline]    fn
index(&self, index: Match) -> &str {        &self[index.start..index.end]    }}
```

Note the `#[derive(Copy)]`—this is significant because `std::ops::Range<usize>` does not implement `Copy`. The `Index` implementations allow seamless integration with Rust's slice syntax.

---

## Section 3: Line Terminators and Cross-Platform Reality

```rust
/// A line terminator represents the end of a line.
#[derive(Clone, Copy, Debug, Eq, Hash, PartialEq)]
pub struct LineTerminator(LineTerminatorImp);  // Public struct wraps private
enum

#[derive(Clone, Copy, Debug, Eq, Hash, PartialEq)]
enum LineTerminatorImp {
    /// Any single byte representing a line terminator.
    Byte(u8),
    /// A line terminator represented by `\r\n`.
    CRLF,
}

impl LineTerminator {
    /// Return a new single-byte line terminator.
    #[inline]
    pub fn byte(byte: u8) -> LineTerminator {
        LineTerminator(LineTerminatorImp::Byte(byte))
    }

    /// Return a new CRLF line terminator.
    #[inline]
    pub fn crlf() -> LineTerminator {
        LineTerminator(LineTerminatorImp::CRLF)
    }

    /// Returns this line terminator as a single byte.
    ///
    /// If CRLF, returns `\n`. This simplifies line-oriented algorithms
    /// that can treat `\n` as the significant byte in either mode.
    #[inline]
    pub fn as_byte(&self) -> u8 {
        match self.0 {
            LineTerminatorImp::Byte(byte) => byte,
            LineTerminatorImp::CRLF => b'\n',  // Pragmatic simplification
        }
    }

    /// Returns the full byte sequence for this terminator.
    #[inline]
    pub fn as_bytes(&self) -> &[u8] {
        match self.0 {
            LineTerminatorImp::Byte(ref byte) => std::slice::from_ref(byte),
            LineTerminatorImp::CRLF => &[b'\r', b'\n'],
        }
    }
}

impl Default for LineTerminator {
    #[inline]
    fn default() -> LineTerminator {
        LineTerminator::byte(b'\n')  // Unix-style default on all platforms
    }
}
```

The private enum pattern (`LineTerminatorImp`) allows the internal representation to change without breaking API compatibility. Factory methods (`byte()`, `crlf()`) provide the only construction paths.

---

## Section 4: ByteSet - Optimization Through Constraint

```
/// A set of bytes that can never appear in any match.
#[derive(Clone, Debug)]
pub struct ByteSet(BitSet);

#[derive(Clone, Copy)]
struct BitSet([u64; 4]);  // 4 × 64 = 256 bits, one per possible byte value

impl std::fmt::Debug for BitSet {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        let mut fmtd = f.debug_set();
        for b in 0..=255 {
            if ByteSet(*self).contains(b) {
                fmtd.entry(&b);  // Only print bytes that are in the set
            }
        }
        fmtd.finish()
    }
}

impl ByteSet {
    /// Create an empty set of bytes.
    #[inline]
    pub fn empty() -> ByteSet {
        ByteSet(BitSet([0; 4]))
    }

    /// Create a full set (all 256 bytes present).
    #[inline]
    pub fn full() -> ByteSet {
        ByteSet(BitSet([u64::MAX; 4]))
    }

    /// Add a byte to this set.
    #[inline]
    pub fn add(&mut self, byte: u8) {
        let bucket = byte / 64;       // Which u64 (0-3)
        let bit = byte % 64;          // Which bit within that u64
        (self.0).0[usize::from(bucket)] |= 1 << bit;
    }

    /// Remove a byte from this set.
    #[inline]
    pub fn remove(&mut self, byte: u8) {
        let bucket = byte / 64;
        let bit = byte % 64;
        (self.0).0[usize::from(bucket)] &= !(1 << bit);  // Clear the bit
    }

    /// Return true if the byte is in this set.
    #[inline]
    pub fn contains(&self, byte: u8) -> bool {
        let bucket = byte / 64;
        let bit = byte % 64;
        (self.0).0[usize::from(bucket)] & (1 << bit) > 0
```

```
    }}
```

The bit vector representation uses exactly 32 bytes (4 × 8 bytes) to represent any subset of the 256 possible byte values. All operations are O(1) with no branching except for the bucket selection.

---

## Section 5: The Captures Trait - Abstracting Group Extraction

```
/// A trait that describes implementations of capturing groups.
pub trait Captures {
    /// Return the total number of capturing groups.
    fn len(&self) -> usize;

    /// Return the capturing group match at the given index.
    /// Index 0 is always the overall match.
    fn get(&self, i: usize) -> Option<Match>;

    /// Return the overall match (equivalent to get(0).unwrap()).
    #[inline]
    fn as_match(&self) -> Match {
        self.get(0).unwrap()
    }

    /// Expands `$name` references in `replacement` to captured groups.
    /// Default implementation delegates to the interpolate module.
    #[inline]
    fn interpolate<F>(
        &self,
        name_to_index: F,
        haystack: &[u8],
        replacement: &[u8],
        dst: &mut Vec<u8>,
    ) where
        F: FnMut(&str) -> Option<usize>,
    {
        interpolate(
            replacement,
            |i, dst| {
                if let Some(range) = self.get(i) {
                    dst.extend(&haystack[range]);  // Uses Match's Index impl
                }
            },
            name_to_index,
            dst,
        )
    }
}

/// NoCaptures provides an always-empty implementation of `Captures`.
/// Use this for matchers that don't support capturing groups.
#[derive(Clone, Debug)]
pub struct NoCaptures(());  // Zero-sized type

impl NoCaptures {
    #[inline]
    pub fn new() -> NoCaptures {
        NoCaptures(())
    }
}

impl Captures for NoCaptures {
    #[inline]
```

```
    fn len(&self) -> usize {        0  // No captures ever    }    #[inline]
fn get(&self, _: usize) -> Option<Match> {        None  // No captures to get
}}
```

`NoCaptures` is a **null object**—it satisfies the trait contract but provides no functionality. This allows matchers without capture support to still work with APIs that expect the `Captures` associated type.

---

## Section 6: The NoError Type - When Failure Is Impossible

```rust
/// NoError provides an error type for matchers that never produce errors.
///
/// The Display impl panics—if you see this error, you have a bug.
#[derive(Debug, Eq, PartialEq)]
pub struct NoError(());  // Empty struct, cannot be constructed externally

impl std::error::Error for NoError {
    fn description(&self) -> &str {
        "no error"
    }
}

impl std::fmt::Display for NoError {
    fn fmt(&self, _: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        panic!("BUG for NoError: an impossible error occurred")
    }
}

impl From<NoError> for std::io::Error {
    fn from(_: NoError) -> std::io::Error {
        panic!("BUG for NoError: an impossible error occurred")
    }
}
```

The panicking implementations are intentional. `NoError` exists only to satisfy type system requirements—any actual attempt to use an error value indicates a programming mistake that should fail loudly.

---

## Section 7: The Matcher Trait - Core Interface

```rust
/// A matcher defines an interface for regular expression implementations.
pub trait Matcher {
    /// The concrete type of capturing groups.
    /// Set to `NoCaptures` if not supported.
    type Captures: Captures;

    /// The error type. Use `NoError` for infallible matchers.
    type Error: std::fmt::Display;

    /// Returns the first match in `haystack` after position `at`.
    /// This is the primary required method.
    fn find_at(
        &self,
        haystack: &[u8],
        at: usize,
    ) -> Result<Option<Match>, Self::Error>;

    /// Creates an empty captures container.
    /// The other required method.
    fn new_captures(&self) -> Result<Self::Captures, Self::Error>;

    /// Number of capture groups (0 if unsupported).
    #[inline]
    fn capture_count(&self) -> usize {
        0  // Default: no capture support
    }

    /// Map capture group name to index.
    #[inline]
    fn capture_index(&self, _name: &str) -> Option<usize> {
        None  // Default: no named captures
    }

    /// Find first match starting from beginning.
    #[inline]
    fn find(&self, haystack: &[u8]) -> Result<Option<Match>, Self::Error> {
        self.find_at(haystack, 0)  // Delegates to required method
    }

    /// Check if any match exists (can be faster than find).
    #[inline]
    fn is_match(&self, haystack: &[u8]) -> Result<bool, Self::Error> {
        self.is_match_at(haystack, 0)
    }

    #[inline]
    fn is_match_at(
        &self,
        haystack: &[u8],
        at: usize,
    ) -> Result<bool, Self::Error> {
        Ok(self.shortest_match_at(haystack, at)?.is_some())
    }
}
```

Only `find_at` and `new_captures` are required. All other methods have default implementations that build on these two primitives.

---

## Section 8: Internal Iteration Methods

```rust
impl Matcher {
    /// Iterate over all non-overlapping matches using internal iteration.
    /// The callback returns false to stop iteration.
    #[inline]
    fn try_find_iter_at<F, E>(
        &self,
        haystack: &[u8],
        at: usize,
        mut matched: F,
    ) -> Result<Result<(), E>, Self::Error>
    where
        F: FnMut(Match) -> Result<bool, E>,
    {
        let mut last_end = at;
        let mut last_match = None;

        loop {
            if last_end > haystack.len() {
                return Ok(Ok(()));
            }
            let m = match self.find_at(haystack, last_end)? {
                None => return Ok(Ok(())),
                Some(m) => m,
            };
            // Handle empty matches specially to ensure progress
            if m.start == m.end {
                last_end = m.end + 1;
                // Skip empty matches immediately following another match
                if Some(m.end) == last_match {
                    continue;
                }
            } else {
                last_end = m.end;
            }
            last_match = Some(m.end);
            // Callback controls iteration: Ok(true) continues, Ok(false) stops
            match matched(m) {
                Ok(true) => continue,
                Ok(false) => return Ok(Ok(())),
                Err(err) => return Ok(Err(err)),
            }
        }
    }
}
```

The nested `Result<Result<(), E>, Self::Error>` return type distinguishes between matcher errors (outer) and callback errors (inner). The empty match handling prevents infinite loops on patterns like `a*`.

## Section 9: Blanket Implementation for References

```rust
/// Blanket implementation allows &M to be used wherever M: Matcher is expected.
impl<'a, M: Matcher> Matcher for &'a M {
    type Captures = M::Captures;
    type Error = M::Error;

    #[inline]
    fn find_at(
        &self,
        haystack: &[u8],
        at: usize,
    ) -> Result<Option<Match>, Self::Error> {
        (*self).find_at(haystack, at)  // Delegate through the reference
    }

    #[inline]
    fn new_captures(&self) -> Result<Self::Captures, Self::Error> {
        (*self).new_captures()
    }

    // ... all other methods delegate similarly ...

    #[inline]
    fn non_matching_bytes(&self) -> Option<&ByteSet> {
        (*self).non_matching_bytes()
    }

    #[inline]
    fn line_terminator(&self) -> Option<LineTerminator> {
        (*self).line_terminator()
    }
}
```

This blanket impl means functions accepting `impl Matcher` or `M: Matcher` automatically work with borrowed matchers. The `#[inline]` hints help eliminate the indirection cost.

## Quick Reference

### Key Types

| Type | Purpose |
| --- | --- |

| | |
|---|---|
| `Match` | Byte offset range with `start <= end` invariant |
| `LineTerminator` | Single byte or CRLF line ending |
| `ByteSet` | 256-bit set of bytes (for optimization hints) |
| `NoCaptures` | Null implementation of `Captures` trait |
| `NoError` | Placeholder error type for infallible matchers |

## Matcher Trait Required Methods

```
fn find_at(&self, haystack: &[u8], at: usize) -> Result<Option<Match>,
Self::Error>;
fn new_captures(&self) -> Result<Self::Captures, Self::Error>;
```

## Matcher Associated Types

```
type Captures: Captures;  // Use NoCaptures if unsupported
type Error: std::fmt::Display;  // Use NoError if infallible
```

## Optimization Hints

```
fn non_matching_bytes(&self) -> Option<&ByteSet>;  // Bytes never in a match
fn line_terminator(&self) -> Option<LineTerminator>;  // Line-oriented
optimization
fn find_candidate_line(&self, haystack: &[u8]) -> Result<Option<LineMatchKind>,
Self::Error>;
```

## LineMatchKind Variants

```
LineMatchKind::Confirmed(usize)  // Definite match at position
LineMatchKind::Candidate(usize)  // Possible match, needs verification
```

# Capture Group Interpolation: Code Companion

Reference code for the Capture Group Interpolation lecture. Sections correspond to the lecture document.

---

## Section 1: The Interpolation Function Signature

```
/// Interpolate capture references in `replacement` and write the interpolation
/// result to `dst`. References in `replacement` take the form of $N or $name,
/// where `N` is a capture group index and `name` is a capture group name.
#[inline]
pub fn interpolate<A, N>(
    mut replacement: &[u8],       // Mutable slice - will be narrowed as we parse
    mut append: A,                // Closure: writes captured text given an index
    mut name_to_index: N,         // Closure: resolves named groups to indices
    dst: &mut Vec<u8>,            // Output buffer, caller-controlled
) where
    A: FnMut(usize, &mut Vec<u8>),        // May mutate state between calls
    N: FnMut(&str) -> Option<usize>,      // Returns None for unknown names
{
    // ... implementation
}
```

The `FnMut` bounds allow closures that capture and modify external state. The `Option<usize>` return type for `name_to_index` handles the case where a named group doesn't exist—the caller decides whether that's an error or should be silently ignored.

## Section 2: Efficient Scanning with memchr

```
while !replacement.is_empty() {
    match memchr(b'$', replacement) {
        None => break,  // No more $ characters, exit loop
        Some(i) => {
            // Copy literal text before the $ to output
            dst.extend(&replacement[..i]);
            // Narrow slice to start at the $
            replacement = &replacement[i..];
        }
    }

    // ... handle the $ reference ...
}
// After loop: append any remaining literal text
dst.extend(replacement);
```

The `memchr` crate provides SIMD-accelerated byte searching on supported platforms. By narrowing `replacement` rather than tracking an index, each iteration always starts at a `$` character, simplifying the parsing logic.

---

## Section 3: Escape Sequence Handling

```
// Check if this is an escaped dollar sign ($$)
if replacement.get(1).map_or(false, |&b| b == b'$') {
    dst.push(b'$');           // Output single literal $
    replacement = &replacement[2..];  // Skip both $ characters
    continue;                 // Restart loop to find next $
}
```

The `get(1)` returns `Option<&u8>`, safely handling the case where `$` is at the end of the string. The `map_or(false, ...)` pattern provides a default when the index is out of bounds, avoiding a separate length check.

---

## Section 4: The CaptureRef and Ref Types

```rust
/// Represents a parsed capture reference with its position.
/// The lifetime 'a ties this to the input slice for zero-copy parsing.
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
struct CaptureRef<'a> {
    cap: Ref<'a>,     // The actual reference (number or name)
    end: usize,       // Byte position immediately after the reference
}

/// A reference to a capture group - either by index or by name.
/// e.g., `$2`, `$foo`, `${foo}`.
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
enum Ref<'a> {
    Named(&'a str),   // Borrows from input - no allocation
    Number(usize),    // Self-contained, no borrowing needed
}
```

The `end` field enables the caller to advance past the reference correctly—`${foo}` consumes 6 bytes while `$foo` consumes 4. The lifetime on `Ref::Named` ensures the borrowed string slice remains valid.

---

## Section 5: The From Trait Implementations

```rust
impl<'a> From<&'a str> for Ref<'a> {
    #[inline]
    fn from(x: &'a str) -> Ref<'a> {
        Ref::Named(x)  // Lifetime flows from input to output
    }
}

impl From<usize> for Ref<'static> {
    #[inline]
    fn from(x: usize) -> Ref<'static> {
        Ref::Number(x)  // 'static because numbers don't borrow
    }
}

// Test macro exploiting the From implementations
macro_rules! c {
    ($name_or_number:expr, $pos:expr) => {
        CaptureRef { cap: $name_or_number.into(), end: $pos }
    };
}

// Usage in tests:
// c!("foo", 4)  -> CaptureRef with Named("foo")
// c!(5, 2)      -> CaptureRef with Number(5)
```

The difference in lifetimes (`'a` vs `'static`) reflects a real semantic distinction: named references borrow from their source, while numeric references are independent values.

---

## Section 6: Parsing Capture References

```
#[inline]
fn find_cap_ref(replacement: &[u8]) -> Option<CaptureRef<'_>> {
    let mut i = 0;
    // Must have at least $ and one more character
    if replacement.len() <= 1 || replacement[0] != b'$' {
        return None;
    }

    let mut brace = false;
    i += 1;  // Skip past $

    // Check for braced form: ${...}
    if replacement[i] == b'{' {
        brace = true;
        i += 1;
    }

    // Scan valid identifier characters
    let mut cap_end = i;
    while replacement.get(cap_end).map_or(false, is_valid_cap_letter) {
        cap_end += 1;
    }

    // Must have at least one valid character
    if cap_end == i {
        return None;
    }

    // Safe because is_valid_cap_letter only accepts ASCII
    let cap = std::str::from_utf8(&replacement[i..cap_end])
        .expect("valid UTF-8 capture name");

    // Braced form requires closing brace
    if brace {
        if !replacement.get(cap_end).map_or(false, |&b| b == b'}') {
            return None;
        }
        cap_end += 1;
    }

    // Determine if it's a number or a name
    Some(CaptureRef {
        cap: match cap.parse::<u32>() {
            Ok(i) => Ref::Number(i as usize),
            Err(_) => Ref::Named(cap),
        },
        end: cap_end,
    })
}
```

The function returns `None` for malformed references rather than erroring, allowing the caller to treat unrecognized patterns as literal text.

---

## Section 7: Number vs Name Disambiguation

```
/// Returns true if and only if the given byte is allowed in a capture name.
#[inline]
fn is_valid_cap_letter(b: &u8) -> bool {
    match *b {
        b'0'..=b'9' | b'a'..=b'z' | b'A'..=b'Z' | b'_' => true,
        _ => false,
    }
}

// In find_cap_ref, after extracting the identifier:
Some(CaptureRef {
    cap: match cap.parse::<u32>() {
        Ok(i) => Ref::Number(i as usize),  // "123" -> Number(123)
        Err(_) => Ref::Named(cap),         // "foo" or "42a" -> Named
    },
    end: cap_end,
})
```

The `parse::<u32>()` attempt determines the reference type. Note that `$42a` becomes `Named("42a")` because "42a" fails to parse as a number. Using `u32` before casting to `usize` prevents accepting negative numbers while keeping the index size platform-appropriate.

---

## Quick Reference

### Supported Reference Syntax

| Syntax | Example | Meaning |
|---|---|---|
| `$N` | `$1` | Numeric reference to group N |
| `$name` | `$foo` | Named reference (greedy scan) |
| `${N}` | `${1}` | Braced numeric (for disambiguation) |

| | | |
|---|---|---|
| `${name}` | `${foo}` | Braced named (for disambiguation) |
| `$$` | `$$` | Literal `$` character |

## Key Types

```
struct CaptureRef<'a> { cap: Ref<'a>, end: usize }
enum Ref<'a> { Named(&'a str), Number(usize) }
```

## Function Signatures

```
pub fn interpolate<A, N>(replacement: &[u8], append: A, name_to_index: N, dst:
&mut Vec<u8>)
where
    A: FnMut(usize, &mut Vec<u8>),
    N: FnMut(&str) -> Option<usize>;

fn find_cap_ref(replacement: &[u8]) -> Option<CaptureRef<'_>>;
fn is_valid_cap_letter(b: &u8) -> bool;
```

## Valid Identifier Characters

- Digits: `0-9`
- Lowercase: `a-z`
- Uppercase: `A-Z`
- Underscore: `_`

# The Sink Trait: Code Companion

Reference code for the The Sink Trait lecture. Sections correspond to the lecture document.

---

## Section 1: Push vs Pull Iteration Models

```
/// A trait that defines how results from searchers are handled.
///
/// In this crate, a searcher follows the "push" model. What that means is that
/// the searcher drives execution, and pushes results back to the caller. This
/// is in contrast to a "pull" model where the caller drives execution and
/// takes results as they need them. These are also known as "internal" and
/// "external" iteration strategies, respectively.
///
/// For a variety of reasons, including the complexity of the searcher
/// implementation, this crate chooses the "push" or "internal" model of
/// execution. Thus, in order to act on search results, callers must provide
/// an implementation of this trait to a searcher, and the searcher is then
/// responsible for calling the methods on this trait.
```

The documentation explicitly acknowledges the architectural trade-off. The "push" model keeps complex state management inside the searcher rather than exposing it through iterator suspension points.

---

## Section 2: The SinkError Trait and Error Flexibility

```rust
/// A trait that describes errors that can be reported by searchers and
/// implementations of `Sink`.
pub trait SinkError: Sized {
    /// A constructor for converting any value that satisfies the
    /// `std::fmt::Display` trait into an error.
    fn error_message<T: std::fmt::Display>(message: T) -> Self;

    /// A constructor for converting I/O errors that occur while searching.
    /// By default, this is implemented via the `error_message` constructor.
    fn error_io(err: io::Error) -> Self {
        Self::error_message(err)  // Default: convert to string
    }

    /// A constructor for converting configuration errors.
    /// By default, this is implemented via the `error_message` constructor.
    fn error_config(err: ConfigError) -> Self {
        Self::error_message(err)  // Default: convert to string
    }
}

/// An `std::io::Error` can be used as an error for `Sink` implementations.
impl SinkError for io::Error {
    fn error_message<T: std::fmt::Display>(message: T) -> io::Error {
        io::Error::new(io::ErrorKind::Other, message.to_string())
    }

    fn error_io(err: io::Error) -> io::Error {
        err  // Optimization: no conversion needed for io::Error!
    }
}

/// A `Box<dyn std::error::Error>` can also be used - maximum flexibility.
impl SinkError for Box<dyn std::error::Error> {
    fn error_message<T: std::fmt::Display>(
        message: T,
    ) -> Box<dyn std::error::Error> {
        Box::<dyn std::error::Error>::from(message.to_string())
    }
}
```

The `Sized` bound on `SinkError` is required because the trait returns `Self`. The `io::Error` implementation demonstrates the optimization pattern—`error_io` avoids unnecessary string conversion.

---

## Section 3: The Core Sink Trait

```rust
pub trait Sink {
    /// The type of an error that should be reported by a searcher.
    /// Errors of this type are used both by Sink methods AND by the
    /// searcher implementation itself (e.g., for I/O errors).
    type Error: SinkError;  // Associated type with trait bound

    /// Called whenever a match is found.
    /// Returns: Ok(true) = continue, Ok(false) = stop gracefully, Err = abort
    fn matched(
        &mut self,
        _searcher: &Searcher,       // Access to searcher configuration
        _mat: &SinkMatch<'_>,       // Match details with borrowed data
    ) -> Result<bool, Self::Error>;

    /// Called for context lines. Default: ignore and continue.
    #[inline]
    fn context(
        &mut self,
        _searcher: &Searcher,
        _context: &SinkContext<'_>,
    ) -> Result<bool, Self::Error> {
        Ok(true)  // Default implementation
    }

    // ... additional methods with similar patterns
}
```

The tri-state return type `Result<bool, Self::Error>` appears throughout the trait, providing fine-grained control over search flow. The `&Searcher` parameter lets sinks query configuration without storing it themselves.

---

## Section 4: Lifecycle and Event Methods

```rust
/// Called when a search has begun, before any search is executed.
#[inline]
fn begin(&mut self, _searcher: &Searcher) -> Result<bool, Self::Error> {
    Ok(true)  // Default: do nothing, continue
}

/// Called when a search has completed successfully.
/// Note: returns Result<(), ...> - no bool needed, search is done
#[inline]
fn finish(
    &mut self,
    _searcher: &Searcher,
    _: &SinkFinish,  // Summary statistics
) -> Result<(), Self::Error> {
    Ok(())
}

/// Called when a break in contextual lines is found (the "--" separator).
#[inline]
fn context_break(
    &mut self,
    _searcher: &Searcher,
) -> Result<bool, Self::Error> {
    Ok(true)
}

/// Called when binary data is found during search.
#[inline]
fn binary_data(
    &mut self,
    _searcher: &Searcher,
    _binary_byte_offset: u64,  // Absolute position of binary data
) -> Result<bool, Self::Error> {
    Ok(true)
}
```

All lifecycle methods except `finish` return `Result<bool, ...>` for consistent early-termination control. The `#[inline]` hints help the compiler optimize away empty default implementations.

## Section 5: Blanket Implementations for Indirection

```rust
/// Allows passing `&mut sink` instead of moving ownership
impl<'a, S: Sink> Sink for &'a mut S {
    type Error = S::Error;

    #[inline]
    fn matched(
        &mut self,
        searcher: &Searcher,
        mat: &SinkMatch<'_>,
    ) -> Result<bool, S::Error> {
        (**self).matched(searcher, mat)  // Deref twice: &mut &mut S -> S
    }

    // ... other methods follow same pattern
}

/// Enables dynamic dispatch with trait objects
impl<S: Sink + ?Sized> Sink for Box<S> {
    //         ^^^^^^^ Allows Box<dyn Sink<Error = E>>
    type Error = S::Error;

    #[inline]
    fn matched(
        &mut self,
        searcher: &Searcher,
        mat: &SinkMatch<'_>,
    ) -> Result<bool, S::Error> {
        (**self).matched(searcher, mat)  // Deref Box to inner type
    }

    // ... other methods follow same pattern
}
```

The `?Sized` bound is crucial—without it, `Box<dyn Sink>` wouldn't work because trait objects are unsized. These implementations let you choose between borrowed access and dynamic dispatch.

---

## Section 6: Data Types - SinkFinish and SinkMatch

```rust
/// Summary data reported at the end of a search.
#[derive(Clone, Debug)]
pub struct SinkFinish {
    pub(crate) byte_count: u64,              // Total bytes searched
    pub(crate) binary_byte_offset: Option<u64>, // First binary data location
}

impl SinkFinish {
    #[inline]
    pub fn byte_count(&self) -> u64 {
        self.byte_count
    }

    #[inline]
    pub fn binary_byte_offset(&self) -> Option<u64> {
        self.binary_byte_offset
    }
}

/// A type that describes a match reported by a searcher.
#[derive(Clone, Debug)]
pub struct SinkMatch<'b> {
    pub(crate) line_term: LineTerminator,
    pub(crate) bytes: &'b [u8],           // The matched bytes
    pub(crate) absolute_byte_offset: u64,  // Position in entire input
    pub(crate) line_number: Option<u64>,   // Only if line counting enabled
    pub(crate) buffer: &'b [u8],           // Surrounding buffer context
    pub(crate) bytes_range_in_buffer: std::ops::Range<usize>,
}

impl<'b> SinkMatch<'b> {
    /// Return an iterator over lines in this match (may be multiple in multi-
line mode)
    #[inline]
    pub fn lines(&self) -> LineIter<'b> {
        LineIter::new(self.line_term.as_byte(), self.bytes)
    }
}
```

Fields are `pub(crate)` with public accessor methods—the struct can only be constructed by the searcher, but users can read all fields. The `'b` lifetime ties match data to the underlying buffer.

## Section 7: Convenience Sinks Module

```
pub mod sinks {
    /// A sink that provides line numbers and matches as strings.
    /// Returns error on invalid UTF-8 or if line numbers not enabled.
    #[derive(Clone, Debug)]
    pub struct UTF8<F>(pub F)  // Tuple struct wrapping a closure
    where
        F: FnMut(u64, &str) -> Result<bool, io::Error>;

    impl<F> Sink for UTF8<F>
    where
        F: FnMut(u64, &str) -> Result<bool, io::Error>,
    {
        type Error = io::Error;

        fn matched(
            &mut self,
            _searcher: &Searcher,
            mat: &SinkMatch<'_>,
        ) -> Result<bool, io::Error> {
            // Validate UTF-8
            let matched = match std::str::from_utf8(mat.bytes()) {
                Ok(matched) => matched,
                Err(err) => return Err(io::Error::error_message(err)),
            };
            // Require line numbers
            let line_number = match mat.line_number() {
                Some(line_number) => line_number,
                None => {
                    let msg = "line numbers not enabled";
                    return Err(io::Error::error_message(msg));
                }
            };
            (self.0)(line_number, &matched)  // Call the wrapped closure
        }
    }

    /// Like UTF8, but replaces invalid UTF-8 with replacement characters.
    pub struct Lossy<F>(pub F)
    where
        F: FnMut(u64, &str) -> Result<bool, io::Error>;
}
```

The `UTF8` and `Lossy` sinks demonstrate the trade-off between strictness and convenience. These are tuple structs wrapping closures, providing a functional style for simple use cases.

## Quick Reference

## Sink Method Return Semantics

| Return Value | Meaning |
|---|---|
| `Ok(true)` | Continue searching |
| `Ok(false)` | Stop gracefully, still call `finish` |
| `Err(e)` | Abort immediately, do NOT call `finish` |

## Sink Lifecycle

```
begin() → [matched() | context() | context_break() | binary_data()]* → finish()
```

## Key Types

```rust
trait SinkError: Sized {
    fn error_message<T: Display>(message: T) -> Self;
    fn error_io(err: io::Error) -> Self { ... }
    fn error_config(err: ConfigError) -> Self { ... }
}

trait Sink {
    type Error: SinkError;
    fn matched(&mut self, &Searcher, &SinkMatch) -> Result<bool, Self::Error>;
    // Optional: context, context_break, binary_data, begin, finish
}
```

## Convenience Sinks

| Type | Closure Signature | Behavior |
|---|---|---|
| `UTF8<F>` | `FnMut(u64, &str) -> Result<bool, io::Error>` | Error on invalid UTF-8 |
| `Lossy<F>` | `FnMut(u64, &str) -> Result<bool, io::Error>` | Replace invalid UTF-8 |

| | | |
|---|---|---|
| `Bytes<F>` | `FnMut(u64, &[u8]) -> Result<bool, io::Error>` | Raw bytes, no conversion |

# ripgrep crates/regex/src/lib.rs: Code Companion

Reference code for the Regex Crate Entry lecture. Sections correspond to the lecture document.

---

## Section 1: The Documentation Attribute

```
/*!
An implementation of `grep-matcher`'s `Matcher` trait for Rust's regex engine.
*/
```

The `/*!` syntax creates an inner documentation comment that attaches to the enclosing item—in this case, the crate itself. This becomes the crate-level documentation visible in `cargo doc` output.

---

## Section 2: The Missing Docs Lint

```
#![deny(missing_docs)]
```

The `#!` prefix makes this an inner attribute applying to the entire crate. Any public item without documentation will cause a compilation error, not just a warning.

## Section 3: The Public API Through Re-exports

```
pub use crate::{
    // Error handling types from the error module
    error::{Error, ErrorKind},
    // Core matching types from the matcher module
    matcher::{RegexCaptures, RegexMatcher, RegexMatcherBuilder},
};
```

The `pub use` statement selectively re-exports only five types from internal modules, defining the complete public API. The `crate::` prefix explicitly anchors paths to the crate root.

## Section 4: The Builder Pattern in Context

```
pub use crate::{
    // ...
    matcher::{
        RegexCaptures,         // Holds captured groups from a match
        RegexMatcher,          // The matcher implementing grep-matcher's trait
        RegexMatcherBuilder,   // Builder for configuring RegexMatcher
    },
};
```

Exporting `RegexMatcherBuilder` alongside `RegexMatcher` indicates users construct matchers through the builder pattern rather than direct instantiation. This mirrors the builder ecosystem found throughout ripgrep.

## Section 5: The Internal Module Structure

```
// Private modules - no `pub` keyword means internal to this crate
mod ast;          // Abstract syntax tree manipulation
mod ban;          // Pattern/feature restrictions
mod config;       // Configuration handling
mod error;        // Error types (partially re-exported)
mod literal;      // Literal extraction for optimization
mod matcher;      // Matcher trait implementation (partially re-exported)
mod non_matching;// Handling of non-matching patterns
mod strip;        // Pattern transformation/stripping
```

Eight private modules contain the implementation complexity. Only selected types from `error` and `matcher` are exposed publicly through the `pub use` declaration above.

## Section 6: Why These Specific Modules?

```
// From the re-exports, we can infer the public-facing module purposes:
error::{Error, ErrorKind},  // Structured error handling for regex compilation
failures
matcher::{
    RegexCaptures,          // Capture group storage for the Matcher trait
    RegexMatcher,           // The Matcher trait implementation
    RegexMatcherBuilder,    // Configuration and construction
},
```

The `error` module provides structured diagnostics for compilation failures. The `matcher` module bridges `grep-matcher`'s abstract `Matcher` trait with the concrete `regex` crate. The `literal` module (internal) enables fast-path searching by extracting fixed strings from patterns.

---

## Section 7: The Supporting Modules

```
// Internal modules handling specialized concerns:
mod ast;          // Parse and analyze regex pattern structure
mod ban;          // Restrict problematic regex features
mod config;       // Centralized configuration management
mod non_matching; // Handle patterns guaranteed not to match
mod strip;        // Transform patterns (e.g., strip comments)
```

These modules handle regex analysis and transformation at the AST level, pattern restrictions for grep-style searching, and configuration management—all hidden from crate users behind the minimal five-type public API.

---

## Quick Reference

### Public API Summary

| Type | Module | Purpose |
|---|---|---|
| `Error` | error | Error type for regex operations |
| `ErrorKind` | error | Enum categorizing error causes |
| `RegexMatcher` | matcher | Implements `grep-matcher::Matcher` |

| | | |
|---|---|---|
| `RegexMatcherBuilder` | matcher | Configures and builds `RegexMatcher` |
| `RegexCaptures` | matcher | Stores captured groups from matches |

## Attribute Syntax Reference

| Syntax | Name | Scope |
|---|---|---|
| `#[...]` | Outer attribute | Applies to the following item |
| `#![...]` | Inner attribute | Applies to the enclosing item |
| `/*! ... */` | Inner doc comment | Documents the enclosing item |
| `/// ...` | Outer doc comment | Documents the following item |

## Module Visibility

```
mod private_mod;      // Private to this crate
pub mod public_mod;   // Visible outside this crate
pub use mod::Type;    // Re-export specific items as public
```

# ripgrep crates/regex/src/config.rs: Code Companion

Reference code for the Regex Configuration lecture. Sections correspond to the lecture document.

---

## Section 1: The Configuration Structure

```rust
/// Config represents the configuration of a regex matcher in this crate.
/// The configuration is itself a rough combination of the knobs found in
/// the `regex` crate itself, along with additional `grep-matcher` specific
/// options.
#[derive(Clone, Debug)]
pub(crate) struct Config {
    // Standard regex options
    pub(crate) case_insensitive: bool,
    pub(crate) case_smart: bool,           // ripgrep's intelligent case
detection
    pub(crate) multi_line: bool,
    pub(crate) dot_matches_new_line: bool,
    pub(crate) swap_greed: bool,
    pub(crate) ignore_whitespace: bool,
    pub(crate) unicode: bool,
    pub(crate) octal: bool,

    // Resource limits
    pub(crate) size_limit: usize,
    pub(crate) dfa_size_limit: usize,
    pub(crate) nest_limit: u32,

    // ripgrep-specific options
    pub(crate) line_terminator: Option<LineTerminator>,
    pub(crate) ban: Option<u8>,            // byte to ban from matches
    pub(crate) crlf: bool,                 // Windows line ending support
    pub(crate) word: bool,                 // -w flag: word boundary matching
    pub(crate) fixed_strings: bool,        // -F flag: literal matching
    pub(crate) whole_line: bool,           // -x flag: full line matching
}
```

All fields use `pub(crate)` visibility—configuration is set through the builder API, not by direct manipulation. The `Clone` and `Debug` derives are essential for configuration types that need to be duplicated and debugged.

## Section 2: Default Values and Their Rationale

```
impl Default for Config {
    fn default() -> Config {
        Config {
            case_insensitive: false,
            case_smart: false,
            multi_line: false,
            dot_matches_new_line: false,
            swap_greed: false,
            ignore_whitespace: false,
            unicode: true,              // User-friendly default: Unicode support
on
            octal: false,
            // These size limits are much bigger than what's in the regex
            // crate by default.
            size_limit: 100 * (1 << 20),        // 100 MB for NFA
            dfa_size_limit: 1000 * (1 << 20),   // 1 GB for lazy DFA cache
            nest_limit: 250,                     // Max parenthesis nesting depth
            line_terminator: None,
            ban: None,
            crlf: false,
            word: false,
            fixed_strings: false,
            whole_line: false,
        }
    }
}
```

The generous size limits (100MB NFA, 1GB DFA cache) reflect that ripgrep is an end-user tool where users expect to search complex patterns. The `unicode: true` default prioritizes correctness over performance.

## Section 3: Smart Case Detection

```
impl Config {
    /// Accounting for the `smart_case` config knob, return true if and only if
    /// this pattern should be matched case insensitively.
    fn is_case_insensitive(&self, analysis: &AstAnalysis) -> bool {
        // Explicit case-insensitive always wins
        if self.case_insensitive {
            return true;
        }
        // If smart case is off, don't do anything clever
        if !self.case_smart {
            return false;
        }
        // Smart case logic: case-insensitive if pattern has literals
        // but no uppercase characters
        // "foo" -> case insensitive (matches Foo, FOO)
        // "Foo" -> case sensitive (user explicitly used uppercase)
        analysis.any_literal() && !analysis.any_uppercase()
    }
}
```

The `AstAnalysis` parameter contains pre-computed information about the pattern's AST, avoiding repeated traversals. This method is called during HIR translation to determine the final case sensitivity setting.

---

## Section 4: Fixed String Detection

```
/// Returns whether the given patterns should be treated as "fixed strings"
/// literals.
fn is_fixed_strings<P: AsRef<str>>(&self, patterns: &[P]) -> bool {
    // Case transforms require full parsing pipeline
    if self.case_insensitive || self.case_smart {
        return false;
    }

    // User explicitly requested fixed strings (-F flag)
    if self.fixed_strings {
        // But bail if any literal contains a line terminator
        if let Some(lineterm) = self.line_terminator {
            for p in patterns.iter() {
                if has_line_terminator(lineterm, p.as_ref()) {
                    return false;  // Let normal error handling catch this
                }
            }
        }
        return true;
    }

    // Check if patterns are effectively literals (no metacharacters)
    for p in patterns.iter() {
        let p = p.as_ref();
        // Any metacharacter means we need full parsing
        if p.chars().any(regex_syntax::is_meta_character) {
            return false;
        }
        // Same line terminator check as above
        if let Some(lineterm) = self.line_terminator {
            if has_line_terminator(lineterm, p) {
                return false;
            }
        }
    }
    true  // All patterns are plain literals
}
```

The generic `P: AsRef<str>` bound allows this method to accept `&[String]`, `&[&str]`, or any other string-like slice. The method returns `false` conservatively to ensure correctness over optimization.

## Section 5: The ConfiguredHIR Type

```rust
/// A "configured" HIR expression, which is aware of the configuration which
/// produced this HIR.
#[derive(Clone, Debug)]
pub(crate) struct ConfiguredHIR {
    config: Config,
    hir: Hir,
}

impl ConfiguredHIR {
    /// Return a reference to the underlying configuration.
    pub(crate) fn config(&self) -> &Config {
        &self.config
    }

    /// Return a reference to the underlying HIR.
    pub(crate) fn hir(&self) -> &Hir {
        &self.hir
    }

    /// Turns this configured HIR into an equivalent one, but where it must
    /// match at word boundaries.
    pub(crate) fn into_word(self) -> ConfiguredHIR {
        // Consume self, wrap the HIR, return new ConfiguredHIR
        let hir = Hir::concat(vec![
            Hir::look(if self.config.unicode {
                hir::Look::WordStartHalfUnicode  // Unicode word boundary
            } else {
                hir::Look::WordStartHalfAscii    // ASCII word boundary
            }),
            self.hir,                            // Original pattern
            Hir::look(if self.config.unicode {
                hir::Look::WordEndHalfUnicode
            } else {
                hir::Look::WordEndHalfAscii
            }),
        ]);
        ConfiguredHIR { config: self.config, hir }
    }

    /// Turns this configured HIR into an equivalent one, but where it must
    /// match at the start and end of a line.
    pub(crate) fn into_whole_line(self) -> ConfiguredHIR {
        let line_anchor_start = Hir::look(self.line_anchor_start());
        let line_anchor_end = Hir::look(self.line_anchor_end());
        let hir =
            Hir::concat(vec![line_anchor_start, self.hir, line_anchor_end]);
        ConfiguredHIR { config: self.config, hir }
    }
}
```

The `into_` methods take `self` by value (consuming ownership) and return a new `ConfiguredHIR`. This ownership-based transformation pattern is idiomatic Rust—you explicitly move the original to create the modified version.

---

## Section 6: Building HIR from Patterns

```
impl ConfiguredHIR {
    /// Parse the given patterns into a single HIR expression that represents
    /// an alternation of the patterns given.
    fn new<P: AsRef<str>>(
        config: Config,
        patterns: &[P],
    ) -> Result<ConfiguredHIR, Error> {
        let hir = if config.is_fixed_strings(patterns) {
            // Fast path: directly construct HIR from literal bytes
            let mut alts = vec![];
            for p in patterns.iter() {
                alts.push(Hir::literal(p.as_ref().as_bytes()));
            }
            log::debug!(
                "assembling HIR from {} fixed string literals",
                alts.len()
            );
            Hir::alternation(alts)
        } else {
            // Normal path: parse and translate through regex-syntax
            let mut alts = vec![];
            for p in patterns.iter() {
                alts.push(if config.fixed_strings {
                    // Escape metacharacters for -F with case transforms
                    format!("(?:{})", regex_syntax::escape(p.as_ref()))
                } else {
                    format!("(?:{})", p.as_ref())
                });
            }
            let pattern = alts.join("|");

            // Parse to AST
            let ast = ast::parse::ParserBuilder::new()
                .nest_limit(config.nest_limit)
                .octal(config.octal)
                .ignore_whitespace(config.ignore_whitespace)
                .build()
                .parse(&pattern)
                .map_err(Error::generic)?;

            // Analyze AST for smart case detection
            let analysis = AstAnalysis::from_ast(&ast);

            // Translate AST to HIR with configuration
            let mut hir = hir::translate::TranslatorBuilder::new()
                .utf8(false)  // Allow matching invalid UTF-8
                .case_insensitive(config.is_case_insensitive(&analysis))
                .multi_line(config.multi_line)
                .dot_matches_new_line(config.dot_matches_new_line)
                .crlf(config.crlf)
                .swap_greed(config.swap_greed)
                .unicode(config.unicode)
                .build()
                .translate(&pattern, &ast)
```

```
                .map_err(Error::generic)?;            // Check for banned bytes
(e.g., NUL)            if let Some(byte) = config.ban {
ban::check(&hir, byte)?;            }            // Strip line terminators from
match if configured            hir = match config.line_terminator {
None => hir,                Some(line_term) => strip_from_match(hir, line_term)?,
};            hir        };        Ok(ConfiguredHIR { config, hir })    }}
```

The pipeline shows the full journey from patterns to HIR: join patterns with alternation, parse to AST, analyze for smart case, translate with configuration, apply post-processing. The fast path skips everything for pure literals.

---

## Section 7: Regex Compilation

```
impl ConfiguredHIR {
    /// Convert this HIR to a regex that can be used for matching.
    pub(crate) fn to_regex(&self) -> Result<Regex, Error> {
        let meta = Regex::config()
            .utf8_empty(false)
            .nfa_size_limit(Some(self.config.size_limit))
            // One-pass DFA gets extra room (10 MB)
            .onepass_size_limit(Some(10 * (1 << 20)))
            // Full DFA: small but larger than regex crate default
            .dfa_size_limit(Some(1 * (1 << 20)))
            .dfa_state_limit(Some(1_000))
            // Lazy DFA gets the big cache
            .hybrid_cache_capacity(self.config.dfa_size_limit);

        Regex::builder()
            .configure(meta)
            .build_from_hir(&self.hir)
            .map_err(Error::regex)
    }
}
```

The configuration sets different limits for different regex engine strategies. The lazy DFA (hybrid) gets the largest allocation because it's ripgrep's primary matching strategy for most patterns.

## Section 8: Line Terminator Handling

```
impl ConfiguredHIR {
    /// Returns the line terminator configured on this expression.
    pub(crate) fn line_terminator(&self) -> Option<LineTerminator> {
        // Text anchors (\A, \z) disable fast line searching
        if self.hir.properties().look_set().contains_anchor_haystack() {
            None
        } else {
            self.config.line_terminator
        }
    }

    /// Returns the "start line" anchor for this configuration.
    fn line_anchor_start(&self) -> hir::Look {
        if self.config.crlf {
            hir::Look::StartCRLF   // Windows: handle \r\n
        } else {
            hir::Look::StartLF    // Unix: just \n
        }
    }

    /// Returns the "end line" anchor for this configuration.
    fn line_anchor_end(&self) -> hir::Look {
        if self.config.crlf {
            hir::Look::EndCRLF
        } else {
            hir::Look::EndLF
        }
    }
}

/// Returns true if the given literal string contains any byte from the line
/// terminator given.
fn has_line_terminator(lineterm: LineTerminator, literal: &str) -> bool {
    if lineterm.is_crlf() {
        // CRLF mode: check for both \r and \n
        literal.as_bytes().iter().copied().any(|b| b == b'\r' || b == b'\n')
    } else {
        // Single byte terminator
        literal.as_bytes().iter().copied().any(|b| b == lineterm.as_byte())
    }
}
```

The `line_terminator()` method demonstrates a subtle correctness concern: text anchors (`^`/`$` without multi-line mode) interact poorly with line-by-line searching, so the method returns `None` to disable the optimization in those cases.

## Quick Reference

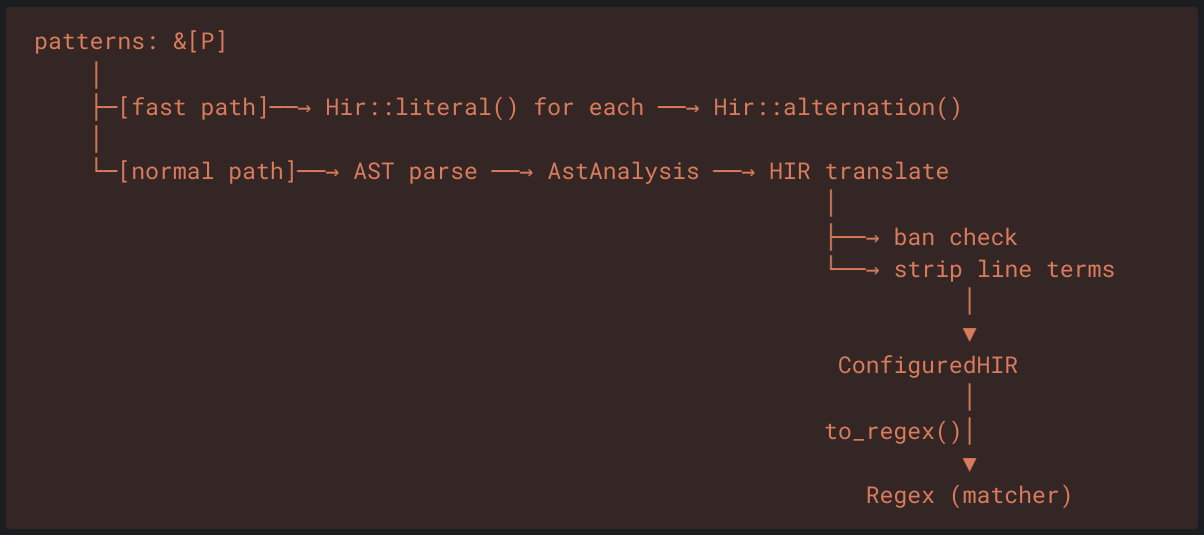
## Config Fields by Category

| Category | Fields |
|---|---|
| Case handling | `case_insensitive`, `case_smart` |
| Line mode | `multi_line`, `dot_matches_new_line`, `crlf` |
| Pattern syntax | `ignore_whitespace`, `unicode`, `octal`, `swap_greed` |
| Resource limits | `size_limit`, `dfa_size_limit`, `nest_limit` |
| ripgrep features | `line_terminator`, `ban`, `word`, `fixed_strings`, `whole_line` |

## Key Methods

| Method | Purpose |
|---|---|
| `Config::build_many()` | Entry point: patterns → ConfiguredHIR |
| `Config::is_case_insensitive()` | Resolve smart case logic |
| `Config::is_fixed_strings()` | Detect fast path eligibility |
| `ConfiguredHIR::new()` | Full compilation pipeline |
| `ConfiguredHIR::to_regex()` | HIR → executable Regex |
| `ConfiguredHIR::into_word()` | Add word boundary assertions |
| `ConfiguredHIR::into_whole_line()` | Add line anchor assertions |

## Compilation Pipeline


patterns: &[P]
[fast path]──→ Hir::literal() for each ──→ Hir::alternation()
[normal path]──→ AST parse ──→ AstAnalysis ──→ HIR translate
ban check
strip line terms
ConfiguredHIR
to_regex()
Regex (matcher)

# ripgrep crates/regex/src/matcher.rs: Code Companion

Reference code for the Implementing Matcher lecture. Sections correspond to the lecture document.

## Section 1: The Builder Pattern Completed

```rust
/// A builder for constructing a `Matcher` using regular expressions.
///
/// This builder re-exports many of the same options found on the regex crate's
/// builder, in addition to a few other options such as smart case, word
/// matching and the ability to set a line terminator which may enable certain
/// types of optimizations.
#[derive(Clone, Debug)]  // Clone enables reuse, Debug enables logging
pub struct RegexMatcherBuilder {
    config: Config,  // Delegates storage to Config struct
}

impl Default for RegexMatcherBuilder {
    fn default() -> RegexMatcherBuilder {
        RegexMatcherBuilder::new()
    }
}

impl RegexMatcherBuilder {
    /// Create a new builder for configuring a regex matcher.
    pub fn new() -> RegexMatcherBuilder {
        RegexMatcherBuilder { config: Config::default() }
    }
}
```

The builder holds a single `Config` field, creating a clean separation between the fluent API (builder methods) and the actual configuration storage. The `Default` trait delegates to `new()`, ensuring consistent initialization.

## Section 2: From Pattern to Matcher

```rust
/// Build a new matcher using the current configuration for the provided
/// pattern.
pub fn build(&self, pattern: &str) -> Result<RegexMatcher, Error> {
    self.build_many(&[pattern])  // Single pattern delegates to multi-pattern
}

/// Build a new matcher using the current configuration for the provided
/// patterns. The resulting matcher behaves as if all of the patterns
/// given are joined together into a single alternation.
pub fn build_many<P: AsRef<str>>(
    &self,
    patterns: &[P],
) -> Result<RegexMatcher, Error> {
    // Step 1: Build configured HIR from patterns
    let mut chir = self.config.build_many(patterns)?;

    // Step 2: Apply whole-line or word transformations
    // 'whole_line' is a strict subset of 'word', so when it is enabled,
    // we don't need to both with any specific to word matching.
    if chir.config().whole_line {
        chir = chir.into_whole_line();
    } else if chir.config().word {
        chir = chir.into_word();
    }

    // Step 3: Compile HIR to actual regex automaton
    let regex = chir.to_regex()?;
    log::trace!("final regex: {:?}", chir.hir().to_string());

    // ... optimization setup continues in next section
}
```

The generic bound `P: AsRef<str>` allows passing `&[&str]`, `&[String]`, or any slice of string-like types. The `into_whole_line()` and `into_word()` methods consume and transform the HIR.

## Section 3: The Fast-Line Optimization

```
pub fn build_many<P: AsRef<str>>(
    &self,
    patterns: &[P],
) -> Result<RegexMatcher, Error> {
    // ... pattern compilation from previous section ...

    let non_matching_bytes = chir.non_matching_bytes();

    // If we can pick out some literals from the regex, then we might be
    // able to build a faster regex that quickly identifies candidate
    // matching lines. For example, for a regex like `\w+foo\w+`, we can
    // look for `foo`, and when a match is found, look for the line
    // containing `foo` and then run the original regex on only that line.
    let fast_line_regex = InnerLiterals::new(&chir, &regex).one_regex()?;

    // We override the line terminator in case the configured HIR doesn't
    // support it.
    let mut config = self.config.clone();
    config.line_terminator = chir.line_terminator();

    // Construct the final matcher with all four components
    Ok(RegexMatcher {
        config,              // Preserved for line_terminator queries
        regex,               // The main matching engine
        fast_line_regex,     // Optional optimization (may be None)
        non_matching_bytes   // Bytes that can never appear in matches
    })
}
```

The `Option<Regex>` type for `fast_line_regex` makes the optimization's optionality explicit—callers must handle both cases.

## Section 4: Builder Methods as Documentation

```
/// Whether to enable "smart case" or not.
///
/// When smart case is enabled, the builder will automatically enable
/// case insensitive matching based on how the pattern is written. Namely,
/// case insensitive mode is enabled when both of the following things
/// are true:
///
/// 1. The pattern contains at least one literal character.
/// 2. Of the literals in the pattern, none of them are considered to be
///    uppercase according to Unicode.
pub fn case_smart(&mut self, yes: bool) -> &mut RegexMatcherBuilder {
    self.config.case_smart = yes;
    self  // Returns &mut Self for method chaining
}

/// Set an ASCII line terminator for the matcher.
///
/// The purpose of setting a line terminator is to enable a certain class
/// of optimizations that can make line oriented searching faster. Namely,
/// when a line terminator is enabled, then the builder will guarantee that
/// the resulting matcher will never be capable of producing a match that
/// contains the line terminator.
///
/// If the aforementioned guarantee cannot be made because of how the pattern
/// was written, then the builder will return an error. For example, the
/// pattern `a\nb` will result in an error since the `\n` cannot be easily
/// removed without changing the fundamental intent of the pattern.
pub fn line_terminator(
    &mut self,
    line_term: Option<u8>,
) -> &mut RegexMatcherBuilder {
    self.config.line_terminator = line_term.map(LineTerminator::byte);
    self
}

/// Require that all matches occur on word boundaries.
///
/// Enabling this option is subtly different than putting `\b` assertions
/// on both sides of your pattern. A `\b` assertion requires that one side
/// match a word character while the other match a non-word character.
/// This option merely requires that one side match a non-word character.
pub fn word(&mut self, yes: bool) -> &mut RegexMatcherBuilder {
    self.config.word = yes;
    self
}
```

Each builder method modifies the config and returns `&mut Self`, enabling fluent chaining like `builder.case_smart(true).word(true).build(pattern)`.

## Section 5: The RegexMatcher Structure

```
/// An implementation of the `Matcher` trait using Rust's standard regex
/// library.
#[derive(Clone, Debug)]
pub struct RegexMatcher {
    /// The configuration specified by the caller.
    config: Config,

    /// The regular expression compiled from the pattern provided by the
    /// caller.
    regex: Regex,  // regex_automata::meta::Regex

    /// A regex that never reports false negatives but may report false
    /// positives that is believed to be capable of being matched more quickly
    /// than `regex`. Typically, this is a single literal or an alternation
    /// of literals.
    fast_line_regex: Option<Regex>,

    /// A set of bytes that will never appear in a match.
    non_matching_bytes: ByteSet,
}

impl RegexMatcher {
    /// Create a new matcher from the given pattern using the default
    /// configuration.
    pub fn new(pattern: &str) -> Result<RegexMatcher, Error> {
        RegexMatcherBuilder::new().build(pattern)
    }

    /// Create a new matcher from the given pattern using the default
    /// configuration, but matches lines terminated by `\n`.
    ///
    /// This is meant to be a convenience constructor for enabling
    /// line-oriented optimizations.
    pub fn new_line_matcher(pattern: &str) -> Result<RegexMatcher, Error> {
        RegexMatcherBuilder::new().line_terminator(Some(b'\n')).build(pattern)
    }
}
```

The four fields each serve distinct purposes: `config` for queries, `regex` for matching, `fast_line_regex` for optimization, and `non_matching_bytes` for skipping.

## Section 6: Implementing the Matcher Trait

```
use grep_matcher::{
    ByteSet, Captures, LineMatchKind, LineTerminator, Match, Matcher, NoError,
};

impl Matcher for RegexMatcher {
    type Captures = RegexCaptures;
    type Error = NoError;  // Regex operations are infallible once compiled

    #[inline]
    fn find_at(
        &self,
        haystack: &[u8],
        at: usize,
    ) -> Result<Option<Match>, NoError> {
        // Create input spanning from 'at' to end of haystack
        let input = Input::new(haystack).span(at..haystack.len());
        // Convert regex_automata::Match to grep_matcher::Match
        Ok(self.regex.find(input).map(|m| Match::new(m.start(), m.end())))
    }

    #[inline]
    fn new_captures(&self) -> Result<RegexCaptures, NoError> {
        Ok(RegexCaptures::new(self.regex.create_captures()))
    }

    #[inline]
    fn capture_count(&self) -> usize {
        self.regex.captures_len()
    }

    #[inline]
    fn capture_index(&self, name: &str) -> Option<usize> {
        // Look up named capture group in pattern 0
        self.regex.group_info().to_index(PatternID::ZERO, name)
    }

    #[inline]
    fn non_matching_bytes(&self) -> Option<&ByteSet> {
        Some(&self.non_matching_bytes)
    }

    #[inline]
    fn line_terminator(&self) -> Option<LineTerminator> {
        self.config.line_terminator
    }
}
```

The associated type `Error = NoError` indicates that regex operations cannot fail at runtime—all validation happens at compile time.

## Section 7: The Fast-Line Search in Action

```
impl Matcher for RegexMatcher {
    // ... other methods ...

    #[inline]
    fn find_candidate_line(
        &self,
        haystack: &[u8],
    ) -> Result<Option<LineMatchKind>, NoError> {
        Ok(match self.fast_line_regex {
            // Fast path: use optimized literal search
            Some(ref regex) => {
                let input = Input::new(haystack);
                regex
                    .search_half(&input)  // Only need match position, not full
match
                    .map(|hm| LineMatchKind::Candidate(hm.offset()))
            }
            // Slow path: run full regex
            None => {
                self.shortest_match(haystack)?.map(LineMatchKind::Confirmed)
            }
        })
    }

    #[inline]
    fn shortest_match_at(
        &self,
        haystack: &[u8],
        at: usize,
    ) -> Result<Option<usize>, NoError> {
        let input = Input::new(haystack).span(at..haystack.len());
        // search_half returns just the end position, not the full match
        Ok(self.regex.search_half(&input).map(|hm| hm.offset()))
    }
}
```

`LineMatchKind::Candidate` indicates a potential match that needs verification; `LineMatchKind::Confirmed` indicates a definite match. The `search_half` method is faster than `find` because it only determines *if* a match exists and where it ends.

## Section 8: Captures Implementation

```rust
use regex_automata::util::captures::Captures as AutomataCaptures;

/// Represents the match offsets of each capturing group in a match.
///
/// The first, or `0`th capture group, always corresponds to the entire match.
/// Note that not all capturing groups are guaranteed to be present in a match.
#[derive(Clone, Debug)]
pub struct RegexCaptures {
    /// Where the captures are stored.
    caps: AutomataCaptures,
}

impl Captures for RegexCaptures {
    #[inline]
    fn len(&self) -> usize {
        self.caps.group_info().all_group_len()
    }

    #[inline]
    fn get(&self, i: usize) -> Option<Match> {
        // Convert from regex_automata's Span to grep_matcher's Match
        self.caps.get_group(i).map(|sp| Match::new(sp.start, sp.end))
    }
}

impl RegexCaptures {
    #[inline]
    pub(crate) fn new(caps: AutomataCaptures) -> RegexCaptures {
        RegexCaptures { caps }
    }

    #[inline]
    pub(crate) fn captures_mut(&mut self) -> &mut AutomataCaptures {
        &mut self.caps
    }
}
```

The `pub(crate)` visibility restricts `new` and `captures_mut` to the crate—external users interact through the `Captures` trait only.

---

## Section 9: Iteration Implementation

```rust
impl Matcher for RegexMatcher {
    #[inline]
    fn try_find_iter<F, E>(
        &self,
        haystack: &[u8],
        mut matched: F,
    ) -> Result<Result<(), E>, NoError>
    where
        F: FnMut(Match) -> Result<bool, E>,
    {
        for m in self.regex.find_iter(haystack) {
            match matched(Match::new(m.start(), m.end())) {
                Ok(true) => continue,      // Keep searching
                Ok(false) => return Ok(Ok(())),  // Caller wants to stop
                Err(err) => return Ok(Err(err)), // Caller encountered error
            }
        }
        Ok(Ok(()))
    }

    #[inline]
    fn captures_at(
        &self,
        haystack: &[u8],
        at: usize,
        caps: &mut RegexCaptures,
    ) -> Result<bool, NoError> {
        let input = Input::new(haystack).span(at..haystack.len());
        let caps = caps.captures_mut();  // Get mutable access to inner storage
        self.regex.search_captures(&input, caps);
        Ok(caps.is_match())
    }
}
```

The `Result<Result<(), E>, NoError>` return type separates regex errors (outer `Result`, always `Ok`) from callback errors (inner `Result`). The callback returns `bool` to indicate whether iteration should continue.

---

## Quick Reference

### RegexMatcher Fields

| Field | Type | Purpose |
|---|---|---|

| | | |
|---|---|---|
| `config` | `Config` | Stores line terminator for queries |
| `regex` | `Regex` | Main matching engine |
| `fast_line_regex` | `Option<Regex>` | Literal-based optimization |
| `non_matching_bytes` | `ByteSet` | Bytes that never match |

## Key Matcher Trait Methods

| Method | Returns | Purpose |
|---|---|---|
| `find_at` | `Option<Match>` | Find first match at offset |
| `shortest_match_at` | `Option<usize>` | Find match end position only |
| `find_candidate_line` | `Option<LineMatchKind>` | Fast pre-filtering |
| `captures_at` | `bool` | Populate capture groups |
| `try_find_iter` | `Result<(), E>` | Iterate with early exit |

## LineMatchKind Variants

```
enum LineMatchKind {
    Candidate(usize),  // Potential match, needs verification
    Confirmed(usize),  // Definite match
}
```

## Builder Method Chaining Pattern

```
RegexMatcherBuilder::new()
    .case_smart(true)
    .word(true)
    .line_terminator(Some(b'\n'))
    .build(pattern)?
```

# ripgrep crates/regex/src/literal.rs: Code Companion

Reference code for the Literal Optimization lecture. Sections correspond to the lecture document.

---

## Section 1: The Inner Literal Strategy

```
/// A type that encapsulates "inner" literal extraction from a regex.
///
/// The main idea underlying the validity of this technique is the fact
/// that ripgrep searches individual lines and not across lines.
#[derive(Clone, Debug)]
pub(crate) struct InnerLiterals {
    seq: Seq,  // The extracted literal sequence
}

impl InnerLiterals {
    pub(crate) fn new(chir: &ConfiguredHIR, re: &Regex) -> InnerLiterals {
        // Inner literal optimization requires line boundaries to work
        if chir.config().line_terminator.is_none() {
            log::trace!(
                "skipping inner literal extraction, \
                 no line terminator is set"
            );
            return InnerLiterals::none();
        }

        // Defer to regex engine's optimizations unless Unicode word
        // boundaries might cause slowdowns
        if re.is_accelerated() {
            if !chir.hir().properties().look_set().contains_word_unicode() {
                log::trace!(
                    "skipping inner literal extraction, \
                     existing regex is believed to already be accelerated",
                );
                return InnerLiterals::none();
            }
        }

        // Pure alternation of literals: regex engine handles this optimally
        if chir.hir().properties().is_alternation_literal() {
            log::trace!(
                "skipping inner literal extraction, \
                 found alternation of literals, deferring to regex engine",
            );
            return InnerLiterals::none();
        }

        let seq = Extractor::new().extract_untagged(chir.hir());
        InnerLiterals { seq }
    }

    /// Returns an infinite set - signals no useful literals found
    pub(crate) fn none() -> InnerLiterals {
        InnerLiterals { seq: Seq::infinite() }
    }
}
```

The `is_accelerated()` check queries whether the regex engine has its own internal optimizations. The `contains_word_unicode()` check identifies patterns with `\b` in Unicode mode, which can cause the regex engine to fall back to slower paths.

---

## Section 2: The Extractor Architecture

```
/// An inner literal extractor with configurable limits.
#[derive(Debug)]
struct Extractor {
    limit_class: usize,       // Max chars to expand from character class
    limit_repeat: usize,      // Max repetition unrolling iterations
    limit_literal_len: usize, // Max length of any single literal
    limit_total: usize,       // Max total literals in final sequence
}

impl Extractor {
    fn new() -> Extractor {
        Extractor {
            limit_class: 10,
            limit_repeat: 10,
            limit_literal_len: 100,
            limit_total: 64,
        }
    }

    /// Main entry point - extracts and post-processes literals
    fn extract_untagged(&self, hir: &Hir) -> Seq {
        let mut seq = self.extract(hir);
        log::trace!("extracted inner literals: {:?}", seq.seq);

        // Optimization pass for prefix-based searching
        seq.seq.optimize_for_prefix_by_preference();
        log::trace!(
            "extracted inner literals after optimization: {:?}",
            seq.seq
        );

        // Heuristic check: are these literals actually useful?
        if !seq.is_good() {
            log::trace!(
                "throwing away inner literals because they might be slow"
            );
            seq.make_infinite();
        }
        seq.seq
    }

    /// Recursive extraction dispatches based on HIR node type
    fn extract(&self, hir: &Hir) -> TSeq {
        use regex_syntax::hir::HirKind::*;

        match *hir.kind() {
            Empty | Look(_) => TSeq::singleton(Literal::exact(vec![])),
            Literal(hir::Literal(ref bytes)) => {
                let mut seq =
                    TSeq::singleton(Literal::exact(bytes.to_vec()));
                self.enforce_literal_len(&mut seq);
                seq
            }
            Class(hir::Class::Unicode(ref cls)) => {
```

```
                self.extract_class_unicode(cls)            }
Class(hir::Class::Bytes(ref cls)) => self.extract_class_bytes(cls),
Repetition(ref rep) => self.extract_repetition(rep),
Capture(hir::Capture { ref sub, .. }) => self.extract(sub),            Concat(ref
hirs) => self.extract_concat(hirs.iter()),            Alternation(ref hirs) =>
self.extract_alternation(hirs.iter()),        }    }}
```

The `TSeq` wrapper ("tagged sequence") tracks whether the sequence represents a prefix position in the regex. This metadata is stripped away in `extract_untagged` before returning the final `Seq`.

## Section 3: Handling Concatenation

```
/// Extract from concatenation via cross product of sub-expressions
fn extract_concat<'a, I: Iterator<Item = &'a Hir>>(&self, it: I) -> TSeq {
    let mut seq = TSeq::singleton(Literal::exact(vec![]));
    let mut prev: Option<TSeq> = None;  // Tracks best candidate so far

    for hir in it {
        // Once all literals are inexact, cross product is a no-op
        if seq.is_inexact() {
            // Empty sequence = impossible match, bail early
            if seq.is_empty() {
                return seq;
            }
            // Found something really good? Stop looking
            if seq.is_really_good() {
                return seq;
            }
            // Save current best, start fresh for next segment
            prev = Some(match prev {
                None => seq,
                Some(prev) => prev.choose(seq),  // Pick better of two
            });
            seq = TSeq::singleton(Literal::exact(vec![]));
            seq.make_not_prefix();  // No longer at pattern start
        }
        // Cross product combines current seq with next sub-expression
        seq = self.cross(seq, self.extract(hir));
    }
    // Return best of saved candidate and final sequence
    if let Some(prev) = prev { prev.choose(seq) } else { seq }
}
```

The `prev` variable implements the "keep the best so far" strategy. When the current sequence becomes inexact (hitting a gap like `[a-z]+`), we save it and start fresh, later choosing whichever is better.

## Section 4: Handling Alternation

```
/// Extract from alternation via union of sub-expressions
fn extract_alternation<'a, I: Iterator<Item = &'a Hir>>(
    &self,
    it: I,
) -> TSeq {
    let mut seq = TSeq::empty();
    for hir in it {
        // Infinite union with anything stays infinite - short circuit
        if !seq.is_finite() {
            break;
        }
        seq = self union(seq, &mut self.extract(hir));
    }
    seq
}

/// Union with limit enforcement and trimming optimization
fn union(&self, mut seq1: TSeq, seq2: &mut TSeq) -> TSeq {
    if seq1.max_union_len(seq2).map_or(false, |len| len > self.limit_total)
    {
        // Try trimming to 4 bytes before giving up
        // 4 bytes chosen for Teddy algorithm compatibility
        seq1.keep_first_bytes(4);
        seq2.keep_first_bytes(4);
        seq1.dedup();
        seq2.dedup();

        // Still over limit? Make infinite
        if seq1
            .max_union_len(seq2)
            .map_or(false, |len| len > self.limit_total)
        {
            seq2.make_infinite();
        }
    }
    seq1.union(seq2);
    assert!(seq1.len().map_or(true, |x| x <= self.limit_total));
    seq1.prefix = seq1.prefix && seq2.prefix;
    seq1
}
```

The magic number 4 for `keep_first_bytes` is an abstraction leak from the Teddy SIMD algorithm, which efficiently searches for literals up to 4 bytes. Trimming to this length preserves usefulness even when the full literals would exceed limits.

# Section 5: Handling Repetition

```
fn extract_repetition(&self, rep: &hir::Repetition) -> TSeq {
    let mut subseq = self.extract(&rep.sub);

    match *rep {
        // Optional: a? means "a or empty"
        hir::Repetition { min: 0, max, greedy, .. } => {
            // max=1 preserves exactness (a? = a|"")
            if max != Some(1) {
                subseq.make_inexact();
            }
            let mut empty = TSeq::singleton(Literal::exact(vec![]));
            // Greedy vs lazy affects union order
            if !greedy {
                std::mem::swap(&mut subseq, &mut empty);
            }
            self.union(subseq, &mut empty)
        }

        // Fixed repetition: a{3} -> "aaa"
        hir::Repetition { min, max: Some(max), .. } if min == max => {
            assert!(min > 0);
            let limit = u32::try_from(self.limit_repeat).unwrap_or(u32::MAX);
            let mut seq = TSeq::singleton(Literal::exact(vec![]));

            // Unroll the repetition via repeated cross product
            for _ in 0..std::cmp::min(min, limit) {
                if seq.is_inexact() {
                    break;
                }
                seq = self.cross(seq, subseq.clone());
            }
            // Exceeded limit means we didn't unroll completely
            if usize::try_from(min).is_err() || min > limit {
                seq.make_inexact();
            }
            seq
        }

        // Bounded range: a{3,5} -> inexact "aaa"
        hir::Repetition { min, max: Some(max), .. } if min < max => {
            assert!(min > 0);
            let limit = u32::try_from(self.limit_repeat).unwrap_or(u32::MAX);
            let mut seq = TSeq::singleton(Literal::exact(vec![]));

            for _ in 0..std::cmp::min(min, limit) {
                if seq.is_inexact() {
                    break;
                }
                seq = self.cross(seq, subseq.clone());
            }
            seq.make_inexact();  // Variable count = always inexact
            seq
        }
```

```
        // Unbounded: a+ -> inexact "a"        hir::Repetition { .. } => {
subseq.make_inexact();              subseq        }    }}
```

The `greedy` flag affects the order of alternatives in the union. For `a*` (greedy), we prefer matching `a` first; for `a*?` (lazy), we prefer the empty match first. This ordering propagates to how the literals are used in searching.

---

# Quick Reference

## TSeq (Tagged Sequence) Key Methods

| Method | Purpose |
|---|---|
| `is_exact()` | All literals are complete matches |
| `is_inexact()` | At least one literal is partial |
| `is_finite()` | Has enumerable literals |
| `is_good()` | Heuristically useful for acceleration |
| `is_really_good()` | Good enough to short-circuit extraction |
| `choose(other)` | Pick better sequence via heuristics |
| `make_not_prefix()` | Mark as not at pattern start |

## Heuristic Thresholds

```
// "Good" for acceleration:
min_literal_len >= 2 && count <= 64
// Or if very short literals:
min_literal_len <= 1 && count <= 3

// "Really good" (short-circuit worthy):
min_literal_len >= 3 && count <= 8
```

## Poisonous Literals

```
/// Literals likely to match too frequently
fn is_poisonous(lit: &Literal) -> bool {
    use regex_syntax::hir::literal::rank;
    // Empty string or high-frequency single byte
    lit.is_empty() || (lit.len() == 1 && rank(lit.as_bytes()[0]) >= 250)
}
```

The `rank` function returns a frequency score (0-255) based on how commonly a byte appears in typical text. Rank 250+ includes space, newline, and other ubiquitous characters.

## Extraction Limits

| Limit | Default | Purpose |
|---|---|---|
| `limit_class` | 10 | Max chars from `[abc...]` |
| `limit_repeat` | 10 | Max unroll iterations for `a{n}` |
| `limit_literal_len` | 100 | Max bytes per literal |
| `limit_total` | 64 | Max literals in sequence |

# ripgrep crates/regex/src/ast.rs: Code Companion

Reference code for the AST Manipulation lecture. Sections correspond to the lecture document.

---

## Section 1: The Analysis Result Structure

```rust
use regex_syntax::ast::{self, Ast};

/// The results of analyzing AST of a regular expression (e.g., for supporting
/// smart case).
#[derive(Clone, Debug)]
pub(crate) struct AstAnalysis {
    /// True if and only if a literal uppercase character occurs in the regex.
    any_uppercase: bool,
    /// True if and only if the regex contains any literal at all.
    any_literal: bool,
}
```

The `pub(crate)` visibility restricts this struct to the current crate only. The `regex_syntax` crate provides the AST types we'll traverse.

---

## Section 2: The Entry Points

```
impl AstAnalysis {
    /// Returns a `AstAnalysis` value by doing analysis on the AST of `pattern`.
    ///
    /// If `pattern` is not a valid regular expression, then `None` is
    /// returned.
    #[cfg(test)]  // Only compiled in test builds
    pub(crate) fn from_pattern(pattern: &str) -> Option<AstAnalysis> {
        regex_syntax::ast::parse::Parser::new()
            .parse(pattern)           // Returns Result<Ast, Error>
            .map(|ast| AstAnalysis::from_ast(&ast))  // Transform Ok variant
            .ok()                     // Convert Result to Option, discarding
error
    }

    /// Perform an AST analysis given the AST.
    pub(crate) fn from_ast(ast: &Ast) -> AstAnalysis {
        let mut analysis = AstAnalysis::new();  // Start with both flags false
        analysis.from_ast_impl(ast);            // Mutate during traversal
        analysis
    }
}
```

The `from_pattern` method chains parser creation, parsing, mapping, and error handling in a single expression. The `from_ast` method follows a common Rust pattern: create default state, mutate it, return it.

## Section 3: The Accessor Methods

```
impl AstAnalysis {
    /// Returns true if and only if a literal uppercase character occurs in
    /// the pattern.
    ///
    /// For example, a pattern like `\pL` contains no uppercase literals,
    /// even though `L` is uppercase and the `\pL` class contains uppercase
    /// characters.
    pub(crate) fn any_uppercase(&self) -> bool {
        self.any_uppercase
    }

    /// Returns true if and only if the regex contains any literal at all.
    ///
    /// For example, a pattern like `\pL` reports `false`, but a pattern like
    /// `\pLfoo` reports `true`.
    pub(crate) fn any_literal(&self) -> bool {
        self.any_literal
    }

    /// Creates a new `AstAnalysis` value with an initial configuration.
    fn new() -> AstAnalysis {
        AstAnalysis { any_uppercase: false, any_literal: false }
    }
}
```

Accessors return `bool` directly rather than `&bool` since `bool` is `Copy`. The `new()` constructor is private—callers must use `from_ast` or `from_pattern`.

---

## Section 4: The Tree Traversal Strategy

```
fn from_ast_impl(&mut self, ast: &Ast) {
    // Early termination: stop if we've found everything
    if self.done() {
        return;
    }
    match *ast {
        // Nodes with no literals - do nothing
        Ast::Empty(_) => {}
        Ast::Flags(_)
        | Ast::Dot(_)
        | Ast::Assertion(_)
        | Ast::ClassUnicode(_)    // \pL, \p{Greek}, etc.
        | Ast::ClassPerl(_) => {} // \d, \w, \s, etc.

        // Nodes containing literals directly
        Ast::Literal(ref x) => {
            self.from_ast_literal(x);
        }
        Ast::ClassBracketed(ref x) => {
            self.from_ast_class_set(&x.kind);  // [a-z], [^0-9], etc.
        }

        // Nodes containing sub-trees - recurse
        Ast::Repetition(ref x) => {
            self.from_ast_impl(&x.ast);  // a*, a+, a{3}, etc.
        }
        Ast::Group(ref x) => {
            self.from_ast_impl(&x.ast);  // (a), (?:a), (?P<name>a)
        }
        Ast::Alternation(ref alt) => {
            for x in &alt.asts {          // a|b|c
                self.from_ast_impl(x);
            }
        }
        Ast::Concat(ref alt) => {
            for x in &alt.asts {          // abc (sequence)
                self.from_ast_impl(x);
            }
        }
    }
}
```

The `match *ast` dereferences to match on the enum directly. The `ref` keyword borrows the inner data without moving it out of the AST.

## Section 5: Handling Character Classes

```
fn from_ast_class_set(&mut self, ast: &ast::ClassSet) {
    if self.done() {
        return;
    }
    match *ast {
        ast::ClassSet::Item(ref item) => {
            self.from_ast_class_set_item(item);
        }
        ast::ClassSet::BinaryOp(ref x) => {
            // Intersection: [a-z&&[aeiou]]
            // Subtraction: [a-z--[aeiou]]
            self.from_ast_class_set(&x.lhs);
            self.from_ast_class_set(&x.rhs);
        }
    }
}

fn from_ast_class_set_item(&mut self, ast: &ast::ClassSetItem) {
    if self.done() {
        return;
    }
    match *ast {
        // Built-in classes have no user literals
        ast::ClassSetItem::Empty(_)
        | ast::ClassSetItem::Ascii(_)   // [:alpha:], [:digit:]
        | ast::ClassSetItem::Unicode(_) // \p{Greek}
        | ast::ClassSetItem::Perl(_) => {}  // \d, \w

        // Single literal in class: [a] or [abc]
        ast::ClassSetItem::Literal(ref x) => {
            self.from_ast_literal(x);
        }
        // Range: [a-z] has two literals (start and end)
        ast::ClassSetItem::Range(ref x) => {
            self.from_ast_literal(&x.start);
            self.from_ast_literal(&x.end);
        }
        // Nested brackets: [[a-z][0-9]]
        ast::ClassSetItem::Bracketed(ref x) => {
            self.from_ast_class_set(&x.kind);
        }
        // Union of items: [abc0-9]
        ast::ClassSetItem::Union(ref union) => {
            for x in &union.items {
                self.from_ast_class_set_item(x);
            }
        }
    }
}
```

Character classes have their own mini-AST with items, ranges, and operations. The traversal mirrors this structure with dedicated methods.

---

## Section 6: Literal Detection

```
fn from_ast_literal(&mut self, ast: &ast::Literal) {
    self.any_literal = true;
    // is_uppercase() handles full Unicode, not just ASCII
    self.any_uppercase = self.any_uppercase || ast.c.is_uppercase();
}
```

The `ast.c` field is a `char`. The `||` short-circuits: once `any_uppercase` is true, `is_uppercase()` won't be called.

---

## Section 7: The Early Termination Condition

```
/// Returns true if and only if the attributes can never change no matter
/// what other AST it might see.
fn done(&self) -> bool {
    self.any_uppercase && self.any_literal
}
```

Both flags must be true to stop early. This is a **monotonic** analysis—flags only transition `false → true`, never backwards.

---

## Section 8: Test Coverage and Edge Cases

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

    fn analysis(pattern: &str) -> AstAnalysis {
        AstAnalysis::from_pattern(pattern).unwrap()
    }

    #[test]
    fn various() {
        // Empty pattern: no literals at all
        let x = analysis("");
        assert!(!x.any_uppercase);
        assert!(!x.any_literal);

        // Lowercase only
        let x = analysis("foo");
        assert!(!x.any_uppercase);
        assert!(x.any_literal);

        // Uppercase present
        let x = analysis("Foo");
        assert!(x.any_uppercase);
        assert!(x.any_literal);

        // Key distinction: \S (escape) vs \\S (literal backslash + S)
        let x = analysis(r"foo\S");      // \S is a metacharacter
        assert!(!x.any_uppercase);
        assert!(x.any_literal);

        let x = analysis(r"foo\\S");     // Literal \ followed by literal S
        assert!(x.any_uppercase);
        assert!(x.any_literal);

        // Unicode property class - no literals despite uppercase L
        let x = analysis(r"\p{Ll}");
        assert!(!x.any_uppercase);
        assert!(!x.any_literal);

        // Character class ranges: [A-Z] contains uppercase boundaries
        let x = analysis(r"foo[A-Z]");
        assert!(x.any_uppercase);
        assert!(x.any_literal);

        // Unicode escape: \u0061 is 'a', still counts as literal
        let x = analysis(r"a\u0061");
        assert!(!x.any_uppercase);
        assert!(x.any_literal);
    }
}
```

The `r"..."` raw string syntax avoids double-escaping backslashes. Each test case targets a specific edge case in the analysis logic.

---

## Quick Reference

### AstAnalysis API

| Method | Returns | Purpose |
|---|---|---|
| `from_ast(&Ast)` | `AstAnalysis` | Production entry point |
| `from_pattern(&str)` | `Option<AstAnalysis>` | Test-only convenience method |
| `any_uppercase(&self)` | `bool` | Check for uppercase literals |
| `any_literal(&self)` | `bool` | Check for any literals |

### AST Node Categories

| Category | Examples | Contains Literals? |
|---|---|---|
| Empty/Meta | `Empty`, `Dot`, `Assertion` | No |
| Classes | `ClassUnicode`, `ClassPerl` | No (built-in) |
| Literal | `Literal` | Yes |
| Bracketed | `ClassBracketed` | Depends on contents |
| Containers | `Repetition`, `Group`, `Alternation`, `Concat` | Recurse to check |

### Smart Case Behavior

| Pattern | any_uppercase | any_literal | Case Mode |
|---|---|---|---|
| foo | false | true | Insensitive |
| Foo | true | true | Sensitive |
| \pL | false | false | (no literals) |
| [A-Z] | true | true | Sensitive |
| \S | false | false | (no literals) |
| \\S | true | true | Sensitive |

# ripgrep crates/searcher/src/lib.rs: Code Companion

Reference code for the Searcher Overview lecture. Sections correspond to the lecture document.

---

## Section 1: The Crate Documentation as Architecture Document

```
/*!
This crate provides an implementation of line oriented search, with optional
support for multi-line search.

# Brief overview

The principle type in this crate is a [`Searcher`], which can be configured
and built by a [`SearcherBuilder`]. A `Searcher` is responsible for reading
bytes from a source (e.g., a file), executing a search of those bytes using
a `Matcher` (e.g., a regex) and then reporting the results of that search to
a [`Sink`] (e.g., stdout). The `Searcher` itself is principally responsible
for managing the consumption of bytes from a source and applying a `Matcher`
over those bytes in an efficient way. The `Searcher` is also responsible for
inverting a search, counting lines, reporting contextual lines, detecting
binary data and even deciding whether or not to use memory maps.
*/
```

The `//!` doc comment syntax marks this as module-level documentation. The `[Searcher]` syntax creates links to types in generated documentation.

---

## Section 2: Public API Design Through Re-exports

```
pub use crate::{
    // Line iteration utilities from the lines module
    lines::{LineIter, LineStep},

    // Core searcher types and configuration
    searcher::{
        BinaryDetection,     // How to handle binary files
        ConfigError,         // Configuration validation errors
        Encoding,            // Text encoding settings
        MmapChoice,          // Memory-mapped file options
        Searcher,            // The main search executor
        SearcherBuilder,     // Builder for configuring Searcher
    },

    // Result handling types
    sink::{
        Sink,              // Trait for receiving search results
        SinkContext,       // Context lines around matches
        SinkContextKind,   // Before/after context distinction
        SinkError,         // Error type for sink operations
        SinkFinish,        // End-of-search summary data
        SinkMatch,         // Individual match data
        sinks,             // Module of convenience implementations
    },
};
```

The `crate::` prefix refers to the current crate's root. Users import these types directly from `grep_searcher::` without knowing the internal module structure.

## Section 3: The Builder Pattern Ecosystem

```
pub use crate::{
    searcher::{
        // The builder creates configured Searcher instances
        SearcherBuilder,
        // The built type with all configuration applied
        Searcher,
    },
    // ...
};
```

The builder pattern pairs visible here: `SearcherBuilder` constructs `Searcher`. This mirrors `RegexMatcherBuilder` / `RegexMatcher` from the regex crate.

## Section 4: The Sink Abstraction

```
pub use crate::{
    sink::{
        Sink,              // The core trait - defines callback methods
        SinkContext,       // Data passed for context lines
        SinkContextKind,   // Enum: Before, After, or Other
        SinkError,         // Errors that can occur in sink callbacks
        SinkFinish,        // Summary data at search completion
        SinkMatch,         // Data passed for each matching line
        sinks,             // Convenience implementations
    },
};
```

The supporting types (`SinkContext`, `SinkMatch`, etc.) carry data to the `Sink` trait's callback methods. This is a push-based streaming API rather than a pull-based collection.

## Section 5: Convenience Sinks and Closures

```
// From the documentation example - using the UTF8 convenience sink
use grep_searcher::sinks::UTF8;

// The closure signature: (line_number, line_content) -> Result<bool, Error>
// Return Ok(true) to continue, Ok(false) to stop early
Searcher::new().search_slice(&matcher, SHERLOCK, UTF8(|lnum, line| {
    // Process the match...
    Ok(true)  // Continue searching
}))?;
```

The `UTF8` sink wraps a closure, handling byte-to-string conversion. The `bool` return value provides early termination control—useful for "find first match" scenarios.

## Section 6: The Documentation Example Dissected

```
use {
    grep_matcher::Matcher,       // The trait (from grep-matcher crate)
    grep_regex::RegexMatcher,    // The implementation (from grep-regex crate)
    grep_searcher::Searcher,     // The search orchestrator
    grep_searcher::sinks::UTF8, // Convenience sink for UTF-8 text
};

// Test data as a byte slice
const SHERLOCK: &'static [u8] = b"...";

// Step 1: Build the matcher
let matcher = RegexMatcher::new(r"Doctor \w+")?;

// Step 2: Prepare result storage
let mut matches: Vec<(u64, String)> = vec![];

// Step 3: Execute the search
Searcher::new().search_slice(&matcher, SHERLOCK, UTF8(|lnum, line| {
    // The searcher found a line with a match, but we re-run
    // the matcher to get precise match boundaries
    let mymatch = matcher.find(line.as_bytes())?.unwrap();
    matches.push((lnum, line[mymatch].to_string()));
    Ok(true)
}))?;
```

Three crates compose: `grep_matcher` defines the interface, `grep_regex` provides the implementation, `grep_searcher` orchestrates the search. The nested `matcher.find()` call extracts precise match text from matching lines.

## Section 7: Module Organization and Visibility

```
// Macro definitions available to subsequent modules
#[macro_use]
mod macros;

// Internal modules (no pub = private to crate)
mod line_buffer;  // Buffered reading implementation
mod lines;        // Line iteration utilities
mod searcher;     // Core Searcher type and builder
mod sink;         // Sink trait and implementations

// Test utilities only compiled in test builds
#[cfg(test)]
mod testutil;
```

The `#[macro_use]` attribute must appear before modules that use the macros. The `#[cfg(test)]` conditional compilation excludes test utilities from release builds.

---

## Quick Reference

### Architecture Flow

```
Source (file/stdin/bytes)
        ↓
    Searcher (orchestration)
        ↓
    Matcher (pattern matching)
        ↓
    Sink (result handling)
```

### Key Types

| Type | Purpose |
|---|---|
| `Searcher` | Orchestrates search over byte sources |
| `SearcherBuilder` | Configures and constructs `Searcher` |
| `Sink` | Trait for receiving search results |
| `SinkMatch` | Data for a matching line |
| `SinkContext` | Data for context lines |
| `UTF8` | Convenience sink wrapping a closure |

### Re-export Pattern

```rust
// Internal structure:
mod internal_module {
    pub struct SomeType;
}

// Public API (in lib.rs):
pub use crate::internal_module::SomeType;

// User sees:
use my_crate::SomeType;  // Clean, flat path
```

## Closure Sink Return Values

| Return | Effect |
|---|---|
| `Ok(true)` | Continue searching |
| `Ok(false)` | Stop search early |
| `Err(e)` | Abort with error |

# ripgrep crates/searcher/src/line_buffer.rs: Code Companion

Reference code for the Efficient Line Reading lecture. Sections correspond to the lecture document.

---

## Section 1: The Builder Pattern in Action

```rust
/// The configuration of a buffer. This contains options that are fixed once
/// a buffer has been constructed.
#[derive(Clone, Copy, Debug)]  // All fields are primitives/small enums - cheap
to copy
struct Config {
    capacity: usize,
    lineterm: u8,
    buffer_alloc: BufferAllocation,
    binary: BinaryDetection,
}

impl Default for Config {
    fn default() -> Config {
        Config {
            capacity: DEFAULT_BUFFER_CAPACITY,  // 64 KB
            lineterm: b'\n',
            buffer_alloc: BufferAllocation::default(),
            binary: BinaryDetection::default(),
        }
    }
}

/// A builder for constructing line buffers.
#[derive(Clone, Debug, Default)]
pub(crate) struct LineBufferBuilder {
    config: Config,
}

impl LineBufferBuilder {
    pub(crate) fn new() -> LineBufferBuilder {
        LineBufferBuilder { config: Config::default() }
    }

    /// Create a new line buffer from this builder's configuration.
    pub(crate) fn build(&self) -> LineBuffer {
        LineBuffer {
            config: self.config,  // Copies config, doesn't move - builder
reusable
            buf: vec![0; self.config.capacity],
            pos: 0,
            last_lineterm: 0,
            end: 0,
            absolute_byte_offset: 0,
            binary_byte_offset: None,
        }
    }

    /// Each setter returns &mut Self for method chaining
    pub(crate) fn capacity(
        &mut self,
        capacity: usize,
    ) -> &mut LineBufferBuilder {
        self.config.capacity = capacity;
        self
```

```
    }    pub(crate) fn line_terminator(        &mut self,        lineterm: u8,
) -> &mut LineBufferBuilder {        self.config.lineterm = lineterm;        self
}}
```

The `build()` method takes `&self` rather than `self`, allowing multiple buffers to be created from the same builder configuration.

---

## Section 2: Memory Allocation Strategies

```rust
/// The default buffer capacity that we use for the line buffer.
pub(crate) const DEFAULT_BUFFER_CAPACITY: usize = 64 * (1 << 10); // 64 KB

/// Controls additional memory allocation beyond initial capacity
#[derive(Clone, Copy, Debug)]
pub(crate) enum BufferAllocation {
    /// Grow without limit to fit lines (default)
    Eager,
    /// Limit *additional* memory to this many bytes; error if exceeded
    Error(usize),
}

impl Default for BufferAllocation {
    fn default() -> BufferAllocation {
        BufferAllocation::Eager
    }
}

/// Create a new error to be used when a configured allocation limit has been
/// reached.
pub(crate) fn alloc_error(limit: usize) -> io::Error {
    let msg = format!("configured allocation limit ({}) exceeded", limit);
    io::Error::new(io::ErrorKind::Other, msg)
}

// In ensure_capacity():
fn ensure_capacity(&mut self) -> Result<(), io::Error> {
    if !self.free_buffer().is_empty() {
        return Ok(());
    }
    let len = std::cmp::max(1, self.buf.len());
    let additional = match self.config.buffer_alloc {
        BufferAllocation::Eager => len * 2,  // Double the buffer
        BufferAllocation::Error(limit) => {
            let used = self.buf.len() - self.config.capacity;  // Already
allocated beyond initial
            let n = std::cmp::min(len * 2, limit - used);
            if n == 0 {
                return Err(alloc_error(self.config.capacity + limit));
            }
            n
        }
    };
    self.buf.resize(newlen, 0);
    Ok(())
}
```

The `Error(0)` case guarantees zero additional allocation—the buffer will never grow beyond its initial capacity, providing predictable memory usage.

## Section 3: Binary Detection Heuristics

```
/// The behavior of binary detection in the line buffer.
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
pub(crate) enum BinaryDetection {
    /// No detection - treat everything as text (default)
    None,
    /// Stop reading when this byte is found (typically b'\x00')
    Quit(u8),
    /// Replace this byte with line terminator, continue reading
    Convert(u8),
}

impl BinaryDetection {
    /// Returns true if detection demands early EOF on binary data
    fn is_quit(&self) -> bool {
        match *self {
            BinaryDetection::Quit(_) => true,
            _ => false,
        }
    }
}

// Inside the fill() method - binary detection logic:
match self.config.binary {
    BinaryDetection::None => {} // nothing to do
    BinaryDetection::Quit(byte) => {
        if let Some(i) = newbytes.find_byte(byte) {
            self.end = oldend + i;  // Truncate at binary byte
            self.last_lineterm = self.end;
            self.binary_byte_offset =
                Some(self.absolute_byte_offset + self.end as u64);
            // Report whether any data remains
            return Ok(self.pos < self.end);
        }
    }
    BinaryDetection::Convert(byte) => {
        if let Some(i) =
            replace_bytes(newbytes, byte, self.config.lineterm)
        {
            // Record only the first binary offset
            if self.binary_byte_offset.is_none() {
                self.binary_byte_offset = Some(
                    self.absolute_byte_offset + (oldend + i) as u64,
                );
            }
        }
    }
}
```

The `binary_byte_offset` is an absolute position in the input stream, suitable for user-facing error messages that report where binary data was found.

---

## Section 4: The Buffer State Machine

```rust
/// A line buffer manages a (typically fixed) buffer for holding lines.
#[derive(Clone, Debug)]
pub(crate) struct LineBuffer {
    config: Config,
    /// The primary buffer with which to hold data.
    buf: Vec<u8>,
    /// Current position - start of unconsumed data (valid index into buf)
    pos: usize,
    /// End of complete lines - where buffer() slice ends
    last_lineterm: usize,
    /// End of all valid data (>= last_lineterm, bytes after are partial line)
    end: usize,
    /// Logical offset from start of input stream (not a memory address)
    absolute_byte_offset: u64,
    /// Absolute offset where binary data was first detected
    binary_byte_offset: Option<u64>,
}

impl LineBuffer {
    /// Return the contents of this buffer - only complete lines
    fn buffer(&self) -> &[u8] {
        &self.buf[self.pos..self.last_lineterm]
    }

    /// Return free space for reading new data
    fn free_buffer(&mut self) -> &mut [u8] {
        &mut self.buf[self.end..]
    }

    /// Consume bytes - advances position and absolute offset
    fn consume(&mut self, amt: usize) {
        assert!(amt <= self.buffer().len());
        self.pos += amt;
        self.absolute_byte_offset += amt as u64;
    }

    /// Reset for reuse with a new reader
    fn clear(&mut self) {
        self.pos = 0;
        self.last_lineterm = 0;
        self.end = 0;
        self.absolute_byte_offset = 0;
        self.binary_byte_offset = None;
    }
}
```

The three-position design (`pos`, `last_lineterm`, `end`) separates consumed data, complete lines ready for processing, and incomplete trailing data that needs more input.

# Section 5: The Fill and Roll Cycle

```
fn fill<R: io::Read>(&mut self, mut rdr: R) -> Result<bool, io::Error> {
    // Early exit if binary detection already triggered EOF
    if self.config.binary.is_quit() && self.binary_byte_offset.is_some() {
        return Ok(!self.buffer().is_empty());
    }

    self.roll();  // Move unconsumed data to front
    assert_eq!(self.pos, 0);  // Sanity check: roll always resets pos

    loop {
        self.ensure_capacity()?;
        let readlen = rdr.read(self.free_buffer().as_bytes_mut())?;
        if readlen == 0 {
            // EOF: expose any remaining data as the final "line"
            self.last_lineterm = self.end;
            return Ok(!self.buffer().is_empty());
        }

        let oldend = self.end;
        self.end += readlen;
        let newbytes = &mut self.buf[oldend..self.end];

        // [Binary detection happens here - see Section 3]

        // Look for line terminator in newly read bytes
        if let Some(i) = newbytes.rfind_byte(self.config.lineterm) {
            self.last_lineterm = oldend + i + 1;  // Include the terminator
            return Ok(true);
        }
        // No complete line yet - keep reading
    }
}

/// Roll unconsumed bytes to front of buffer, freeing space at end
fn roll(&mut self) {
    if self.pos == self.end {
        // Buffer fully consumed - just reset
        self.pos = 0;
        self.last_lineterm = 0;
        self.end = 0;
        return;
    }

    let roll_len = self.end - self.pos;
    self.buf.copy_within(self.pos..self.end, 0);  // Efficient in-place move
    self.pos = 0;
    self.last_lineterm = roll_len;
    self.end = roll_len;
}
```

The `roll()` operation uses `copy_within` for efficient in-place byte movement, avoiding allocation. After rolling, `last_lineterm` equals `end` because the partial line is now at the buffer's end.

---

## Quick Reference

### Buffer Position Diagram

```
buf: [consumed | complete lines | partial line | free space]
      ^         ^                ^              ^
      0         pos              last_lineterm  end         len

After roll():
buf: [complete lines + partial | free space              ]
      ^                        ^
      pos=0                    last_lineterm=end         len
```

### Key Methods

| Method | Returns | Purpose |
|---|---|---|
| `buffer()` | `&[u8]` | Complete lines ready for processing |
| `free_buffer()` | `&mut [u8]` | Space available for reading |
| `consume(amt)` | `()` | Mark bytes as processed |
| `fill(&mut rdr)` | `Result<bool, io::Error>` | Read until complete line or EOF |
| `roll()` | `()` | Move unconsumed data to buffer front |

### Binary Detection Summary

| Variant | On Binary Byte | Continues Reading | Records Offset |
|---|---|---|---|

| | | | |
|---|---|---|---|
| `None` | No action | Yes | No |
| `Quit(byte)` | Acts as EOF | No | Yes |
| `Convert(byte)` | Replace with `\n` | Yes | Yes (first only) |

## Default Configuration

```
Config {
    capacity: 64 * 1024,           // 64 KB
    lineterm: b'\n',               // Unix line endings
    buffer_alloc: Eager,           // Grow without limit
    binary: BinaryDetection::None, // No detection
}
```

# ripgrep crates/searcher/src/lines.rs: Code Companion

Reference code for the Line Handling Utilities lecture. Sections correspond to the lecture document.

---

## Section 1: The Two Iterator Designs

```rust
/// An iterator over lines in a particular slice of bytes.
///
/// Line terminators are considered part of the line they terminate. All lines
/// yielded by the iterator are guaranteed to be non-empty.
///
/// `'b` refers to the lifetime of the underlying bytes.
#[derive(Debug)]
pub struct LineIter<'b> {
    bytes: &'b [u8],       // Borrows the slice - limits what caller can do
    stepper: LineStep,      // Delegates to LineStep for actual logic
}

impl<'b> Iterator for LineIter<'b> {
    type Item = &'b [u8];

    fn next(&mut self) -> Option<&'b [u8]> {
        // Converts Match ranges back to actual slices
        self.stepper.next_match(self.bytes).map(|m| &self.bytes[m])
    }
}

/// An explicit iterator over lines in a particular slice of bytes.
///
/// This iterator avoids borrowing the bytes themselves, and instead requires
/// callers to explicitly provide the bytes when moving through the iterator.
#[derive(Debug)]
pub struct LineStep {
    line_term: u8,    // Single byte terminator for fast searching
    pos: usize,       // Current position in iteration
    end: usize,       // Boundary - won't iterate past this
}

impl LineStep {
    /// Create a new line iterator over the given range of bytes.
    /// Callers must provide the actual bytes for each call to `next`.
    pub fn new(line_term: u8, start: usize, end: usize) -> LineStep {
        LineStep { line_term, pos: start, end }
    }

    /// Return the start and end position of the next line.
    /// The caller must pass exactly the same slice of bytes for each call.
    pub fn next(&mut self, bytes: &[u8]) -> Option<(usize, usize)> {
        self.next_impl(bytes)
    }
}
```

`LineIter` provides standard iterator ergonomics but holds a borrow. `LineStep` stores only position state, allowing callers to manage buffer lifetime independently—essential for complex buffer refilling scenarios.

## Section 2: The Line Terminator Philosophy

```
use {
    bstr::ByteSlice,
    grep_matcher::{LineTerminator, Match},
};

// LineStep uses a single u8 for the terminator
pub struct LineStep {
    line_term: u8,    // Just '\n', not the full "\r\n" sequence
    pos: usize,
    end: usize,
}

// But without_terminator accepts the full LineTerminator type
pub(crate) fn without_terminator(
    bytes: &[u8],
    line_term: LineTerminator,  // Full type from grep_matcher
) -> &[u8] {
    let line_term = line_term.as_bytes();  // Could be b"\n" or b"\r\n"
    // ... stripping logic
}
```

Single-byte search is faster than multi-byte sequence matching. Finding `\n` locates line boundaries regardless of whether the actual terminator is `\n` or `\r\n`. The full `LineTerminator` type is only needed when stripping terminators from display output.

---

## Section 3: The Core Iteration Logic

```rust
#[inline(always)]
fn next_impl(&mut self, mut bytes: &[u8]) -> Option<(usize, usize)> {
    // Clip to configured end position - bytes slice may be larger
    bytes = &bytes[..self.end];

    // find_byte is SIMD-accelerated via bstr crate
    match bytes[self.pos..].find_byte(self.line_term) {
        None => {
            // No terminator found - handle final unterminated line
            if self.pos < bytes.len() {
                let m = (self.pos, bytes.len());
                assert!(m.0 <= m.1);  // Debug check: range is valid

                self.pos = m.1;       // Mark iteration complete
                Some(m)
            } else {
                None                  // Already consumed everything
            }
        }
        Some(line_end) => {
            // Found terminator at relative position line_end
            // Range includes the terminator byte (+1)
            let m = (self.pos, self.pos + line_end + 1);
            assert!(m.0 <= m.1);      // Debug check: range is valid

            self.pos = m.1;           // Advance past terminator
            Some(m)
        }
    }
}
```

The `assert!` statements catch range inversion bugs in debug builds but compile away in release. The `+1` ensures line terminators are included in the yielded range.

---

## Section 4: Counting Lines Efficiently

```rust
/// Count the number of occurrences of `line_term` in `bytes`.
pub(crate) fn count(bytes: &[u8], line_term: u8) -> u64 {
    // memchr_iter: SIMD-accelerated iterator over all occurrences
    // Contains thousands of lines of architecture-specific assembly
    memchr::memchr_iter(line_term, bytes).count() as u64
}
```

One line of code, backed by extensive CPU-specific optimizations in the `memchr` crate. Returns `u64` to handle extremely large files or files with very short lines.

---

## Section 5: Stripping Line Terminators

```
/// Given a line that possibly ends with a terminator, return that line without
/// the terminator.
#[inline(always)]
pub(crate) fn without_terminator(
    bytes: &[u8],
    line_term: LineTerminator,
) -> &[u8] {
    let line_term = line_term.as_bytes();  // b"\n" or b"\r\n"

    // saturating_sub prevents underflow if bytes is shorter than terminator
    let start = bytes.len().saturating_sub(line_term.len());

    // Check if bytes actually ends with the terminator
    if bytes.get(start..) == Some(line_term) {
        // Strip the terminator
        return &bytes[..bytes.len() - line_term.len()];
    }

    // No terminator present - return unchanged
    bytes
}
```

The `saturating_sub` handles edge cases where `bytes.len()` is smaller than `line_term.len()`. For CRLF endings, this correctly strips both `\r` and `\n`.

---

## Section 6: Locating Lines Around Matches

```rust
/// Return the start and end offsets of the lines containing the given range
/// of bytes.
///
/// Line terminators are considered part of the line they terminate.
#[inline(always)]
pub(crate) fn locate(bytes: &[u8], line_term: u8, range: Match) -> Match {
    // Search backward for preceding line terminator
    // map_or(0, |i| i + 1): if not found, line starts at 0
    //                       if found at i, line starts at i + 1
    let line_start =
        bytes[..range.start()].rfind_byte(line_term).map_or(0, |i| i + 1);

    let line_end =
        // Optimization: if match already ends at a terminator, don't search
        if range.end() > line_start && bytes[range.end() - 1] == line_term {
            range.end()
        } else {
            // Search forward for next line terminator
            bytes[range.end()..]
                .find_byte(line_term)
                .map_or(bytes.len(), |i| range.end() + i + 1)
        };

    Match::new(line_start, line_end)
}
```

The bidirectional search expands any match range to encompass complete lines. The optimization checking `bytes[range.end() - 1] == line_term` avoids unnecessary forward scanning when the match already ends at a line boundary.

---

## Section 7: Navigating Backwards for Context

```rust
/// Returns the minimal starting offset of the line that occurs `count` lines
/// before the last line in `bytes`.
pub(crate) fn preceding(bytes: &[u8], line_term: u8, count: usize) -> usize {
    preceding_by_pos(bytes, bytes.len(), line_term, count)
}

fn preceding_by_pos(
    bytes: &[u8],
    mut pos: usize,
    line_term: u8,
    mut count: usize,
) -> usize {
    if pos == 0 {
        return 0;
    // Handle position immediately after a terminator
    } else if bytes[pos - 1] == line_term {
        pos -= 1;  // Step back to treat terminator as part of previous line
    }

    loop {
        // Search backward for line terminator
        match bytes[..pos].rfind_byte(line_term) {
            None => {
                return 0;  // No more terminators - start of buffer
            }
            Some(i) => {
                if count == 0 {
                    return i + 1;  // Found enough lines - return start
                } else if i == 0 {
                    return 0;      // At buffer start
                }
                count -= 1;
                pos = i;           // Continue searching backward
            }
        }
    }
}
```

This function enables context display (showing N lines before a match). The special handling of `bytes[pos - 1] == line_term` correctly treats a position immediately after a newline as belonging to the preceding line.

## Quick Reference

| Function | Purpose | Returns |
| --- | --- | --- |

| | | |
|---|---|---|
| `LineIter::new(term, bytes)` | Standard line iterator | `Iterator<Item = &[u8]>` |
| `LineStep::new(term, start, end)` | Non-borrowing iterator | Position-only state |
| `LineStep::next(bytes)` | Get next line range | `Option<(usize, usize)>` |
| `count(bytes, term)` | Count line terminators | `u64` |
| `without_terminator(bytes, term)` | Strip trailing terminator | `&[u8]` |
| `locate(bytes, term, range)` | Expand match to full lines | `Match` |
| `preceding(bytes, term, count)` | Find N lines before end | `usize` |

## Key Type Signatures

```
// From grep_matcher - represents a byte range
pub struct Match { start: usize, end: usize }

// Full terminator (handles CRLF)
pub struct LineTerminator { /* b'\n' or b"\r\n" */ }
impl LineTerminator {
    fn as_bytes(&self) -> &[u8];
}
```

## Performance Notes

- `find_byte` / `rfind_byte`: SIMD-accelerated via `bstr`
- `memchr_iter`: SIMD-accelerated via `memchr`
- Single-byte terminator search is faster than sequence matching
- `#[inline(always)]` on hot paths ensures no function call overhead

# ripgrep crates/searcher/src/searcher/mod.rs: Code Companion

Reference code for the Searcher Core lecture. Sections correspond to the lecture document.

---

## Section 1: The Builder Pattern in Action

```rust
/// A builder for configuring a searcher.
///
/// Once a searcher has been built, it is beneficial to reuse that searcher
/// for multiple searches, if possible.
#[derive(Clone, Debug)]
pub struct SearcherBuilder {
    config: Config,
}

impl Default for SearcherBuilder {
    fn default() -> SearcherBuilder {
        SearcherBuilder::new()
    }
}

impl SearcherBuilder {
    /// Create a new searcher builder with a default configuration.
    pub fn new() -> SearcherBuilder {
        SearcherBuilder { config: Config::default() }
    }

    /// Build a searcher with the given matcher.
    pub fn build(&self) -> Searcher {
        let mut config = self.config.clone();
        // Normalize passthru mode: context makes no sense when passing through
all lines
        if config.passthru {
            config.before_context = 0;
            config.after_context = 0;
        }

        // Configure the transcoding layer
        let mut decode_builder = DecodeReaderBytesBuilder::new();
        decode_builder
            .encoding(self.config.encoding.as_ref().map(|e| e.0))
            .utf8_passthru(true)  // Pass UTF-8 through unchanged
            .strip_bom(self.config.bom_sniffing)
            .bom_override(true)   // BOM takes precedence over explicit encoding
            .bom_sniffing(self.config.bom_sniffing);

        Searcher {
            config,
            decode_builder,
            // Pre-allocate 8KB for transcoding scratch space
            decode_buffer: RefCell::new(vec![0; 8 * (1 << 10)]),
            line_buffer: RefCell::new(self.config.line_buffer()),
            multi_line_buffer: RefCell::new(vec![]),
        }
    }

    /// Example of a fluent builder method returning &mut Self
    pub fn line_terminator(
        &mut self,
        line_term: LineTerminator,
```

```
    ) -> &mut SearcherBuilder {        self.config.line_term = line_term;
self    }    // ... additional builder methods follow the same pattern}
```

The `build` method performs normalization (passthru disables context) and initializes all infrastructure the `Searcher` needs. This ensures the returned searcher is immediately ready for use.

---

## Section 2: Configuration as a First-Class Concept

```rust
/// The internal configuration of a searcher. This is shared among several
/// search related types, but is only ever written to by the SearcherBuilder.
#[derive(Clone, Debug)]
pub struct Config {
    line_term: LineTerminator,
    invert_match: bool,
    after_context: usize,
    before_context: usize,
    passthru: bool,
    line_number: bool,
    /// When None, no explicit limit. When Some(0), only mmap strategy available.
    heap_limit: Option<usize>,
    mmap: MmapChoice,
    binary: BinaryDetection,
    multi_line: bool,
    encoding: Option<Encoding>,
    bom_sniffing: bool,
    stop_on_nonmatch: bool,
    max_matches: Option<u64>,
}

impl Default for Config {
    fn default() -> Config {
        Config {
            line_term: LineTerminator::default(),
            invert_match: false,
            after_context: 0,
            before_context: 0,
            passthru: false,
            line_number: true,          // Enabled by default
            heap_limit: None,           // No limit by default
            mmap: MmapChoice::default(),
            binary: BinaryDetection::default(),  // Disabled by default
            multi_line: false,
            encoding: None,
            bom_sniffing: true,         // Enabled by default
            stop_on_nonmatch: false,
            max_matches: None,
        }
    }
}

impl Config {
    /// Return the maximal amount of lines needed to fulfill this
    /// configuration's context.
    fn max_context(&self) -> usize {
        cmp::max(self.before_context, self.after_context)
    }

    /// Build a line buffer from this configuration.
    fn line_buffer(&self) -> LineBuffer {
        let mut builder = LineBufferBuilder::new();
        builder
            .line_terminator(self.config.line_term.as_byte())
```

```
            .binary_detection(self.config.binary.0);        // Configure heap
limits if set        if let Some(limit) = self.heap_limit {            let
(capacity, additional) = if limit <= DEFAULT_BUFFER_CAPACITY {
(limit, 0)            } else {                // Split limit between initial
capacity and growth allowance                (DEFAULT_BUFFER_CAPACITY, limit -
DEFAULT_BUFFER_CAPACITY)            };            builder
.capacity(capacity)
.buffer_alloc(BufferAllocation::Error(additional));        }
builder.build()    }}
```

The `Config` struct centralizes all search parameters. The `line_buffer` method shows how configuration translates into concrete infrastructure, properly handling heap limits by splitting them between initial capacity and growth allowance.

## Section 3: Binary Detection Strategies

```rust
/// The behavior of binary detection while searching.
#[derive(Clone, Debug, Default, Eq, PartialEq)]
pub struct BinaryDetection(line_buffer::BinaryDetection);

impl BinaryDetection {
    /// No binary detection is performed. Data reported by the searcher may
    /// contain arbitrary bytes. This is the default.
    pub fn none() -> BinaryDetection {
        BinaryDetection(line_buffer::BinaryDetection::None)
    }

    /// Binary detection is performed by looking for the given byte.
    /// When found, the search stops as if it reached EOF.
    ///
    /// Behavior differs by search mode:
    /// - Fixed buffer: all content is checked
    /// - Memory map/heap: only matched regions + initial prefix checked
    pub fn quit(binary_byte: u8) -> BinaryDetection {
        BinaryDetection(line_buffer::BinaryDetection::Quit(binary_byte))
    }

    /// Binary detection replaces the given byte with the line terminator.
    ///
    /// With fixed buffer: caller never observes this byte
    /// With memory map: this setting has no effect (data is read-only)
    pub fn convert(binary_byte: u8) -> BinaryDetection {
        BinaryDetection(line_buffer::BinaryDetection::Convert(binary_byte))
    }

    /// If this uses "quit" strategy, returns the trigger byte.
    pub fn quit_byte(&self) -> Option<u8> {
        match self.0 {
            line_buffer::BinaryDetection::Quit(b) => Some(b),
            _ => None,
        }
    }

    /// If this uses "convert" strategy, returns the byte to replace.
    pub fn convert_byte(&self) -> Option<u8> {
        match self.0 {
            line_buffer::BinaryDetection::Convert(b) => Some(b),
            _ => None,
        }
    }
}
```

`BinaryDetection` wraps the internal `line_buffer::BinaryDetection` type, providing a public API while keeping the implementation detail private. The three strategies (none, quit, convert) handle the inherent ambiguity of what constitutes "binary" data.

## Section 4: Encoding and Transcoding Support

```
/// An encoding to use when searching.
///
/// An `Encoding` will always be cheap to clone.
#[derive(Clone, Debug, Eq, PartialEq)]
pub struct Encoding(&'static encoding_rs::Encoding);

impl Encoding {
    /// Create a new encoding for the specified label.
    ///
    /// The encoding label is mapped via the Encoding Standard.
    /// Returns an error if the label doesn't correspond to a valid encoding.
    pub fn new(label: &str) -> Result<Encoding, ConfigError> {
        let label = label.as_bytes();
        match encoding_rs::Encoding::for_label_no_replacement(label) {
            Some(encoding) => Ok(Encoding(encoding)),
            None => {
                Err(ConfigError::UnknownEncoding { label: label.to_vec() })
            }
        }
    }
}

impl Searcher {
    /// Returns true if and only if the given slice needs to be transcoded.
    fn slice_needs_transcoding(&self, slice: &[u8]) -> bool {
        // Explicit encoding always requires transcoding
        self.config.encoding.is_some()
            // BOM sniffing enabled AND a BOM is actually present
            || (self.config.bom_sniffing && slice_has_bom(slice))
    }
}

/// Returns true if the slice begins with a UTF-8 or UTF-16 BOM.
fn slice_has_bom(slice: &[u8]) -> bool {
    let enc = match encoding_rs::Encoding::for_bom(slice) {
        None => return false,
        Some((enc, _)) => enc,
    };
    log::trace!("found byte-order mark (BOM) for encoding {enc:?}");
    // Only these three encodings are recognized via BOM
    [encoding_rs::UTF_16LE, encoding_rs::UTF_16BE, encoding_rs::UTF_8]
        .contains(&enc)
}
```

The `Encoding` type wraps `encoding_rs` to provide encapsulation. The `slice_needs_transcoding` method optimizes the common case: plain ASCII/UTF-8 data without a BOM can be searched

directly without transcoding overhead.

## Section 5: The Searcher Type and Interior Mutability

```
/// A searcher executes searches over a haystack and writes results to a caller
/// provided sink.
#[derive(Clone, Debug)]
pub struct Searcher {
    /// The configuration for this searcher.
    config: Config,
    /// A builder for constructing a streaming transcoder.
    /// When no transcoding is needed, passes through bytes unchanged.
    decode_builder: DecodeReaderBytesBuilder,
    /// Buffer for transcoding scratch space.
    decode_buffer: RefCell<Vec<u8>>,
    /// Line buffer for incremental line-oriented searching.
    ///
    /// We wrap it in RefCell to permit lending out borrows of `Searcher`
    /// to sinks. We still require a mutable borrow to execute a search, so
    /// we statically prevent callers from causing RefCell to panic at runtime
    /// due to a borrowing violation.
    line_buffer: RefCell<LineBuffer>,
    /// Buffer for storing entire contents when multi-line searching.
    /// Multi-line searches cannot be performed incrementally.
    multi_line_buffer: RefCell<Vec<u8>>,
}

impl Searcher {
    /// Create a new searcher with a default configuration.
    pub fn new() -> Searcher {
        SearcherBuilder::new().build()
    }

    /// Dynamically update binary detection (useful between searches).
    pub fn set_binary_detection(&mut self, detection: BinaryDetection) {
        self.config.binary = detection.clone();
        // Also update the line buffer's detection setting
        self.line_buffer.borrow_mut().set_binary_detection(detection.0);
    }
}
```

The `RefCell` wrappers enable a specific borrowing pattern: the searcher can lend references to its configuration to sink implementations while still mutating its buffers. The `&mut self` requirement on search methods prevents runtime borrow panics.

# Section 6: Search Strategy Selection

```
impl Searcher {
    /// Execute a search over the file with the given path.
    pub fn search_path<P, M, S>(
        &mut self,
        matcher: M,
        path: P,
        write_to: S,
    ) -> Result<(), S::Error>
    where
        P: AsRef<Path>,
        M: Matcher,
        S: Sink,
    {
        let path = path.as_ref();
        let file = File::open(path).map_err(S::Error::error_io)?;
        self.search_file_maybe_path(matcher, Some(path), &file, write_to)
    }

    fn search_file_maybe_path<M, S>(
        &mut self,
        matcher: M,
        path: Option<&Path>,
        file: &File,
        write_to: S,
    ) -> Result<(), S::Error>
    where
        M: Matcher,
        S: Sink,
    {
        // Strategy 1: Memory map if enabled and available
        if let Some(mmap) = self.config.mmap.open(file, path) {
            log::trace!("{:?}: searching via memory map", path);
            return self.search_slice(matcher, &mmap, write_to);
        }

        // Strategy 2: Multi-line fast path for files (pre-allocate based on file
size)
        if self.multi_line_with_matcher(&matcher) {
            log::trace!("{:?}: reading entire file on to heap for multiline",
path);
            self.fill_multi_line_buffer_from_file::<S>(file)?;
            log::trace!("{:?}: searching via multiline strategy", path);
            MultiLine::new(
                self,
                matcher,
                &*self.multi_line_buffer.borrow(),
                write_to,
            )
            .run()
        } else {
            // Strategy 3: Generic incremental reader
            log::trace!("{:?}: searching using generic reader", path);
            self.search_reader(matcher, file, write_to)
        }
```

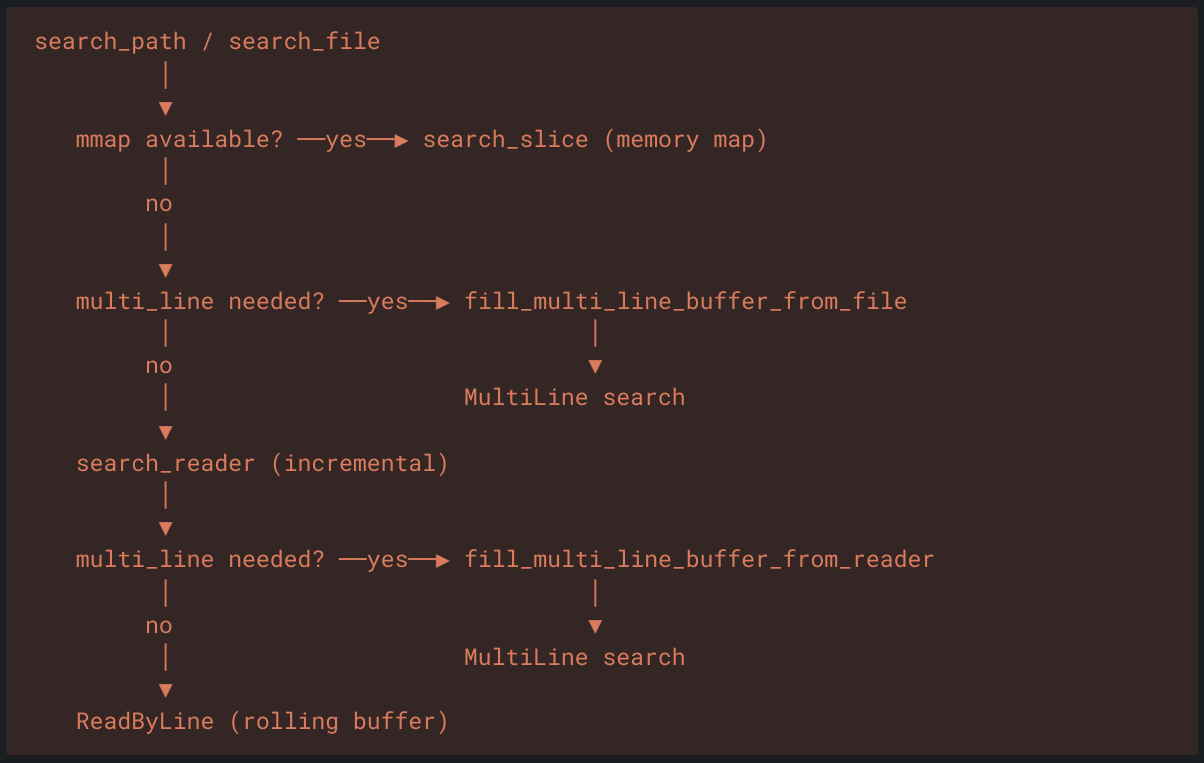
```
    }    /// Execute a search over any implementation of `std::io::Read`.    pub
fn search_reader<M, R, S>(        &mut self,        matcher: M,        read_from:
R,        write_to: S,    ) -> Result<(), S::Error>    where        M: Matcher,
R: io::Read,        S: Sink,    {
self.check_config(&matcher).map_err(S::Error::error_config)?;        // Set up
transcoding layer        let mut decode_buffer = self.decode_buffer.borrow_mut();
let decoder = self            .decode_builder
.build_with_buffer(read_from, &mut *decode_buffer)
.map_err(S::Error::error_io)?;        if self.multi_line_with_matcher(&matcher) {
// Multi-line: read everything into memory
self.fill_multi_line_buffer_from_reader::<_, S>(decoder)?;
MultiLine::new(                self,                matcher,
&*self.multi_line_buffer.borrow(),                write_to,            )
.run()        } else {            // Incremental: use rolling line buffer
let mut line_buffer = self.line_buffer.borrow_mut();            let rdr =
LineBufferReader::new(decoder, &mut *line_buffer);
ReadByLine::new(self, matcher, rdr, write_to).run()        }    }}
```

The search methods implement a tiered strategy: memory map first (fastest for large files already in memory), then file-specific multi-line (efficient pre-allocation), then generic incremental (works with any `Read`).

---

## Quick Reference

### Search Strategy Decision Tree




## Key Type Signatures

| Type | Purpose |
|---|---|
| `SearcherBuilder` | Fluent builder for `Searcher` configuration |
| `Searcher` | Main search orchestrator with reusable buffers |
| `Config` | All search parameters in one cloneable struct |
| `BinaryDetection` | Strategy for handling binary data (none/quit/convert) |
| `Encoding` | Wrapper around `encoding_rs::Encoding` |
| `ConfigError` | Build-time configuration errors |

## Search Methods

| Method | Input Type | Best For |
|---|---|---|
| `search_path` | `Path` | Files (enables mmap) |
| `search_file` | `&File` | Open file handles |
| `search_reader` | `impl Read` | Arbitrary streams |
| `search_slice` | `&[u8]` | In-memory data |

## Default Configuration

| Setting | Default | Notes |
|---|---|---|
| `line_number` | `true` | Small perf cost |
| `bom_sniffing` | `true` | Detects UTF-16 |
| `binary` | `none` | No detection |
| `mmap` | `never` | Explicit enable required |
| `multi_line` | `false` | Requires full file in memory |
| `heap_limit` | `None` | No limit |

# ripgrep crates/searcher/src/searcher/core.rs: Code Companion

Reference code for the Search Algorithm lecture. Sections correspond to the lecture document.

---

## Section 1: The Core State Machine

```rust
#[derive(Debug)]
pub(crate) struct Core<'s, M: 's, S> {
    // Configuration and dependencies (borrowed)
    config: &'s Config,
    matcher: M,
    searcher: &'s Searcher,
    sink: S,

    // Binary detection state
    binary: bool,
    binary_byte_offset: Option<usize>,

    // Position tracking - dual granularity
    pos: usize,                     // Position within current buffer
    absolute_byte_offset: u64,      // Position in the overall file

    // Line number optimization
    line_number: Option<u64>,       // Current line number (if tracking)
    last_line_counted: usize,       // Checkpoint for incremental counting
    last_line_visited: usize,       // Last line we processed

    // Context and match state
    after_context_left: usize,      // Remaining after-context lines to emit
    has_sunk: bool,                 // Have we output anything?
    has_matched: bool,              // Have we found any match?
    count: u64,                     // Total matches found
}
```

The struct is generic over `M: Matcher` and `S: Sink`, allowing different regex engines and output destinations. The `'s` lifetime ties borrowed references to the searcher's lifetime.

```
impl<'s, M: Matcher, S: Sink> Core<'s, M, S> {
    pub(crate) fn new(
        searcher: &'s Searcher,
        matcher: M,
        sink: S,
        binary: bool,
    ) -> Core<'s, M, S> {
        let line_number =
            if searcher.config.line_number { Some(1) } else { None };
        let core = Core {
            config: &searcher.config,
            matcher,
            searcher,
            sink,
            binary,
            pos: 0,
            absolute_byte_offset: 0,
            binary_byte_offset: None,
            line_number,
            last_line_counted: 0,
            last_line_visited: 0,
            after_context_left: 0,
            has_sunk: false,
            has_matched: false,
            count: 0,
        };
        // Log which search path will be used
        if !core.searcher.multi_line_with_matcher(&core.matcher) {
            if core.is_line_by_line_fast() {
                log::trace!("searcher core: will use fast line searcher");
            } else {
                log::trace!("searcher core: will use slow line searcher");
            }
        }
        core
    }
}
```

Line number tracking is optional (`Option<u64>`) to avoid counting overhead when not needed.

## Section 2: The Fast Path vs. Slow Path Decision

```rust
fn is_line_by_line_fast(&self) -> bool {
    debug_assert!(!self.searcher.multi_line_with_matcher(&self.matcher));

    // Passthrough mode requires slow path (must see every line)
    if self.config.passthru {
        return false;
    }
    // Stop-on-nonmatch after a match needs slow path
    if self.config.stop_on_nonmatch && self.has_matched {
        return false;
    }

    // Check if matcher understands line terminators
    if let Some(line_term) = self.matcher.line_terminator() {
        // FIXME: Workaround for grep-regex bug with NUL terminators
        // Line anchors like (?m:^) and (?m:$) don't work with \x00
        if line_term.as_byte() == b'\x00' {
            return false;
        }
        // Matcher's line terminator matches our config
        if line_term == self.config.line_term {
            return true;
        }
    }

    // Alternative: check non_matching_bytes
    if let Some(non_matching) = self.matcher.non_matching_bytes() {
        // For CRLF, we only care about \n (the required part)
        if non_matching.contains(self.config.line_term.as_byte()) {
            return true;
        }
    }
    false
}
```

The method returns `true` only when the matcher can correctly handle line boundaries during its search, enabling the optimized fast path.

```
pub(crate) fn match_by_line(
    &mut self,
    buf: &[u8],
) -> Result<bool, S::Error> {
    if self.is_line_by_line_fast() {
        match self.match_by_line_fast(buf)? {
            FastMatchResult::SwitchToSlow => self.match_by_line_slow(buf),
            FastMatchResult::Continue => Ok(true),
            FastMatchResult::Stop => Ok(false),
        }
    } else {
        self.match_by_line_slow(buf)
    }
}
```

The `FastMatchResult` enum allows the fast path to hand off to the slow path mid-search when conditions change.

---

## Section 3: The Fast Line Search Algorithm

```rust
#[inline(always)]
fn find_by_line_fast(
    &mut self,
    buf: &[u8],
) -> Result<Option<Range>, S::Error> {
    debug_assert!(!self.searcher.multi_line_with_matcher(&self.matcher));
    debug_assert!(self.is_line_by_line_fast());

    let mut pos = self.pos();
    while !buf[pos..].is_empty() {
        if self.has_exceeded_match_limit() {
            return Ok(None);
        }
        // Ask matcher for candidate positions
        match self.matcher.find_candidate_line(&buf[pos..]) {
            Err(err) => return Err(S::Error::error_message(err)),
            Ok(None) => return Ok(None),  // No more candidates

            // Confirmed match - regex engine is certain
            Ok(Some(LineMatchKind::Confirmed(i))) => {
                let line = lines::locate(
                    buf,
                    self.config.line_term.as_byte(),
                    Range::zero(i).offset(pos),
                );
                // Edge case: match at buffer end isn't valid
                if line.start() == buf.len() {
                    pos = buf.len();
                    continue;
                }
                return Ok(Some(line));
            }

            // Candidate - needs verification
            Ok(Some(LineMatchKind::Candidate(i))) => {
                let line = lines::locate(
                    buf,
                    self.config.line_term.as_byte(),
                    Range::zero(i).offset(pos),
                );
                // Verify the candidate is a real match
                if self.is_match(&buf[line])? {
                    return Ok(Some(line));
                }
                // Not a match, advance past this line
                pos = line.end();
            }
        }
    }
    Ok(None)
}
```

The `lines::locate` function finds the full line containing a given position, handling the translation from match position to line boundaries.

```
fn is_match(&self, line: &[u8]) -> Result<bool, S::Error> {
    // Strip terminator to prevent regexes like (?m)^$ from
    // matching the empty position after the line terminator
    let line = lines::without_terminator(line, self.config.line_term);
    self.matcher.is_match(line).map_err(S::Error::error_message)
}
```

---

## Section 4: Inverted Match Handling

```rust
#[inline(always)]
fn match_by_line_fast_invert(
    &mut self,
    buf: &[u8],
) -> Result<bool, S::Error> {
    assert!(self.config.invert_match);

    // Find the next matching line (which we want to exclude)
    let invert_match = match self.find_by_line_fast(buf)? {
        // No match found - everything from pos to end is non-matching
        None => {
            let range = Range::new(self.pos(), buf.len());
            self.set_pos(range.end());
            range
        }
        // Match found - non-matching region is from pos to match start
        Some(line) => {
            let range = Range::new(self.pos(), line.start());
            self.set_pos(line.end());  // Skip past the matching line
            range
        }
    };

    // Empty region means we hit a match immediately
    if invert_match.is_empty() {
        return Ok(true);
    }

    self.has_matched = true;

    // Handle context around the non-matching region
    if !self.after_context_by_line(buf, invert_match.start())? {
        return Ok(false);
    }
    if !self.before_context_by_line(buf, invert_match.start())? {
        return Ok(false);
    }

    // Iterate through each non-matching line
    let mut stepper = LineStep::new(
        self.config.line_term.as_byte(),
        invert_match.start(),
        invert_match.end(),
    );
    while let Some(line) = stepper.next_match(buf) {
        self.increment_count();  // Each line counts as a "match"
        if !self.sink_matched(buf, &line)? {
            return Ok(false);
        }
        // Check limit after each line
        if self.has_exceeded_match_limit() {
            return Ok(false);
        }
    }
```

```
    Ok(true)}
```

The inverted match region boundaries are computed by finding where actual matches occur, then treating everything between as the result.

---

## Section 5: The Slow Path for Complex Scenarios

```
fn match_by_line_slow(&mut self, buf: &[u8]) -> Result<bool, S::Error> {
    debug_assert!(!self.searcher.multi_line_with_matcher(&self.matcher));

    let range = Range::new(self.pos(), buf.len());
    let mut stepper = LineStep::new(
        self.config.line_term.as_byte(),
        range.start(),
        range.end(),
    );

    while let Some(line) = stepper.next_match(buf) {
        // Early exit if we've hit limits and have no more context to emit
        if self.has_exceeded_match_limit()
            && !self.config.passthru
            && self.after_context_left == 0
        {
            return Ok(false);
        }

        // Check for match (with line terminator stripped)
        let matched = {
            let slice = lines::without_terminator(
                &buf[line],
                self.config.line_term,
            );
            self.shortest_match(slice)?.is_some()
        };
        self.set_pos(line.end());

        // XOR with invert_match to get actual success
        let success = matched != self.config.invert_match;

        if success {
            // Case 1: Line matches (or doesn't match in invert mode)
            self.has_matched = true;
            self.increment_count();
            if !self.before_context_by_line(buf, line.start())? {
                return Ok(false);
            }
            if !self.sink_matched(buf, &line)? {
                return Ok(false);
            }
        } else if self.after_context_left >= 1 {
            // Case 2: Emitting after-context from previous match
            if !self.sink_after_context(buf, &line)? {
                return Ok(false);
            }
        } else if self.config.passthru {
            // Case 3: Passthrough mode - emit as "other" context
            if !self.sink_other_context(buf, &line)? {
                return Ok(false);
            }
        }
```

```
        // Case 4: Stop on first non-match after a match        if
self.config.stop_on_nonmatch && !success && self.has_matched {            return
Ok(false);        }    }    Ok(true)}
```

The `shortest_match` method is an optimization—we only need to know if a match exists, not its full extent.

---

## Quick Reference

### FastMatchResult Enum

| Variant | Meaning |
|---|---|
| `Continue` | Keep processing, buffer consumed |
| `Stop` | Stop searching entirely |
| `SwitchToSlow` | Fall back to slow path |

### Key State Fields

| Field | Purpose |
|---|---|
| `pos` | Current position in buffer |
| `absolute_byte_offset` | Position in file (survives buffer rolls) |
| `last_line_counted` | Checkpoint for incremental line counting |
| `after_context_left` | Remaining context lines to emit |
| `count` | Total matches (for `--max-count`) |

### LineMatchKind Variants

```
enum LineMatchKind {
    Confirmed(usize),  // Definite match at offset
    Candidate(usize),  // Possible match, needs verification
}
```

## Fast Path Requirements

The fast path is enabled when: 1. `passthru` mode is disabled 2. `stop_on_nonmatch` hasn't triggered 3. Matcher advertises line terminator support OR non-matching bytes include the terminator 4. Line terminator is not NUL (`\x00`)

# ripgrep crates/searcher/src/searcher/mmap.rs: Code Companion

Reference code for the Memory Mapping lecture. Sections correspond to the lecture document.

---

## Section 1: The Memory Mapping Trade-Off

```rust
use std::{fs::File, path::Path};

use memmap::Mmap;

/// Controls the strategy used for determining when to use memory maps.
///
/// If a searcher is called in circumstances where it is possible to use memory
/// maps, and memory maps are enabled, then it will attempt to do so if it
/// believes it will make the search faster.
///
/// By default, memory maps are disabled.
#[derive(Clone, Debug)]
pub struct MmapChoice(MmapChoiceImpl);
```

The `memmap` crate provides the cross-platform `Mmap` type that handles the actual memory mapping syscalls. The doc comment explicitly notes that memory maps are disabled by default—safety over performance.

---

## Section 2: The Newtype Pattern for Configuration

```rust
/// Public struct wraps private enum - the newtype pattern
#[derive(Clone, Debug)]
pub struct MmapChoice(MmapChoiceImpl);

/// Private enum - external code cannot match on or construct this directly
#[derive(Clone, Debug)]
enum MmapChoiceImpl {
    Auto,   // Memory maps enabled when advantageous
    Never,  // Memory maps completely disabled
}
```

The inner `MmapChoiceImpl` enum has no `pub` visibility modifier, making it private to this module. External code must go through `MmapChoice`'s constructor functions to create values.

---

## Section 3: Safety Contracts and Unsafe Constructors

```
impl MmapChoice {
    /// Use memory maps when they are believed to be advantageous.
    ///
    /// # Safety
    ///
    /// This constructor is not safe because there is no obvious way to
    /// encapsulate the safety of file backed memory maps on all platforms
    /// without simultaneously negating some or all of their benefits.
    ///
    /// The specific contract the caller is required to uphold isn't precise,
    /// but it basically amounts to something like, "the caller guarantees that
    /// the underlying file won't be mutated." This, of course, isn't feasible
    /// in many environments. However, command line tools may still decide to
    /// take the risk of, say, a `SIGBUS` occurring while attempting to read a
    /// memory map.
    pub unsafe fn auto() -> MmapChoice {
        MmapChoice(MmapChoiceImpl::Auto)
    }
}
```

The `unsafe` marker here is semantic rather than operational—the function body is trivial, but calling it requires acknowledging responsibility for the file's stability during mapped access.

---

## Section 4: The Safe Default

```
impl Default for MmapChoice {
    fn default() -> MmapChoice {
        MmapChoice(MmapChoiceImpl::Never)
    }
}

impl MmapChoice {
    /// Never use memory maps, no matter what. This is the default.
    pub fn never() -> MmapChoice {
        MmapChoice(MmapChoiceImpl::Never)
    }
}
```

Both `Default::default()` and `never()` produce the same value. The named constructor makes intent explicit in calling code.

---

## Section 5: The Open Method and Graceful Degradation

```
/// Return a memory map if memory maps are enabled and if creating a
/// memory from the given file succeeded and if memory maps are believed
/// to be advantageous for performance.
///
/// If this does attempt to open a memory map and it fails, then `None`
/// is returned and the corresponding error (along with the file path, if
/// present) is logged at the debug level.
pub(crate) fn open(
    &self,
    file: &File,
    path: Option<&Path>,
) -> Option<Mmap> {
    // First gate: is memory mapping enabled at all?
    if !self.is_enabled() {
        return None;
    }
    // Second gate: platform-specific heuristics
    if cfg!(target_os = "macos") {
        // I guess memory maps on macOS aren't great. Should re-evaluate.
        return None;
    }
    // ... actual mapping happens next
}
```

The `pub(crate)` visibility restricts this method to the searcher crate. Returns `Option<Mmap>` rather than `Result` because failure to map isn't an error—it's a signal to fall back to regular I/O.

---

## Section 6: Propagating Safety Assertions

```
pub(crate) fn open(
    &self,
    file: &File,
    path: Option<&Path>,
) -> Option<Mmap> {
    // ... earlier checks omitted ...

    // SAFETY: This is acceptable because the only way `MmapChoiceImpl` can
    // be `Auto` is if the caller invoked the `auto` constructor, which
    // is itself not safe. Thus, this is a propagation of the caller's
    // assertion that using memory maps is safe.
    match unsafe { Mmap::map(file) } {
        Ok(mmap) => Some(mmap),
        Err(err) => {
            // Log at debug level - not an error, just an optimization that
didn't work
            if let Some(path) = path {
                log::debug!(
                    "{}: failed to open memory map: {}",
                    path.display(),
                    err
                );
            } else {
                log::debug!("failed to open memory map: {}", err);
            }
            None
        }
    }
}
```

The safety comment documents the reasoning chain: reaching this code with `MmapChoiceImpl::Auto` means `unsafe { MmapChoice::auto() }` was called somewhere, so the safety assertion was already made.

## Section 7: The Enabled Check Helper

```
impl MmapChoice {
    /// Whether this strategy may employ memory maps or not.
    pub(crate) fn is_enabled(&self) -> bool {
        match self.0 {
            MmapChoiceImpl::Auto => true,
            MmapChoiceImpl::Never => false,
        }
    }
}
```

Simple pattern match on the inner enum. The `pub(crate)` visibility allows other modules in the searcher crate to check the memory map policy without accessing the private enum directly.

---

## Quick Reference

| Constructor | Safety | Returns | Use Case |
|---|---|---|---|
| `MmapChoice::auto()` | `unsafe` | `MmapChoice(Auto)` | Performance-critical, trusted environment |
| `MmapChoice::never()` | safe | `MmapChoice(Never)` | Safety-first, default behavior |
| `Default::default()` | safe | `MmapChoice(Never)` | Generic code, no explicit preference |

### Type Summary

```
// Public API type
pub struct MmapChoice(MmapChoiceImpl);

// Private implementation
enum MmapChoiceImpl { Auto, Never }

// Key method signatures
impl MmapChoice {
    pub unsafe fn auto() -> MmapChoice;
    pub fn never() -> MmapChoice;
    pub(crate) fn open(&self, file: &File, path: Option<&Path>) -> Option<Mmap>;
    pub(crate) fn is_enabled(&self) -> bool;
}
```

### Safety Flow

```
Caller decides mmap is safe
          ↓
unsafe { MmapChoice::auto() }  ← Safety assertion made here
          ↓
MmapChoice value passed around (safe)
          ↓
open() called (safe method)
          ↓
unsafe { Mmap::map(file) }     ← Justified by earlier assertion
```

# ripgrep crates/printer/src/lib.rs: Code Companion

Reference code for the Printer Architecture lecture. Sections correspond to the lecture document.

---

## Section 1: The Public API Surface

```
/*!
This crate provides featureful and fast printers that interoperate with the
[`grep-searcher`](https://docs.rs/grep-searcher)
crate.

# Brief overview

The [`Standard`] printer shows results in a human readable format...
The [`JSON`] printer shows results in a machine readable format...
The [`Summary`] printer shows *aggregate* results for a single search...

# Example

This example shows how to create a "standard" printer and execute a search.
```

use { grep_regex::RegexMatcher, grep_printer::Standard, grep_searcher::Searcher, };

// Sample text to search - note the byte string literal (&[u8]) const SHERLOCK: &'static [u8] = b"\ For the Doctor Watsons of this world, as opposed to the Sherlock Holmeses, success in the province of detective work must always ... ";

let matcher = RegexMatcher::new(r"Sherlock")?; let mut printer = Standard::new_no_color(vec![]);

// Key integration point: printer.sink() creates a Sink adapter // that the Searcher can write matches to Searcher::new().search_slice(&matcher, SHERLOCK, printer.sink(&matcher))?;

// Unwrap nested writers: Standard>> -> NoColor> -> Vec let output = String::from_utf8(printer.into_inner().into_inner())?;

```
*/
```

The doc comment serves as both documentation and a compile-time test. The example demonstrates the full integration pattern between searcher and printer crates.

## Section 2: The Re-export Strategy

```
pub use crate::{
    // Color-related types for terminal output styling
    color::{ColorError, ColorSpecs, UserColorSpec, default_color_specs},

    // Hyperlink support for modern terminals (clickable file paths)
    hyperlink::{
        HyperlinkAlias, HyperlinkConfig, HyperlinkEnvironment,
        HyperlinkFormat, HyperlinkFormatError, hyperlink_aliases,
    },

    // Simple path-only output (just prints matching file paths)
    path::{PathPrinter, PathPrinterBuilder},

    // Human-readable grep-style output
    standard::{Standard, StandardBuilder, StandardSink},

    // Match counting and statistics
    stats::Stats,

    // Aggregate results (counts, file lists)
    summary::{Summary, SummaryBuilder, SummaryKind, SummarySink},
};
```

Notice the pattern: each major type (`Standard`, `Summary`, `PathPrinter`) has a corresponding `Builder` and often a `Sink` type. Users import from the crate root, not the internal modules.

## Section 3: Conditional Compilation for Optional Features

```
// Module-level attribute ensures docs.rs shows feature requirements
#![cfg_attr(docsrs, feature(doc_cfg))]

// JSON types only compiled when serde feature is enabled
#[cfg(feature = "serde")]
pub use crate::json::{JSON, JSONBuilder, JSONSink};

// ... later in the file, the implementing modules are also gated:

#[cfg(feature = "serde")]
mod json;
#[cfg(feature = "serde")]
mod jsont;  // JSON type definitions (schemas)
```

Both the `pub use` re-exports and the `mod` declarations must be gated. This ensures the types, their implementations, and all dependencies are consistently included or excluded.

---

## Section 4: The Look-Ahead Kludge

```
// The maximum number of bytes to execute a search to account for look-ahead.
//
// This is an unfortunate kludge since PCRE2 doesn't provide a way to search
// a substring of some input while accounting for look-ahead. In theory, we
// could refactor the various 'grep' interfaces to account for it, but it would
// be a large change. So for now, we just let PCRE2 go looking a bit for a
// match without searching the entire rest of the contents.
//
// Note that this kludge is only active in multi-line mode.
const MAX_LOOK_AHEAD: usize = 128;
```

This constant demonstrates pragmatic software engineering: documenting the ideal solution (interface refactoring), explaining why the workaround exists (PCRE2 look-ahead), and noting its scope (multi-line mode only).

---

## Section 5: Module Organization and Privacy

```rust
// Makes macros from this module available throughout the crate
// without explicit imports (older Rust pattern)
#[macro_use]
mod macros;

// Public API implementations (exposed via pub use above)
mod color;
mod hyperlink;
mod path;
mod standard;
mod stats;
mod summary;

// Internal utilities (not in pub use - pure implementation details)
mod counter;
mod util;

// Feature-gated modules for JSON output
#[cfg(feature = "serde")]
mod json;      // Printer implementation
#[cfg(feature = "serde")]
mod jsont;     // Type definitions (schemas)
```

The module list shows what's exposed publicly (via `pub use`) versus what remains internal (`counter`, `util`). Module privacy is the first layer of API control.

---

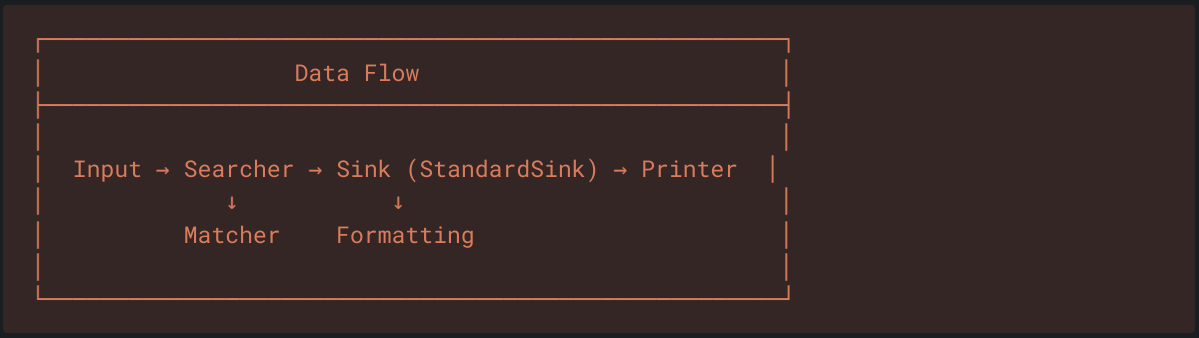
## Section 6: The Sink Pattern and Trait Integration

```rust
// From the documentation example - the key integration point:

let matcher = RegexMatcher::new(r"Sherlock")?;
let mut printer = Standard::new_no_color(vec![]);

// printer.sink(&matcher) creates a StandardSink that implements
// the Sink trait expected by Searcher::search_slice
Searcher::new().search_slice(
    &matcher,           // Used for matching
    SHERLOCK,           // Input bytes to search
    printer.sink(&matcher)  // Sink adapter for output
)?;
```

The `sink(&matcher)` method creates a bridge between the printer and searcher crates. The `Searcher` doesn't know about formatting—it just writes matches to anything implementing `Sink`.




## Section 7: The Builder Pattern Ecosystem Continued

```
// Types re-exported from lib.rs follow consistent naming:

// Standard printer ecosystem
Standard         // The printer itself
StandardBuilder  // Configures and builds Standard
StandardSink     // Implements Sink trait for Searcher integration

// Summary printer ecosystem
Summary          // The printer
SummaryBuilder   // Configuration
SummaryKind      // Enum for output variants
SummarySink      // Sink implementation

// JSON printer ecosystem (feature-gated)
JSON             // The printer
JSONBuilder      // Configuration
JSONSink         // Sink implementation

// Path printer ecosystem
PathPrinter      // Simple path-only output
PathPrinterBuilder  // Configuration
```

Every major printer type follows the same pattern: a main type, a builder for configuration, and (for search integration) a sink adapter.

## Quick Reference

### Printer Types Summary

| Printer | Purpose | Output Format | Sink Type |
|---|---|---|---|
| `Standard` | Human-readable results | grep-style lines | `StandardSink` |
| `Summary` | Aggregate statistics | counts/file lists | `SummarySink` |
| `JSON` | Machine-readable | JSON Lines | `JSONSink` |
| `PathPrinter` | File paths only | one path per line | N/A |

## Feature Flags

| Feature | Enables | Dependencies |
|---|---|---|
| `serde` | `JSON`, `JSONBuilder`, `JSONSink` | serde, serde_json |

## Module Visibility

| Module | Public Types | Notes |
|---|---|---|
| `color` | `ColorError`, `ColorSpecs`, `UserColorSpec` | Terminal coloring |
| `hyperlink` | `HyperlinkConfig`, `HyperlinkFormat`, etc. | Clickable links |
| `standard` | `Standard`, `StandardBuilder`, `StandardSink` | Main grep output |
| `summary` | `Summary`, `SummaryBuilder`, `SummaryKind`, `SummarySink` | Aggregate results |
| `stats` | `Stats` | Match counting |
| `path` | `PathPrinter`, `PathPrinterBuilder` | Path-only output |
| `counter` | (none) | Internal only |
| `util` | (none) | Internal only |

## Integration Pattern

```
// 1. Create matcher
let matcher = RegexMatcher::new(pattern)?;

// 2. Create and configure printer
let mut printer = StandardBuilder::new()
    .color_specs(specs)
    .build(writer);

// 3. Create sink from printer
let sink = printer.sink(&matcher);

// 4. Execute search with sink
Searcher::new().search_slice(&matcher, input, sink)?;
```

# ripgrep crates/printer/src/color.rs: Code Companion

Reference code for the Terminal Colors lecture. Sections correspond to the lecture document.

## Section 1: Platform-Aware Default Colors

```rust
/// Returns a default set of color specifications.
///
/// This may change over time, but the color choices are meant to be fairly
/// conservative that work across terminal themes.
pub fn default_color_specs() -> Vec<UserColorSpec> {
    vec![
        // Conditional compilation: only one of these is included in the binary
        #[cfg(unix)]
        "path:fg:magenta".parse().unwrap(),
        #[cfg(windows)]
        "path:fg:cyan".parse().unwrap(),

        // These apply on all platforms
        "line:fg:green".parse().unwrap(),

        // Match gets two specifications: color AND style
        // Later specs don't replace earlier ones—they merge
        "match:fg:red".parse().unwrap(),
        "match:style:bold".parse().unwrap(),
    ]
}
```

The `#[cfg(...)]` attributes are evaluated at compile time, not runtime. The `.parse().unwrap()` calls are safe here because these are hardcoded valid specifications—a panic would indicate a bug in the defaults themselves.

## Section 2: A Domain-Specific Error Type

```rust
/// An error that can occur when parsing color specifications.
#[derive(Clone, Debug, Eq, PartialEq)]
pub enum ColorError {
    /// This occurs when an unrecognized output type is used.
    UnrecognizedOutType(String),      // e.g., "pth" instead of "path"
    /// This occurs when an unrecognized spec type is used.
    UnrecognizedSpecType(String),     // e.g., "color" instead of "fg"
    /// This occurs when an unrecognized color name is used.
    UnrecognizedColor(String, String), // Carries both the invalid input and the
message
    /// This occurs when an unrecognized style attribute is used.
    UnrecognizedStyle(String),        // e.g., "bolder" instead of "bold"
    /// This occurs when the format of a color specification is invalid.
    InvalidFormat(String),            // Wrong number of colons, missing parts
}

impl std::fmt::Display for ColorError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match *self {
            ColorError::UnrecognizedOutType(ref name) => write!(
                f,
                "unrecognized output type '{}'. Choose from: \
                 path, line, column, match, highlight.",
                name,
            ),
            ColorError::UnrecognizedSpecType(ref name) => write!(
                f,
                "unrecognized spec type '{}'. Choose from: \
                 fg, bg, style, none.",
                name,
            ),
            // Each message includes the invalid input AND valid alternatives
            ColorError::UnrecognizedStyle(ref name) => write!(
                f,
                "unrecognized style attribute '{}'. Choose from: \
                 nobold, bold, nointense, intense, nounderline, \
                 underline, noitalic, italic.",
                name,
            ),
            // ... other variants
        }
    }
}
```

Each variant stores the problematic input string, enabling actionable error messages. The `Display` implementation constructs complete sentences that guide users toward valid input.

---

## Section 3: The Two-Layer Specification Model

```rust
/// A merged set of color specifications.
///
/// This set represents the various color types supported by the printers.
#[derive(Clone, Debug, Default, Eq, PartialEq)]
pub struct ColorSpecs {
    path: ColorSpec,       // Ready-to-use ColorSpec from termcolor
    line: ColorSpec,
    column: ColorSpec,
    matched: ColorSpec,
    highlight: ColorSpec,
}

/// A single color specification provided by the user.
///
/// ## Format
/// The format is a triple: `{type}:{attribute}:{value}`
#[derive(Clone, Debug, Eq, PartialEq)]
pub struct UserColorSpec {
    ty: OutType,           // WHERE the color applies (path, line, etc.)
    value: SpecValue,      // WHAT to do (set fg, set bg, apply style, clear)
}
```

`UserColorSpec` preserves user intent as structured data. `ColorSpecs` is the merged result—the printer asks "what color for paths?" and gets a direct answer without understanding the merge logic.

---

## Section 4: Internal Type Vocabulary

```
/// The set of configurable portions of ripgrep's output.
#[derive(Clone, Debug, Eq, PartialEq)]
enum OutType {
    Path,       // File paths in output
    Line,       // Line numbers
    Column,     // Column numbers
    Match,      // The matched text itself
    Highlight,  // Entire lines containing matches
}

/// The specification type (what kind of modification).
#[derive(Clone, Debug, Eq, PartialEq)]
enum SpecType {
    Fg,     // Foreground color
    Bg,     // Background color
    Style,  // Text style (bold, italic, etc.)
    None,   // Clear all styling
}

/// The actual value given by the specification.
#[derive(Clone, Debug, Eq, PartialEq)]
enum SpecValue {
    None,           // Instruction to clear
    Fg(Color),      // A foreground color to set
    Bg(Color),      // A background color to set
    Style(Style),   // A style modification
}

/// The set of available styles for use in the terminal.
#[derive(Clone, Debug, Eq, PartialEq)]
enum Style {
    Bold,
    NoBold,         // Negations allow unsetting styles
    Intense,
    NoIntense,
    Underline,
    NoUnderline,
    Italic,
    NoItalic,
}
```

`SpecType` is used during parsing, while `SpecValue` represents the parsed result. The `Style` enum includes both positive and negative variants, enabling users to override defaults.

## Section 5: The Merge Strategy

```rust
impl ColorSpecs {
    /// Create color specifications from a list of user supplied specifications.
    pub fn new(specs: &[UserColorSpec]) -> ColorSpecs {
        let mut merged = ColorSpecs::default();  // Start empty
        for spec in specs {
            // Route each spec to the appropriate field
            match spec.ty {
                OutType::Path => spec.merge_into(&mut merged.path),
                OutType::Line => spec.merge_into(&mut merged.line),
                OutType::Column => spec.merge_into(&mut merged.column),
                OutType::Match => spec.merge_into(&mut merged.matched),
                OutType::Highlight => spec.merge_into(&mut merged.highlight),
            }
        }
        merged
    }

    /// Create defaults WITH color (distinct from Default which has NO color).
    pub fn default_with_color() -> ColorSpecs {
        ColorSpecs::new(&default_color_specs())
    }
}

impl SpecValue {
    /// Merge this spec value into the given color specification.
    fn merge_into(&self, cspec: &mut ColorSpec) {
        match *self {
            // None clears EVERYTHING-the escape hatch
            SpecValue::None => cspec.clear(),
            // Colors are additive-set just the fg or bg
            SpecValue::Fg(ref color) => {
                cspec.set_fg(Some(color.clone()));
            }
            SpecValue::Bg(ref color) => {
                cspec.set_bg(Some(color.clone()));
            }
            // Styles are also additive
            SpecValue::Style(ref style) => match *style {
                Style::Bold => { cspec.set_bold(true); }
                Style::NoBold => { cspec.set_bold(false); }
                // ... other styles
            },
        }
    }
}
```

The key insight: `merge_into` modifies rather than replaces. Specifying `match:fg:red` then `match:style:bold` results in red bold text. Use `match:none` first to clear and start fresh.

# Section 6: Parsing with FromStr

```rust
impl std::str::FromStr for UserColorSpec {
    type Err = ColorError;

    fn from_str(s: &str) -> Result<UserColorSpec, ColorError> {
        let pieces: Vec<&str> = s.split(':').collect();

        // Must have 2 or 3 parts: "type:none" or "type:attr:value"
        if pieces.len() <= 1 || pieces.len() > 3 {
            return Err(ColorError::InvalidFormat(s.to_string()));
        }

        // Each piece has its own FromStr—errors are specific
        let otype: OutType = pieces[0].parse()?;

        match pieces[1].parse()? {
            SpecType::None => {
                // "path:none" — no third component needed
                Ok(UserColorSpec { ty: otype, value: SpecValue::None })
            }
            SpecType::Style => {
                if pieces.len() < 3 {
                    return Err(ColorError::InvalidFormat(s.to_string()));
                }
                let style: Style = pieces[2].parse()?;
                Ok(UserColorSpec { ty: otype, value: SpecValue::Style(style) })
            }
            SpecType::Fg => {
                if pieces.len() < 3 {
                    return Err(ColorError::InvalidFormat(s.to_string()));
                }
                // Color parsing delegated to termcolor, errors wrapped
                let color: Color =
                    pieces[2].parse().map_err(ColorError::from_parse_error)?;
                Ok(UserColorSpec { ty: otype, value: SpecValue::Fg(color) })
            }
            // Bg follows same pattern as Fg...
        }
    }
}

impl std::str::FromStr for OutType {
    type Err = ColorError;

    fn from_str(s: &str) -> Result<OutType, ColorError> {
        // Case-insensitive: "PATH", "Path", "path" all work
        match &*s.to_lowercase() {
            "path" => Ok(OutType::Path),
            "line" => Ok(OutType::Line),
            "column" => Ok(OutType::Column),
            "match" => Ok(OutType::Match),
            "highlight" => Ok(OutType::Highlight),
            _ => Err(ColorError::UnrecognizedOutType(s.to_string())),
        }
    }
```

```
}
```

The `?` operator propagates specific errors up the chain. The `&*s.to_lowercase()` pattern converts `String` to `&str` for matching while ensuring case-insensitive comparison.

## Section 7: Applying Specifications to Terminal State

```
impl SpecValue {
    fn merge_into(&self, cspec: &mut ColorSpec) {
        match *self {
            SpecValue::None => cspec.clear(),

            SpecValue::Fg(ref color) => {
                // set_fg takes Option<Color>—Some sets, None would clear
                cspec.set_fg(Some(color.clone()));
            }
            SpecValue::Bg(ref color) => {
                cspec.set_bg(Some(color.clone()));
            }

            SpecValue::Style(ref style) => match *style {
                // Each style is a boolean toggle
                Style::Bold => { cspec.set_bold(true); }
                Style::NoBold => { cspec.set_bold(false); }
                Style::Intense => { cspec.set_intense(true); }
                Style::NoIntense => { cspec.set_intense(false); }
                Style::Underline => { cspec.set_underline(true); }
                Style::NoUnderline => { cspec.set_underline(false); }
                Style::Italic => { cspec.set_italic(true); }
                Style::NoItalic => { cspec.set_italic(false); }
            },
        }
    }
}
```

The `termcolor::ColorSpec` type accumulates settings. Each call modifies one aspect, leaving others unchanged—this is what enables the additive merge behavior.

## Quick Reference

### Color Specification Format

```
{type}:{attribute}:{value}
```

| Component | Valid Values |
|---|---|
| `{type}` | `path`, `line`, `column`, `match`, `highlight` |
| `{attribute}` | `fg`, `bg`, `style`, `none` |
| `{value}` (colors) | `black`, `blue`, `green`, `red`, `cyan`, `magenta`, `yellow`, `white`, or `N` (0-255), or `R,G,B` |
| `{value}` (styles) | `bold`, `nobold`, `intense`, `nointense`, `underline`, `nounderline`, `italic`, `noitalic` |

## Type Hierarchy

```
User Input ("path:fg:magenta")
       ↓ parse via FromStr
UserColorSpec { ty: OutType, value: SpecValue }
       ↓ merge via ColorSpecs::new()
ColorSpecs { path: ColorSpec, line: ColorSpec, ... }
       ↓ access via .path(), .matched(), etc.
termcolor::ColorSpec (used by printer)
```

## Key Methods

| Method | Purpose |
|---|---|
| `ColorSpecs::new(&[UserColorSpec])` | Merge multiple specs into final config |
| `ColorSpecs::default_with_color()` | Get platform-appropriate defaults |
| `UserColorSpec::to_color_spec()` | Convert single spec to termcolor type |
| `SpecValue::merge_into(&mut ColorSpec)` | Apply one spec additively |

# ripgrep crates/printer/src/util.rs: Code Companion

Reference code for the Printer Utilities lecture. Sections correspond to the lecture document.

---

## Section 1: The Replacer Pattern for Amortized Allocation

```
/// A type for handling replacements while amortizing allocation.
pub(crate) struct Replacer<M: Matcher> {
    // Option allows lazy initialization - None until first use
    space: Option<Space<M>>,
}

struct Space<M: Matcher> {
    /// The place to store capture locations.
    caps: M::Captures,  // Generic over matcher's capture type
    /// The place to write a replacement to.
    dst: Vec<u8>,
    /// The place to store match offsets in terms of `dst`.
    matches: Vec<Match>,
}

impl<M: Matcher> Replacer<M> {
    /// Create a new replacer for use with a particular matcher.
    ///
    /// This constructor does not allocate. Instead, space for dealing with
    /// replacements is allocated lazily only when needed.
    pub(crate) fn new() -> Replacer<M> {
        Replacer { space: None }  // Zero allocation at construction
    }

    /// Allocate space for replacements when used with the given matcher and
    /// return a mutable reference to that space.
    fn allocate(&mut self, matcher: &M) -> io::Result<&mut Space<M>> {
        if self.space.is_none() {
            // First allocation happens here - captures come from matcher
            let caps =
                matcher.new_captures().map_err(io::Error::error_message)?;
            self.space = Some(Space { caps, dst: vec![], matches: vec![] });
        }
        Ok(self.space.as_mut().unwrap())
    }

    /// Clear space used for performing a replacement.
    pub(crate) fn clear(&mut self) {
        if let Some(ref mut space) = self.space {
            // Clear contents but retain allocated capacity
            space.dst.clear();
            space.matches.clear();
        }
    }
}
```

The `M::Captures` associated type allows each matcher implementation to define its own capture storage format, maintaining type safety without runtime dispatch.

## Section 2: Multi-Line Mode and the Look-Ahead Problem

```
pub(crate) fn replace_all<'a>(
    &'a mut self,
    searcher: &Searcher,
    matcher: &M,
    mut haystack: &[u8],
    range: std::ops::Range<usize>,
    replacement: &[u8],
) -> io::Result<()> {
    // Detect if we're in multi-line mode to handle look-ahead correctly
    let is_multi_line = searcher.multi_line_with_matcher(&matcher);

    // Get the line_terminator that was removed (if any) so we can add it back
    let line_terminator = if is_multi_line {
        // In multi-line mode: extend buffer for look-ahead
        if haystack[range.end..].len() >= MAX_LOOK_AHEAD {
            haystack = &haystack[..range.end + MAX_LOOK_AHEAD];
        }
        &[]  // No terminator removed
    } else {
        // In single-line mode: remove terminator to prevent false negatives
        let mut m = Match::new(0, range.end);
        let line_terminator =
            trim_line_terminator(searcher, haystack, &mut m);
        haystack = &haystack[..m.end()];
        line_terminator  // Save to restore later
    };

    // ... replacement logic, then restore line_terminator at the end
}
```

```rust
/// Given a buf and some bounds, if there is a line terminator at the end of
/// the given bounds in buf, then the bounds are trimmed to remove the line
/// terminator, returning the slice of the removed line terminator (if any).
pub(crate) fn trim_line_terminator<'b>(
    searcher: &Searcher,
    buf: &'b [u8],
    line: &mut Match,
) -> &'b [u8] {
    let lineterm = searcher.line_terminator();
    if lineterm.is_suffix(&buf[*line]) {
        let mut end = line.end() - 1;
        // Handle CRLF as a two-byte terminator
        if lineterm.is_crlf() && end > 0 && buf.get(end - 1) == Some(&b'\r') {
            end -= 1;
        }
        let orig_end = line.end();
        *line = line.with_end(end);
        &buf[end..orig_end]  // Return the trimmed terminator
    } else {
        &[]
    }
}
```

The `MAX_LOOK_AHEAD` constant (defined elsewhere) caps how much extra context is provided, preventing unbounded buffer growth while still supporting most practical look-ahead patterns.

---

## Section 3: The Sunk Abstraction for Unified Match Handling

```rust
/// A simple layer of abstraction over either a match or a contextual line
/// reported by the searcher.
#[derive(Debug)]
pub(crate) struct Sunk<'a> {
    bytes: &'a [u8],
    absolute_byte_offset: u64,
    line_number: Option<u64>,
    context_kind: Option<&'a SinkContextKind>,  // None for matches, Some for
context
    matches: &'a [Match],           // Positions in (possibly replaced) output
    original_matches: &'a [Match],  // Positions in original text
}

impl<'a> Sunk<'a> {
    #[inline]
    pub(crate) fn from_sink_match(
        sunk: &'a SinkMatch<'a>,
        original_matches: &'a [Match],
        replacement: Option<(&'a [u8], &'a [Match])>,
    ) -> Sunk<'a> {
        // Use replacement data if present, otherwise original
        let (bytes, matches) =
            replacement.unwrap_or_else(|| (sunk.bytes(), original_matches));
        Sunk {
            bytes,
            absolute_byte_offset: sunk.absolute_byte_offset(),
            line_number: sunk.line_number(),
            context_kind: None,  // This is a match, not context
            matches,
            original_matches,
        }
    }

    #[inline]
    pub(crate) fn from_sink_context(
        sunk: &'a SinkContext<'a>,
        original_matches: &'a [Match],
        replacement: Option<(&'a [u8], &'a [Match])>,
    ) -> Sunk<'a> {
        let (bytes, matches) =
            replacement.unwrap_or_else(|| (sunk.bytes(), original_matches));
        Sunk {
            bytes,
            absolute_byte_offset: sunk.absolute_byte_offset(),
            line_number: sunk.line_number(),
            context_kind: Some(sunk.kind()),  // Before, After, or Passthrough
            matches,
            original_matches,
        }
    }

    /// Iterate over lines using the given terminator byte
    #[inline]
    pub(crate) fn lines(&self, line_term: u8) -> LineIter<'a> {
```

```
        LineIter::new(line_term, self.bytes())    }}
```

The `'a` lifetime ties all borrowed data to the same scope, ensuring the `Sunk` value cannot outlive the search results it references.

---

## Section 4: Cross-Platform Path Handling

```rust
#[derive(Clone, Debug)]
pub(crate) struct PrinterPath<'a> {
    // Only stored on Windows - Unix can reconstruct from bytes
    #[cfg(not(unix))]
    path: &'a Path,
    bytes: Cow<'a, [u8]>,  // Borrowed when possible, owned after transforms
    hyperlink: OnceCell<Option<HyperlinkPath>>,  // Lazy, computed once
}

impl<'a> PrinterPath<'a> {
    pub(crate) fn new(path: &'a Path) -> PrinterPath<'a> {
        PrinterPath {
            #[cfg(not(unix))]
            path,
            // Zero-cost on Unix, UTF-8 check on Windows
            bytes: Vec::from_path_lossy(path),
            hyperlink: OnceCell::new(),
        }
    }

    pub(crate) fn with_separator(mut self, sep: Option<u8>) -> PrinterPath<'a> {
        /// Replace path separators in place
        fn replace_separator(bytes: &[u8], sep: u8) -> Vec<u8> {
            let mut bytes = bytes.to_vec();
            for b in bytes.iter_mut() {
                // Windows uses both / and \, Unix only /
                if *b == b'/' || (cfg!(windows) && *b == b'\\') {
                    *b = sep;
                }
            }
            bytes
        }
        let Some(sep) = sep else { return self };
        // Cow becomes Owned after transformation
        self.bytes = Cow::Owned(replace_separator(self.as_bytes(), sep));
        self
    }

    /// Return this path as a hyperlink (computed lazily)
    pub(crate) fn as_hyperlink(&self) -> Option<&HyperlinkPath> {
        self.hyperlink
            .get_or_init(|| HyperlinkPath::from_path(self.as_path()))
            .as_ref()
    }

    /// Platform-specific path reconstruction
    pub(crate) fn as_path(&self) -> &Path {
        #[cfg(unix)]
        fn imp<'p>(p: &'p PrinterPath<'_>) -> &'p Path {
            use std::{ffi::OsStr, os::unix::ffi::OsStrExt};
            // Zero-cost reconstruction from bytes on Unix
            Path::new(OsStr::from_bytes(p.as_bytes()))
        }
        #[cfg(not(unix))]
```

```
        fn imp<'p>(p: &'p PrinterPath<'_>) -> &'p Path {            p.path  //
 Use stored reference on Windows        }        imp(self)    }}
```

The `Cow<'a, [u8]>` type is ideal here: it borrows when no transformation is needed and owns when the separator is replaced, avoiding unnecessary allocation in the common case.

---

## Section 5: Performance-Optimized Number Formatting

```rust
/// A simple formatter for converting `u64` values to ASCII byte strings.
///
/// This avoids going through the formatting machinery which seems to
/// substantially slow things down.
#[derive(Debug)]
pub(crate) struct DecimalFormatter {
    buf: [u8; Self::MAX_U64_LEN],  // Fixed-size array on stack
    start: usize,                  // Index where digits begin
}

impl DecimalFormatter {
    /// Discovered via `u64::MAX.to_string().len()`.
    const MAX_U64_LEN: usize = 20;

    pub(crate) fn new(mut n: u64) -> DecimalFormatter {
        let mut buf = [0; Self::MAX_U64_LEN];
        let mut i = buf.len();  // Start from the end
        loop {
            i -= 1;
            // Extract least significant digit
            let digit = u8::try_from(n % 10).unwrap();
            n /= 10;
            buf[i] = b'0' + digit;  // Convert to ASCII
            if n == 0 {
                break;
            }
        }
        DecimalFormatter { buf, start: i }
    }

    /// Return the decimal formatted as an ASCII byte string.
    pub(crate) fn as_bytes(&self) -> &[u8] {
        &self.buf[self.start..]  // Slice from first digit to end
    }
}

#[cfg(test)]
mod tests {
    #[test]
    fn custom_decimal_format() {
        let fmt = |n: u64| {
            let bytes = DecimalFormatter::new(n).as_bytes().to_vec();
            String::from_utf8(bytes).unwrap()
        };
        let std = |n: u64| n.to_string();

        let ints = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 100, 123, u64::MAX];
        for n in ints {
            assert_eq!(std(n), fmt(n));  // Verify against std lib
        }
    }
}
```

Writing digits from right-to-left avoids the need to reverse the buffer afterward. The `start` index tracks where valid digits begin in the fixed-size buffer.

---

# Quick Reference

## Key Types

| Type | Purpose |
|---|---|
| `Replacer<M>` | Amortized-allocation regex replacement |
| `Sunk<'a>` | Unified match/context abstraction |
| `PrinterPath<'a>` | Cross-platform path with lazy hyperlinks |
| `DecimalFormatter` | Zero-allocation number formatting |
| `NiceDuration` | Human-readable duration display |

## Conditional Compilation Patterns

```
// Store extra data only on non-Unix platforms
#[cfg(not(unix))]
path: &'a Path,

// Platform-specific implementations
#[cfg(unix)]
fn imp(...) { /* Unix version */ }
#[cfg(not(unix))]
fn imp(...) { /* Windows version */ }

// Conditional separator handling
if *b == b'/' || (cfg!(windows) && *b == b'\\') { ... }
```

## Memory Optimization Patterns

```
// Lazy initialization with Option
space: Option<Space<M>>

// Lazy computation with OnceCell
hyperlink: OnceCell<Option<HyperlinkPath>>

// Copy-on-write with Cow
bytes: Cow<'a, [u8]>

// Stack-allocated fixed buffer
buf: [u8; Self::MAX_U64_LEN]
```

## Important Constants

```
const MAX_LOOK_AHEAD: usize;     // Defined in crate root, limits regex look-ahead
const MAX_U64_LEN: usize = 20;   // Maximum digits in u64::MAX
```

# ripgrep crates/printer/src/json.rs: Code Companion

Reference code for the JSON Output lecture. Sections correspond to the lecture document.

---

## Section 1: The Configuration and Builder Pattern

```
/// The configuration for the JSON printer.
///
/// This is manipulated by the JSONBuilder and then referenced by the actual
/// implementation. Once a printer is build, the configuration is frozen and
/// cannot changed.
#[derive(Debug, Clone)]
struct Config {
    pretty: bool,                          // Emit pretty-printed JSON (multi-line)
    always_begin_end: bool,                // Emit begin/end even with no matches
    replacement: Arc<Option<Vec<u8>>>,  // Arc for cheap cloning, Option for "no
replacement"
}

impl Default for Config {
    fn default() -> Config {
        Config {
            pretty: false,
            always_begin_end: false,
            replacement: Arc::new(None),
        }
    }
}

/// A builder for a JSON lines printer.
#[derive(Clone, Debug)]
pub struct JSONBuilder {
    config: Config,
}

impl JSONBuilder {
    /// Return a new builder for configuring the JSON printer.
    pub fn new() -> JSONBuilder {
        JSONBuilder { config: Config::default() }
    }

    /// Create a JSON printer that writes results to the given writer.
    pub fn build<W: io::Write>(&self, wtr: W) -> JSON<W> {
        JSON {
            config: self.config.clone(),  // Clone severs builder-printer
relationship
            wtr: CounterWriter::new(wtr),
            matches: vec![],
        }
    }

    /// Print JSON in a pretty printed format.
    pub fn pretty(&mut self, yes: bool) -> &mut JSONBuilder {
        self.config.pretty = yes;
        self  // Return self for method chaining
    }

    /// When enabled, the `begin` and `end` messages are always emitted.
    pub fn always_begin_end(&mut self, yes: bool) -> &mut JSONBuilder {
        self.config.always_begin_end = yes;
```

```
        self    }    /// Set the bytes that will be used to replace each
occurrence of a match.    pub fn replacement(        &mut self,
replacement: Option<Vec<u8>>,    ) -> &mut JSONBuilder {
self.config.replacement = Arc::new(replacement);        self    }}
```

The `Arc<Option<Vec<u8>>>` pattern allows distinguishing "no replacement configured" (`None`) from "replace with empty string" (`Some(vec![])`), while the `Arc` enables cheap cloning when building printers.

---

## Section 2: The JSON Printer Structure

```
#[derive(Clone, Debug)]
pub struct JSON<W> {
    config: Config,              // Frozen configuration from builder
    wtr: CounterWriter<W>,       // Tracks bytes written for statistics
    matches: Vec<Match>,         // Reusable buffer - cleared between uses
}

impl<W: io::Write> JSON<W> {
    /// Return a JSON lines printer with a default configuration.
    pub fn new(wtr: W) -> JSON<W> {
        // Delegate to builder - builder is the canonical construction path
        JSONBuilder::new().build(wtr)
    }
}

impl<W> JSON<W> {
    /// Returns true if and only if this printer has written at least one byte.
    pub fn has_written(&self) -> bool {
        self.wtr.total_count() > 0
    }

    /// Return a mutable reference to the underlying writer.
    pub fn get_mut(&mut self) -> &mut W {
        self.wtr.get_mut()
    }

    /// Consume this printer and return back ownership of the underlying writer.
    pub fn into_inner(self) -> W {
        self.wtr.into_inner()
    }
}
```

The `matches` vector is allocated once and reused across all searches, amortizing allocation costs. The `CounterWriter` wrapper tracks output volume for statistics reporting.

# Section 3: The Sink Abstraction

```rust
/// An implementation of `Sink` associated with a matcher and an optional file
path.
///
/// Lifetime parameters:
/// * `'p` - lifetime of the file path (or `'static` if no path)
/// * `'s` - lifetime of the borrowed JSON printer
#[derive(Debug)]
pub struct JSONSink<'p, 's, M: Matcher, W> {
    matcher: M,                           // For finding match positions within
lines
    replacer: Replacer<M>,                // For optional text substitution
    json: &'s mut JSON<W>,                // Back-reference to parent printer
    path: Option<&'p Path>,               // Optional file path for this search
    start_time: Instant,                  // For elapsed time calculation
    match_count: u64,                     // Matches found in current search
    binary_byte_offset: Option<u64>,      // Where binary data was detected
    begin_printed: bool,                  // Track if begin message was emitted
    stats: Stats,                         // Accumulated statistics
}

impl<W: io::Write> JSON<W> {
    /// Return an implementation of `Sink` for the JSON printer.
    /// No file path - will never print path with matches.
    pub fn sink<'s, M: Matcher>(
        &'s mut self,
        matcher: M,
    ) -> JSONSink<'static, 's, M, W> {
        JSONSink {
            matcher,
            replacer: Replacer::new(),
            json: self,
            path: None,  // No path association
            start_time: Instant::now(),
            match_count: 0,
            binary_byte_offset: None,
            begin_printed: false,
            stats: Stats::new(),
        }
    }

    /// Return an implementation of `Sink` associated with a file path.
    pub fn sink_with_path<'p, 's, M, P>(
        &'s mut self,
        matcher: M,
        path: &'p P,
    ) -> JSONSink<'p, 's, M, W>
    where
        M: Matcher,
        P: ?Sized + AsRef<Path>,  // Accept Path, PathBuf, str, etc.
    {
        JSONSink {
            matcher,
            replacer: Replacer::new(),
            json: self,
```

```
            path: Some(path.as_ref()),  // Convert to &Path
        start_time: Instant::now(),            match_count: 0,
        binary_byte_offset: None,           begin_printed: false,           stats:
        Stats::new(),        }    }}
```

The `?Sized` bound on `P` allows accepting both `Path` (unsized) and `PathBuf` (sized) through the `AsRef<Path>` trait. The sink borrows the printer mutably, ensuring exclusive access during a search.

---

## Section 4: The Wire Format and Message Types

```rust
impl<W: io::Write> JSON<W> {
    /// Write the given message followed by a new line.
    fn write_message(
        &mut self,
        message: &jsont::Message<'_>,
    ) -> io::Result<()> {
        // Choose compact or pretty output based on configuration
        if self.config.pretty {
            json::to_writer_pretty(&mut self.wtr, message)?;
        } else {
            json::to_writer(&mut self.wtr, message)?;
        }
        // Maintain JSON Lines format invariant: one object per line
        let _ = self.wtr.write(b"\n")?;
        Ok(())
    }
}

impl<'p, 's, M: Matcher, W: io::Write> JSONSink<'p, 's, M, W> {
    /// Write the "begin" message (idempotent - only writes once).
    fn write_begin_message(&mut self) -> io::Result<()> {
        if self.begin_printed {
            return Ok(());  // Already printed, skip
        }
        let msg = jsont::Message::Begin(jsont::Begin { path: self.path });
        self.json.write_message(&msg)?;
        self.begin_printed = true;
        Ok(())
    }
}

// From the Sink trait implementation - the begin callback
fn begin(&mut self, _searcher: &Searcher) -> Result<bool, io::Error> {
    self.json.wtr.reset_count();  // Reset byte counter for this search
    self.start_time = Instant::now();
    self.match_count = 0;
    self.binary_byte_offset = None;

    // Only emit begin immediately if always_begin_end is configured
    if !self.json.config.always_begin_end {
        return Ok(true);
    }
    self.write_begin_message()?;
    Ok(true)
}
```

The message types (`Begin`, `End`, `Match`, `Context`) are defined in the `jsont` module and implement `Serialize` for JSON output. The `begin_printed` flag prevents duplicate begin messages.

## Section 5: Handling Non-UTF-8 Data

From the extensive documentation in the source file:

```
/// ## Text encoding
///
/// JSON may only be encoded in UTF-8, UTF-16 or UTF-32. For the purposes of this
/// printer, we need only worry about UTF-8. The problem here is that searching
/// is not limited to UTF-8 exclusively, which in turn implies that matches
/// may be reported that contain invalid UTF-8.
///
/// Therefore, this printer will emit valid UTF-8 encoded bytes as normal
/// JSON strings and otherwise base64 encode data that isn't valid UTF-8. To
/// communicate whether this process occurs or not, strings are keyed by the
/// name `text` where as arbitrary bytes are keyed by `bytes`.
///
/// For example, when a path is included in a message, it is formatted like so,
/// if and only if the path is valid UTF-8:
///
/// ```json
/// {
///     "path": {
///         "text": "/home/ubuntu/lib.rs"
///     }
/// }
/// ```
///
/// If instead our path was `/home/ubuntu/lib\xFF.rs`, where the `\xFF` byte
/// makes it invalid UTF-8, the path would instead be encoded like so:
///
/// ```json
/// {
///     "path": {
///         "bytes": "L2hvbWUvdWJ1bnR1L2xpYv8ucnM="
///     }
/// }
/// ```
///
/// The printer guarantees that the `text` field is used whenever the
/// underlying bytes are valid UTF-8.
```

The dual representation (`text` vs `bytes`) ensures lossless data transfer: consumers can always recover original bytes either directly from the string or via base64 decoding.

---

## Section 6: Recording Matches Within Lines

```
impl<'p, 's, M: Matcher, W: io::Write> JSONSink<'p, 's, M, W> {
    /// Execute the matcher over the given bytes and record the match
    /// locations if the current configuration demands match granularity.
    fn record_matches(
        &mut self,
        searcher: &Searcher,
        bytes: &[u8],
        range: std::ops::Range<usize>,
    ) -> io::Result<()> {
        self.json.matches.clear();  // Reuse the buffer

        let matches = &mut self.json.matches;
        // find_iter_at_in_context handles context-aware match finding
        find_iter_at_in_context(
            searcher,
            &self.matcher,
            bytes,
            range.clone(),
            |m| {
                // Adjust offsets to be relative to line start, not buffer start
                let (s, e) = (m.start() - range.start, m.end() - range.start);
                matches.push(Match::new(s, e));
                true  // Continue finding matches
            },
        )?;

        // Don't report empty matches appearing at the end of the bytes
        // (prevents confusing output from patterns like `.*`)
        if !matches.is_empty()
            && matches.last().unwrap().is_empty()
            && matches.last().unwrap().start() >= bytes.len()
        {
            matches.pop().unwrap();
        }
        Ok(())
    }
}
```

The offset adjustment (`m.start() - range.start`) normalizes match positions to be relative to the line being reported, not the entire search buffer.

## Section 7: The Match and Context Sink Callbacks

```
impl<'p, 's, M: Matcher, W: io::Write> Sink for JSONSink<'p, 's, M, W> {
    type Error = io::Error;

    fn matched(
        &mut self,
        searcher: &Searcher,
        mat: &SinkMatch<'_>,
    ) -> Result<bool, io::Error> {
        self.match_count += 1;
        self.write_begin_message()?;  // Ensure begin was printed

        // Record individual match positions within this line
        self.record_matches(
            searcher,
            mat.buffer(),
            mat.bytes_range_in_buffer(),
        )?;
        // Compute replacement text if configured
        self.replace(searcher, mat.buffer(), mat.bytes_range_in_buffer())?;

        // Update statistics
        self.stats.add_matches(self.json.matches.len() as u64);
        self.stats.add_matched_lines(mat.lines().count() as u64);

        // Build submatches using small-vector optimization
        let submatches = SubMatches::new(
            mat.bytes(),
            &self.json.matches,
            self.replacer.replacement(),
        );

        let msg = jsont::Message::Match(jsont::Match {
            path: self.path,
            lines: mat.bytes(),
            line_number: mat.line_number(),
            absolute_offset: mat.absolute_byte_offset(),
            submatches: submatches.as_slice(),
        });
        self.json.write_message(&msg)?;
        Ok(true)  // Continue searching
    }

    fn context(
        &mut self,
        searcher: &Searcher,
        ctx: &SinkContext<'_>,
    ) -> Result<bool, io::Error> {
        self.write_begin_message()?;
        self.json.matches.clear();

        // Context lines may contain matches when search is inverted
        let submatches = if searcher.invert_match() {
            self.record_matches(searcher, ctx.bytes(), 0..ctx.bytes().len())?;
            self.replace(searcher, ctx.bytes(), 0..ctx.bytes().len())?;
```

```
            SubMatches::new(                ctx.bytes(),
&self.json.matches,                self.replacer.replacement(),            )
} else {            SubMatches::empty()        };        let msg =
jsont::Message::Context(jsont::Context {            path: self.path,
lines: ctx.bytes(),            line_number: ctx.line_number(),
absolute_offset: ctx.absolute_byte_offset(),            submatches:
submatches.as_slice(),        });        self.json.write_message(&msg)?;
Ok(true)    }}
```

The `matched` callback handles the common case; `context` handles contextual lines which may have matches when using inverted search (`-v` flag).

---

## Section 8: The SubMatches Small-Vector Optimization

```rust
/// SubMatches represents a set of matches in a contiguous range of bytes.
///
/// Optimizes for the common case of exactly one match per line.
enum SubMatches<'a> {
    Empty,
    Small([jsont::SubMatch<'a>; 1]),  // Fixed array - no allocation
    Big(Vec<jsont::SubMatch<'a>>),    // Heap allocation for multiple matches
}

impl<'a> SubMatches<'a> {
    /// Create a new set of match ranges from matches and corresponding bytes.
    fn new(
        bytes: &'a [u8],
        matches: &[Match],
        replacement: Option<(&'a [u8], &'a [Match])>,
    ) -> SubMatches<'a> {
        if matches.len() == 1 {
            // Common case: single match - use fixed-size array
            let mat = matches[0];
            SubMatches::Small([jsont::SubMatch {
                m: &bytes[mat],
                replacement: replacement
                    .map(|(rbuf, rmatches)| &rbuf[rmatches[0]]),
                start: mat.start(),
                end: mat.end(),
            }])
        } else {
            // Rare case: multiple matches - allocate vector
            let mut match_ranges = vec![];
            for (i, &mat) in matches.iter().enumerate() {
                match_ranges.push(jsont::SubMatch {
                    m: &bytes[mat],
                    replacement: replacement
                        .map(|(rbuf, rmatches)| &rbuf[rmatches[i]]),
                    start: mat.start(),
                    end: mat.end(),
                });
            }
            SubMatches::Big(match_ranges)
        }
    }

    /// Create an empty set of match ranges.
    fn empty() -> SubMatches<'static> {
        SubMatches::Empty
    }

    /// Return this set of match ranges as a slice.
    fn as_slice(&self) -> &[jsont::SubMatch<'_>] {
        match *self {
            SubMatches::Empty => &[],
            SubMatches::Small(ref x) => x,
            SubMatches::Big(ref x) => x,
        }
```

```
    }}
```

This enum-based small-vector optimization avoids heap allocation for the most common case (one match per line), while gracefully handling multiple matches when necessary.

---

## Quick Reference

### Message Types

| Type | Purpose | Key Fields |
|---|---|---|
| `begin` | Search started | `path` |
| `end` | Search finished | `path`, `binary_offset`, `stats` |
| `match` | Match found | `path`, `lines`, `line_number`, `absolute_offset`, `submatches` |
| `context` | Context line | Same as `match` |

### Arbitrary Data Object

```
// Valid UTF-8
{"text": "readable string"}

// Invalid UTF-8
{"bytes": "base64encodeddata=="}
```

### Builder Configuration

| Method | Default | Purpose |
|---|---|---|
| `pretty(bool)` | `false` | Multi-line JSON output |

| | | |
|---|---|---|
| `always_begin_end(bool)` | `false` | Emit begin/end with no matches |
| `replacement(Option<Vec<u8>>)` | `None` | Text substitution pattern |

## Key Type Signatures

```rust
// Create printer
pub fn new(wtr: W) -> JSON<W>

// Create sink for searching
pub fn sink<M: Matcher>(&mut self, matcher: M) -> JSONSink<'static, '_, M, W>
pub fn sink_with_path<M, P>(&mut self, matcher: M, path: &P) -> JSONSink<'_, '_,
M, W>

// Sink trait implementation
impl<M: Matcher, W: io::Write> Sink for JSONSink<'_, '_, M, W> {
    type Error = io::Error;
    fn matched(&mut self, ...) -> Result<bool, io::Error>
    fn context(&mut self, ...) -> Result<bool, io::Error>
}
```

# ripgrep crates/printer/src/summary.rs: Code Companion

Reference code for the Summary Output lecture. Sections correspond to the lecture document.

---

## Section 1: The Six Summary Modes

```
/// The type of summary output (if any) to print.
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
pub enum SummaryKind {
    /// Show only a count of the total number of matches (counting each line
    /// at most once) found.
    Count,
    /// Show only a count of the total number of matches (counting possibly
    /// many matches on each line) found.
    CountMatches,
    /// Show only the file path if and only if a match was found.
    PathWithMatch,
    /// Show only the file path if and only if no match was found.
    PathWithoutMatch,
    /// Don't show any output and stop the search once a match is found.
    QuietWithMatch,
    /// Don't show any output and stop the search once a non-matching file
    /// is found.
    QuietWithoutMatch,
}

impl SummaryKind {
    /// Returns true if this mode requires a file path to make sense.
    fn requires_path(&self) -> bool {
        use self::SummaryKind::*;
        match *self {
            PathWithMatch | PathWithoutMatch => true,
            Count | CountMatches | QuietWithMatch | QuietWithoutMatch => false,
        }
    }

    /// Returns true if this mode needs statistics regardless of user config.
    fn requires_stats(&self) -> bool {
        use self::SummaryKind::*;
        match *self {
            CountMatches => true,  // Must count individual matches, not just
lines
            Count | PathWithMatch | PathWithoutMatch | QuietWithMatch
            | QuietWithoutMatch => false,
        }
    }

    /// Returns true if the search can stop after the first match.
    fn quit_early(&self) -> bool {
        use self::SummaryKind::*;
        match *self {
            PathWithMatch | QuietWithMatch => true,  // Only need to know "any
match exists"
            Count | CountMatches | PathWithoutMatch | QuietWithoutMatch => false,
        }
    }
}
```

These three methods form a behavior matrix that the rest of the code consults. `CountMatches` forces statistics on because it must count each match individually, not just matching lines.

---

## Section 2: Configuration Through the Builder Pattern

```rust
/// Configuration frozen after printer is built.
#[derive(Debug, Clone)]
struct Config {
    kind: SummaryKind,
    colors: ColorSpecs,
    hyperlink: HyperlinkConfig,
    stats: bool,
    path: bool,
    exclude_zero: bool,
    separator_field: Arc<Vec<u8>>,     // What goes between path and count
    separator_path: Option<u8>,        // Replace path separators (for
Windows/Cygwin)
    path_terminator: Option<u8>,       // Custom terminator after paths
}

impl Default for Config {
    fn default() -> Config {
        Config {
            kind: SummaryKind::Count,
            colors: ColorSpecs::default(),
            hyperlink: HyperlinkConfig::default(),
            stats: false,
            path: true,
            exclude_zero: true,                      // Don't show "file:0" by
default
            separator_field: Arc::new(b":".to_vec()), // Classic grep format
            separator_path: None,
            path_terminator: None,
        }
    }
}

/// Builder with consistent method chaining pattern.
#[derive(Clone, Debug)]
pub struct SummaryBuilder {
    config: Config,
}

impl SummaryBuilder {
    pub fn new() -> SummaryBuilder {
        SummaryBuilder { config: Config::default() }
    }

    /// Each setter follows the same pattern: mutate config, return &mut self.
    pub fn kind(&mut self, kind: SummaryKind) -> &mut SummaryBuilder {
        self.config.kind = kind;
        self
    }

    pub fn separator_field(&mut self, sep: Vec<u8>) -> &mut SummaryBuilder {
        self.config.separator_field = Arc::new(sep);
        self
    }
```

```
    /// Build clones the config into the final printer.    pub fn build<W:
WriteColor>(&self, wtr: W) -> Summary<W> {        Summary {            config:
self.config.clone(),            wtr: RefCell::new(CounterWriter::new(wtr)),
}    }    /// Convenience method that wraps writer in NoColor.    pub fn
build_no_color<W: io::Write>(&self, wtr: W) -> Summary<NoColor<W>> {
self.build(NoColor::new(wtr))    }}
```

The `Arc<Vec<u8>>` for `separator_field` allows cheap cloning of the config while sharing the separator bytes across all uses.

---

## Section 3: The Summary Printer and RefCell

```
/// The summary printer wraps a writer with interior mutability.
#[derive(Clone, Debug)]
pub struct Summary<W> {
    config: Config,
    wtr: RefCell<CounterWriter<W>>,  // Interior mutability for borrowing
flexibility
}

impl<W: WriteColor> Summary<W> {
    /// Create a sink without a file path association.
    pub fn sink<'s, M: Matcher>(
        &'s mut self,
        matcher: M,
    ) -> SummarySink<'static, 's, M, W> {  // 'static because no path borrowed
        let interpolator = hyperlink::Interpolator::new(&self.config.hyperlink);
        // Enable stats if user requested OR if mode requires them
        let stats = if self.config.stats || self.config.kind.requires_stats() {
            Some(Stats::new())
        } else {
            None
        };
        SummarySink {
            matcher,
            summary: self,
            interpolator,
            path: None,  // No path for this sink
            start_time: Instant::now(),
            match_count: 0,
            binary_byte_offset: None,
            stats,
        }
    }

    /// Create a sink with a file path for output.
    pub fn sink_with_path<'p, 's, M, P>(
        &'s mut self,
        matcher: M,
        path: &'p P,
    ) -> SummarySink<'p, 's, M, W>
    where
        M: Matcher,
        P: ?Sized + AsRef<Path>,
    {
        // If path display is disabled AND mode doesn't require path, skip path
processing
        if !self.config.path && !self.config.kind.requires_path() {
            return self.sink(matcher);
        }
        // ... (similar setup with path normalization)
        let ppath = PrinterPath::new(path.as_ref())
            .with_separator(self.config.separator_path);
        // ...
    }
```

```
    /// Query whether any output has been produced.    pub fn has_written(&self)
-> bool {        self.wtr.borrow().total_count() > 0    }}
```

The `CounterWriter` tracks bytes written, enabling `has_written()` to report output status without examining content.

---

## Section 4: The SummarySink State Machine

```
/// Sink implementation that receives search callbacks.
#[derive(Debug)]
pub struct SummarySink<'p, 's, M: Matcher, W> {
    matcher: M,                                 // The pattern matcher
    summary: &'s mut Summary<W>,                // Borrowed printer
    interpolator: hyperlink::Interpolator,      // For clickable terminal links
    path: Option<PrinterPath<'p>>,              // Borrowed or 'static if none
    start_time: Instant,                        // For performance stats
    match_count: u64,                           // Accumulated matches
    binary_byte_offset: Option<u64>,            // If binary detected, where
    stats: Option<Stats>,                       // Detailed stats when enabled
}

impl<'p, 's, M: Matcher, W: WriteColor> SummarySink<'p, 's, M, W> {
    /// Interprets match_count based on the output mode.
    pub fn has_match(&self) -> bool {
        match self.summary.config.kind {
            // For "without match" modes, success means zero matches
            SummaryKind::PathWithoutMatch | SummaryKind::QuietWithoutMatch => {
                self.match_count == 0
            }
            // For all other modes, success means at least one match
            _ => self.match_count > 0,
        }
    }

    /// Check if the matcher can produce multi-line matches.
    fn multi_line(&self, searcher: &Searcher) -> bool {
        searcher.multi_line_with_matcher(&self.matcher)
    }
}
```

The lifetime `'p` is `'static` when no path is provided, allowing the sink to work uniformly with or without paths.

---

## Section 5: Match Counting Subtleties

```
impl<'p, 's, M: Matcher, W: WriteColor> Sink for SummarySink<'p, 's, M, W> {
    type Error = io::Error;

    fn matched(
        &mut self,
        searcher: &Searcher,
        mat: &SinkMatch<'_>,
    ) -> Result<bool, io::Error> {
        let is_multi_line = self.multi_line(searcher);

        // Simple case: single-line mode without stats, one callback = one match
        let sink_match_count = if self.stats.is_none() && !is_multi_line {
            1
        } else {
            // Complex case: re-scan to count individual matches in the region
            let buf = mat.buffer();
            let range = mat.bytes_range_in_buffer();
            let mut count = 0;
            find_iter_at_in_context(
                searcher,
                &self.matcher,
                buf,
                range,
                |_| {
                    count += 1;
                    true  // Continue iterating
                },
            )?;
            // Guard against edge cases where find_iter misses matches
            count.max(1)
        };

        // Update counts based on search mode
        if is_multi_line {
            self.match_count += sink_match_count;
        } else {
            self.match_count += 1;  // In single-line, count lines not matches
        }

        // Update statistics if enabled
        if let Some(ref mut stats) = self.stats {
            stats.add_matches(sink_match_count);
            stats.add_matched_lines(mat.lines().count() as u64);
        } else if self.summary.config.kind.quit_early() {
            // Early termination: signal searcher to stop
            return Ok(false);
        }
        Ok(true)  // Continue searching
    }
}
```

Returning `Ok(false)` signals the searcher to stop—critical for `PathWithMatch` mode where one match suffices.

---

## Quick Reference

| Mode | Output | Needs Path | Needs Stats | Quits Early |
|---|---|---|---|---|
| `Count` | `file:N` | No | No | No |
| `CountMatches` | `file:N` | No | **Yes** | No |
| `PathWithMatch` | `file` | **Yes** | No | **Yes** |
| `PathWithoutMatch` | `file` | **Yes** | No | No |
| `QuietWithMatch` | (none) | No | No | **Yes** |
| `QuietWithoutMatch` | (none) | No | No | No |

### Key Type Signatures

```
// The Sink trait callback return value
fn matched(&mut self, ...) -> Result<bool, io::Error>
//                                    ^^^^
//                                    false = stop searching

// Lifetime parameters on SummarySink
SummarySink<'p, 's, M, W>
//          'p = path lifetime ('static if no path)
//          's = summary printer lifetime
//          M  = Matcher implementation
//          W  = WriteColor implementation
```

### Builder Method Pattern

```
pub fn option_name(&mut self, value: Type) -> &mut SummaryBuilder {
    self.config.field = value;
    self
}
```

# ripgrep crates/globset/src/lib.rs: Code Companion

Reference code for the Glob Pattern Matching lecture. Sections correspond to the lecture document.

---

## Section 1: The Builder Pattern in Context

```rust
/// GlobSetBuilder builds a group of patterns that can be used to
/// simultaneously match a file path.
#[derive(Clone, Debug)]
pub struct GlobSetBuilder {
    pats: Vec<Glob>,
}

impl GlobSetBuilder {
    /// Create a new `GlobSetBuilder`. A `GlobSetBuilder` can be used to add new
    /// patterns. Once all patterns have been added, `build` should be called
    /// to produce a [`GlobSet`], which can then be used for matching.
    pub fn new() -> GlobSetBuilder {
        GlobSetBuilder { pats: vec![] }
    }

    /// Builds a new matcher from all of the glob patterns added so far.
    ///
    /// Once a matcher is built, no new patterns can be added to it.
    pub fn build(&self) -> Result<GlobSet, Error> {
        GlobSet::new(self.pats.iter())
    }

    /// Add a new pattern to this set.
    /// Returns &mut Self for method chaining
    pub fn add(&mut self, pat: Glob) -> &mut GlobSetBuilder {
        self.pats.push(pat);
        self
    }
}

impl GlobSet {
    /// Alternative constructor that returns a GlobSetBuilder
    /// Provides flexibility at call sites
    #[inline]
    pub fn builder() -> GlobSetBuilder {
        GlobSetBuilder::new()
    }

    /// Create an empty `GlobSet`. An empty set matches nothing.
    /// Note: const fn allows use in static/const contexts
    #[inline]
    pub const fn empty() -> GlobSet {
        GlobSet { len: 0, strats: vec![] }
    }
}
```

The `add` method returns `&mut GlobSetBuilder` to enable method chaining (e.g., `builder.add(glob1).add(glob2)`). The `build` method borrows `&self`, allowing the builder to be reused if needed.

# Section 2: Error Design Philosophy

```rust
/// Represents an error that can occur when parsing a glob pattern.
#[derive(Clone, Debug, Eq, PartialEq)]
pub struct Error {
    /// The original glob provided by the caller.
    glob: Option<String>,
    /// The kind of error.
    kind: ErrorKind,
}

/// The kind of error that can occur when parsing a glob pattern.
#[derive(Clone, Debug, Eq, PartialEq)]
#[non_exhaustive]  // Allows adding new variants in future versions
pub enum ErrorKind {
    /// **DEPRECATED** - kept for backward compatibility
    InvalidRecursive,
    /// Occurs when a character class (e.g., `[abc]`) is not closed.
    UnclosedClass,
    /// Occurs when a range in a character (e.g., `[a-z]`) is invalid.
    InvalidRange(char, char),
    /// Occurs when a `}` is found without a matching `{`.
    UnopenedAlternates,
    /// Occurs when a `{` is found without a matching `}`.
    UnclosedAlternates,
    /// **DEPRECATED** - nested alternates now supported
    NestedAlternates,
    /// Occurs when an unescaped '\' is found at the end of a glob.
    DanglingEscape,
    /// An error associated with parsing or compiling a regex.
    Regex(String),
}

impl ErrorKind {
    fn description(&self) -> &str {
        match *self {
            // Note the helpful suggestion in error messages
            ErrorKind::UnopenedAlternates => {
                "unopened alternate group; missing '{' \
                (maybe escape '}' with '[}]'?)"
            }
            ErrorKind::UnclosedAlternates => {
                "unclosed alternate group; missing '}' \
                (maybe escape '{' with '[{]'?)"
            }
            // Other variants...
            ErrorKind::InvalidRange(_, _) => "invalid character range",
            ErrorKind::UnclosedClass => {
                "unclosed character class; missing ']'"
            }
            // ...
        }
    }
}

impl std::fmt::Display for Error {
```

```
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
match self.glob {            // Include the problematic pattern in the error
message            None => self.kind.fmt(f),            Some(ref glob) => {
write!(f, "error parsing glob '{}': {}", glob, self.kind)            }        }
}}
```

The `#[non_exhaustive]` attribute forces downstream code to handle unknown variants with a wildcard arm, enabling library evolution without breaking changes.

---

## Section 3: The Candidate Abstraction

```rust
/// A candidate path for matching.
///
/// All glob matching in this crate operates on `Candidate` values.
/// Constructing candidates has a very small cost associated with it, so
/// callers may find it beneficial to amortize that cost when matching a single
/// path against multiple globs or sets of globs.
#[derive(Clone)]
pub struct Candidate<'a> {
    path: Cow<'a, [u8]>,      // Full normalized path
    basename: Cow<'a, [u8]>,  // Filename without directory
    ext: Cow<'a, [u8]>,       // File extension (without dot)
}

impl<'a> Candidate<'a> {
    /// Create a new candidate for matching from the given path.
    pub fn new<P: AsRef<Path> + ?Sized>(path: &'a P) -> Candidate<'a> {
        Self::from_cow(Vec::from_path_lossy(path.as_ref()))
    }

    fn from_cow(path: Cow<'a, [u8]>) -> Candidate<'a> {
        // Normalize once, then extract derived values
        let path = normalize_path(path);
        let basename = file_name(&path).unwrap_or(Cow::Borrowed(B("")));
        let ext = file_name_ext(&basename).unwrap_or(Cow::Borrowed(B("")));
        Candidate { path, basename, ext }
    }

    // Efficient prefix/suffix extraction for Aho-Corasick matching
    fn path_prefix(&self, max: usize) -> &[u8] {
        if self.path.len() <= max { &*self.path } else { &self.path[..max] }
    }

    fn path_suffix(&self, max: usize) -> &[u8] {
        if self.path.len() <= max {
            &*self.path
        } else {
            &self.path[self.path.len() - max..]
        }
    }
}
```

`Cow<'a, [u8]>` (Copy-on-Write) allows the candidate to either borrow from the input path or own normalized data, avoiding allocation when the path is already in the correct form.

## Section 4: Strategy-Based Matching Architecture

```rust
#[derive(Clone, Debug)]
enum GlobSetMatchStrategy {
    Literal(LiteralStrategy),           // Exact path match: "src/main.rs"
    BasenameLiteral(BasenameLiteralStrategy), // Filename match: "Makefile"
    Extension(ExtensionStrategy),       // Extension match: "*.rs"
    Prefix(PrefixStrategy),             // Prefix match: "src/**"
    Suffix(SuffixStrategy),             // Suffix match: "**/*.rs"
    RequiredExtension(RequiredExtensionStrategy), // Hybrid: ext + regex
    Regex(RegexSetStrategy),            // Full regex fallback
}

// Pattern categorization during GlobSet construction
pub fn new<I, G>(globs: I) -> Result<GlobSet, Error>
where
    I: IntoIterator<Item = G>,
    G: AsRef<Glob>,
{
    // Initialize collectors for each strategy
    let mut lits = LiteralStrategy::new();
    let mut base_lits = BasenameLiteralStrategy::new();
    let mut exts = ExtensionStrategy::new();
    let mut prefixes = MultiStrategyBuilder::new();
    let mut suffixes = MultiStrategyBuilder::new();
    let mut required_exts = RequiredExtensionStrategyBuilder::new();
    let mut regexes = MultiStrategyBuilder::new();

    for (i, p) in it.enumerate() {
        let p = p.as_ref();
        // Route each pattern to its optimal strategy
        match MatchStrategy::new(p) {
            MatchStrategy::Literal(lit) => {
                lits.add(i, lit);
            }
            MatchStrategy::BasenameLiteral(lit) => {
                base_lits.add(i, lit);
            }
            MatchStrategy::Extension(ext) => {
                exts.add(i, ext);
            }
            MatchStrategy::Prefix(prefix) => {
                prefixes.add(i, prefix);
            }
            MatchStrategy::Suffix { suffix, component } => {
                if component {
                    // Also add to literals for component matching
                    lits.add(i, suffix[1..].to_string());
                }
                suffixes.add(i, suffix);
            }
            MatchStrategy::RequiredExtension(ext) => {
                required_exts.add(i, ext, p.regex().to_owned());
            }
            MatchStrategy::Regex => {
                regexes.add(i, p.regex().to_owned());
```

```
            }        }    }    // ... build strategies and return GlobSet}
```

The `MatchStrategy::new(p)` call analyzes each glob pattern and determines which optimization strategy can handle it most efficiently.

---

## Section 5: The Multi-Pattern Matching Engine

```rust
impl GlobSet {
    /// Returns true if any glob in this set matches the path given.
    /// Short-circuits on first match.
    pub fn is_match_candidate(&self, path: &Candidate<'_>) -> bool {
        if self.is_empty() {
            return false;
        }
        for strat in &self.strats {
            if strat.is_match(path) {
                return true;  // Early exit on first match
            }
        }
        false
    }

    /// Returns true if all globs in this set match the path given.
    /// Empty set returns true (vacuous truth).
    pub fn matches_all_candidate(&self, path: &Candidate<'_>) -> bool {
        for strat in &self.strats {
            if !strat.is_match(path) {
                return false;  // Early exit on first non-match
            }
        }
        true
    }

    /// Adds the sequence number of every glob pattern that matches.
    /// Results are sorted and deduplicated.
    pub fn matches_candidate_into(
        &self,
        path: &Candidate<'_>,
        into: &mut Vec<usize>,
    ) {
        into.clear();  // Always start fresh
        if self.is_empty() {
            return;
        }
        // Collect matches from all strategies
        for strat in &self.strats {
            strat.matches_into(path, into);
        }
        // Remove duplicates (pattern might match multiple strategies)
        into.sort();
        into.dedup();
    }
}
```

The `matches_into` variant lets callers provide a pre-allocated vector, enabling efficient batch processing by reusing allocations across many path matches.

## Section 6: Hash-Based Strategies

```
#[derive(Clone, Debug)]
struct LiteralStrategy(fnv::HashMap<Vec<u8>, Vec<usize>>);

impl LiteralStrategy {
    fn new() -> LiteralStrategy {
        LiteralStrategy(fnv::HashMap::default())
    }

    fn add(&mut self, global_index: usize, lit: String) {
        // Multiple patterns can map to the same literal
        self.0.entry(lit.into_bytes()).or_insert(vec![]).push(global_index);
    }

    fn is_match(&self, candidate: &Candidate<'_>) -> bool {
        // O(1) hash lookup regardless of pattern count
        self.0.contains_key(candidate.path.as_bytes())
    }

    #[inline(never)]  // Hint to avoid inlining (code size optimization)
    fn matches_into(
        &self,
        candidate: &Candidate<'_>,
        matches: &mut Vec<usize>,
    ) {
        if let Some(hits) = self.0.get(candidate.path.as_bytes()) {
            matches.extend(hits);
        }
    }
}

#[derive(Clone, Debug)]
struct ExtensionStrategy(fnv::HashMap<Vec<u8>, Vec<usize>>);

impl ExtensionStrategy {
    fn is_match(&self, candidate: &Candidate<'_>) -> bool {
        if candidate.ext.is_empty() {
            return false;  // No extension means no match
        }
        self.0.contains_key(candidate.ext.as_bytes())
    }
    // ... similar structure to LiteralStrategy
}

#[derive(Clone, Debug)]
struct BasenameLiteralStrategy(fnv::HashMap<Vec<u8>, Vec<usize>>);
// ... same pattern, keyed by basename instead of full path
```

All three strategies use FNV (Fowler-Noll-Vo) hashing, optimized for small keys like file extensions and basenames.

---

## Section 7: Prefix and Suffix Strategies

```rust
#[derive(Clone, Debug)]
struct PrefixStrategy {
    matcher: AhoCorasick,   // Multi-pattern string matching
    map: Vec<usize>,        // Pattern index to global index
    longest: usize,         // Maximum prefix length to check
}

impl PrefixStrategy {
    fn is_match(&self, candidate: &Candidate<'_>) -> bool {
        // Only check the relevant prefix of the path
        let path = candidate.path_prefix(self.longest);
        for m in self.matcher.find_overlapping_iter(path) {
            if m.start() == 0 {  // Must match from the beginning
                return true;
            }
        }
        false
    }

    fn matches_into(
        &self,
        candidate: &Candidate<'_>,
        matches: &mut Vec<usize>,
    ) {
        let path = candidate.path_prefix(self.longest);
        for m in self.matcher.find_overlapping_iter(path) {
            if m.start() == 0 {
                matches.push(self.map[m.pattern()]);
            }
        }
    }
}

#[derive(Clone, Debug)]
struct SuffixStrategy {
    matcher: AhoCorasick,
    map: Vec<usize>,
    longest: usize,
}

impl SuffixStrategy {
    fn is_match(&self, candidate: &Candidate<'_>) -> bool {
        let path = candidate.path_suffix(self.longest);
        for m in self.matcher.find_overlapping_iter(path) {
            if m.end() == path.len() {  // Must match at the end
                return true;
            }
        }
        false
    }
}
```

The `longest` field enables an optimization: only examine the portion of the path that could possibly match, reducing work for long paths.

---

## Section 8: RequiredExtension and Regex Strategies

```rust
#[derive(Clone, Debug)]
struct RequiredExtensionStrategy(fnv::HashMap<Vec<u8>, Vec<(usize, Regex)>>);

impl RequiredExtensionStrategy {
    fn is_match(&self, candidate: &Candidate<'_>) -> bool {
        if candidate.ext.is_empty() {
            return false;
        }
        // First: O(1) extension lookup narrows candidates
        match self.0.get(candidate.ext.as_bytes()) {
            None => false,
            Some(regexes) => {
                // Then: run regex only on filtered candidates
                for &(_, ref re) in regexes {
                    if re.is_match(candidate.path.as_bytes()) {
                        return true;
                    }
                }
                false
            }
        }
    }
}

#[derive(Clone, Debug)]
struct RegexSetStrategy {
    matcher: Regex,
    map: Vec<usize>,
    // Pool of PatternSets to avoid allocation per match
    patset: Arc<Pool<PatternSet, PatternSetPoolFn>>,
}

type PatternSetPoolFn =
    Box<dyn Fn() -> PatternSet + Send + Sync + UnwindSafe + RefUnwindSafe>;

impl RegexSetStrategy {
    fn matches_into(
        &self,
        candidate: &Candidate<'_>,
        matches: &mut Vec<usize>,
    ) {
        let input = regex_automata::Input::new(candidate.path.as_bytes());
        let mut patset = self.patset.get();  // Get from pool
        patset.clear();
        self.matcher.which_overlapping_matches(&input, &mut patset);
        for i in patset.iter() {
            matches.push(self.map[i]);
        }
        PoolGuard::put(patset);  // Return to pool
    }
}
```

The `RequiredExtensionStrategy` demonstrates a two-phase filter: cheap hash lookup first, expensive regex only if needed. The `RegexSetStrategy` uses object pooling to amortize allocation costs.

---

# Quick Reference

## Strategy Selection Summary

| Pattern Type | Example | Strategy | Lookup Complexity |
|---|---|---|---|
| Exact path | `src/main.rs` | Literal | O(1) hash |
| Basename only | `Makefile` | BasenameLiteral | O(1) hash |
| Extension | `*.rs` | Extension | O(1) hash |
| Prefix | `src/**` | Prefix (Aho-Corasick) | O(n) where n = prefix length |
| Suffix | `**/*.rs` | Suffix (Aho-Corasick) | O(n) where n = suffix length |
| Ext + complex | `src/**/*.rs` | RequiredExtension | O(1) + regex |
| Complex | `{foo,bar}/**` | Regex | Full regex cost |

## Key Types

```
// Primary API types
GlobSet           // Immutable, optimized matcher for multiple patterns
GlobSetBuilder    // Accumulates patterns before building
Candidate<'a>     // Pre-processed path for efficient matching

// Strategy enum (internal)
enum GlobSetMatchStrategy {
    Literal(LiteralStrategy),
    BasenameLiteral(BasenameLiteralStrategy),
    Extension(ExtensionStrategy),
    Prefix(PrefixStrategy),
    Suffix(SuffixStrategy),
    RequiredExtension(RequiredExtensionStrategy),
    Regex(RegexSetStrategy),
}
```

## Matching Method Variants

| Method | Returns | Use Case |
|---|---|---|
| `is_match(path)` | `bool` | Quick existence check |
| `is_match_candidate(candidate)` | `bool` | Amortized existence check |
| `matches(path)` | `Vec<usize>` | Get all matching pattern indices |
| `matches_candidate(candidate)` | `Vec<usize>` | Amortized match collection |
| `matches_into(path, vec)` | `()` | Reuse allocation |
| `matches_candidate_into(candidate, vec)` | `()` | Full amortization |
| `matches_all(path)` | `bool` | All patterns must match |

# ripgrep crates/globset/src/glob.rs: Code Companion

Reference code for the Glob Implementation lecture. Sections correspond to the lecture document.

---

## Section 1: The Match Strategy Optimization

```rust
/// Describes a matching strategy for a particular pattern.
///
/// This provides a way to more quickly determine whether a pattern matches
/// a particular file path in a way that scales with a large number of
/// patterns.
#[derive(Clone, Debug, Eq, PartialEq)]
pub(crate) enum MatchStrategy {
    /// Pattern matches if and only if the entire file path matches this literal
    Literal(String),
    /// Pattern matches if and only if the file path's basename matches this
literal
    BasenameLiteral(String),
    /// Pattern matches if and only if the file path's extension matches this
literal
    Extension(String),
    /// Pattern matches if and only if this prefix literal is a prefix of the
path
    Prefix(String),
    /// Pattern matches if and only if this suffix literal is a suffix of the
path
    Suffix {
        suffix: String,
        /// Whether this must start at the beginning of a path component
        component: bool,
    },
    /// Extension match is necessary but NOT sufficient - still need regex
    RequiredExtension(String),
    /// No optimization available - use full regex matching
    Regex,
}

impl MatchStrategy {
    /// Returns a matching strategy for the given pattern.
    /// Tries optimizations in order of specificity/speed.
    pub(crate) fn new(pat: &Glob) -> MatchStrategy {
        // Try each optimization in order - first match wins
        if let Some(lit) = pat.basename_literal() {
            MatchStrategy::BasenameLiteral(lit)
        } else if let Some(lit) = pat.literal() {
            MatchStrategy::Literal(lit)
        } else if let Some(ext) = pat.ext() {
            MatchStrategy::Extension(ext)
        } else if let Some(prefix) = pat.prefix() {
            MatchStrategy::Prefix(prefix)
        } else if let Some((suffix, component)) = pat.suffix() {
            MatchStrategy::Suffix { suffix, component }
        } else if let Some(ext) = pat.required_ext() {
            MatchStrategy::RequiredExtension(ext)
        } else {
            MatchStrategy::Regex
        }
    }
}
```

The ordering in `new` reflects optimization priority: basename literal is checked first because patterns like `**/Makefile` are extremely common in `.gitignore` files and can be matched with a single string comparison.

---

## Section 2: The Glob Type and Builder Pattern

```
/// Glob represents a successfully parsed shell glob pattern.
#[derive(Clone, Eq)]
pub struct Glob {
    glob: String,       // Original pattern text
    re: String,         // Generated regex string
    opts: GlobOptions,  // Parsing/matching options
    tokens: Tokens,     // Parsed token representation
}

// Only compare pattern and options - not internal representation
impl PartialEq for Glob {
    fn eq(&self, other: &Glob) -> bool {
        self.glob == other.glob && self.opts == other.opts
    }
}

impl std::hash::Hash for Glob {
    fn hash<H: std::hash::Hasher>(&self, state: &mut H) {
        self.glob.hash(state);
        self.opts.hash(state);
    }
}

// Compact debug by default, expanded with {:#?}
impl std::fmt::Debug for Glob {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        if f.alternate() {
            // {:#?} shows all internals for debugging
            f.debug_struct("Glob")
                .field("glob", &self.glob)
                .field("re", &self.re)
                .field("opts", &self.opts)
                .field("tokens", &self.tokens)
                .finish()
        } else {
            // {:?} shows just the pattern
            f.debug_tuple("Glob").field(&self.glob).finish()
        }
    }
}
```

```rust
/// A builder for a pattern - lifetime tied to input pattern string.
#[derive(Clone, Debug)]
pub struct GlobBuilder<'a> {
    glob: &'a str,       // Borrowed pattern text
    opts: GlobOptions,   // Accumulated options
}

#[derive(Clone, Copy, Debug, Eq, Hash, PartialEq)]
struct GlobOptions {
    case_insensitive: bool,
    literal_separator: bool,   // When true, * and ? won't match /
    backslash_escape: bool,    // Platform-dependent default
    empty_alternates: bool,    // Allow {,a} to match "" and "a"
    allow_unclosed_class: bool,
}

impl GlobOptions {
    fn default() -> GlobOptions {
        GlobOptions {
            case_insensitive: false,
            literal_separator: false,
            // On Unix, \ is escape; on Windows, \ is path separator
            backslash_escape: !is_separator('\\'),
            empty_alternates: false,
            allow_unclosed_class: false,
        }
    }
}
```

The `backslash_escape` default uses `is_separator('\\')` to detect the platform: on Windows where `\` is a path separator, escaping is disabled by default.

## Section 3: The Token Representation

```rust
/// Wrapper type around Vec<Token> for type clarity
#[derive(Clone, Debug, Default, Eq, PartialEq)]
struct Tokens(Vec<Token>);

// Deref allows using Tokens as a Vec<Token>
impl std::ops::Deref for Tokens {
    type Target = Vec<Token>;
    fn deref(&self) -> &Vec<Token> {
        &self.0
    }
}

impl std::ops::DerefMut for Tokens {
    fn deref_mut(&mut self) -> &mut Vec<Token> {
        &mut self.0
    }
}

#[derive(Clone, Debug, Eq, PartialEq)]
enum Token {
    /// A literal character to match exactly
    Literal(char),
    /// ? - matches any single character
    Any,
    /// * - matches zero or more characters
    ZeroOrMore,
    /// **/ at pattern start - matches any prefix path
    RecursivePrefix,
    /// /** at pattern end - matches any suffix path
    RecursiveSuffix,
    /// /**/ in middle - must match at least one /
    RecursiveZeroOrMore,
    /// Character class like [a-z] or [!0-9]
    Class { negated: bool, ranges: Vec<(char, char)> },
    /// Alternation like {foo,bar,baz}
    Alternates(Vec<Tokens>),
}
```

The three recursive variants (`RecursivePrefix`, `RecursiveSuffix`, `RecursiveZeroOrMore`) capture the semantic differences of `**` in different positions, which affects both regex generation and optimization extraction.

## Section 4: Strategy Extraction Methods

```
impl Glob {
    /// Returns pattern as literal only if it contains no wildcards.
    fn literal(&self) -> Option<String> {
        // Case insensitive requires regex machinery
        if self.opts.case_insensitive {
            return None;
        }
        let mut lit = String::new();
        for t in &*self.tokens {
            // Bail if anything other than a literal
            let Token::Literal(c) = *t else { return None };
            lit.push(c);
        }
        if lit.is_empty() { None } else { Some(lit) }
    }

    /// Returns extension for patterns like *.ext or **/*.ext
    fn ext(&self) -> Option<String> {
        if self.opts.case_insensitive {
            return None;
        }
        // Check for optional **/ prefix
        let start = match *self.tokens.get(0)? {
            Token::RecursivePrefix => 1,
            _ => 0,
        };
        // Must have * that can match path separators
        match *self.tokens.get(start)? {
            Token::ZeroOrMore => {
                // Without **, * must be able to match /
                if start == 0 && self.opts.literal_separator {
                    return None;
                }
            }
            _ => return None,
        }
        // Must have literal dot
        match *self.tokens.get(start + 1)? {
            Token::Literal('.') => {}
            _ => return None,
        }
        // Rest must be literals without . or /
        let mut lit = ".".to_string();
        for t in self.tokens[start + 2..].iter() {
            match *t {
                Token::Literal('.') | Token::Literal('/') => return None,
                Token::Literal(c) => lit.push(c),
                _ => return None,
            }
        }
        if lit.is_empty() { None } else { Some(lit) }
    }

    /// Returns tokens for basename-only matching (e.g., **/filename)
```

```
    fn basename_tokens(&self) -> Option<&[Token]> {        if
self.opts.case_insensitive {            return None;        }        let start =
match *self.tokens.get(0)? {            Token::RecursivePrefix => 1,            _
=> {                // Without ** prefix, can't assume basename-only is correct
return None;            }        };        if self.tokens[start..].is_empty() {
return None;        }        // Check remaining tokens don't escape basename
for t in self.tokens[start..].iter() {            match *t {
Token::Literal('/') => return None,                Token::Literal(_) => {} // OK
Token::Any | Token::ZeroOrMore => {                    if
!self.opts.literal_separator {                        // * or ? could match /,
escaping basename                        return None;                    }
}                Token::RecursivePrefix                | Token::RecursiveSuffix
| Token::RecursiveZeroOrMore => return None,                Token::Class { .. } |
Token::Alternates(..) => {                    // Too complex - give up on
optimization                    return None;                }            }
}        // Return slice to avoid allocation        Some(&self.tokens[start..])
}}
```

Note that `basename_tokens` returns `&[Token]` rather than allocating a new `Vec`—this is efficient since we're just providing a view into existing data.

---

## Section 5: The Parser Structure

```rust
struct Parser<'a> {
    /// The glob pattern being parsed
    glob: &'a str,
    /// Stack tracking where alternation groups begin
    alternates_stack: Vec<usize>,
    /// Active branches - tokens added to last one
    branches: Vec<Tokens>,
    /// Character iterator with peek capability
    chars: std::iter::Peekable<std::str::Chars<'a>>,
    /// Previous character (for context like detecting /**)
    prev: Option<char>,
    /// Current character
    cur: Option<char>,
    /// Tracks if we found unclosed [ (optimization to avoid quadratic time)
    found_unclosed_class: bool,
    /// Options that may influence parsing
    opts: &'a GlobOptions,
}

impl<'a> Parser<'a> {
    fn parse(&mut self) -> Result<(), Error> {
        while let Some(c) = self.bump() {
            match c {
                '?' => self.push_token(Token::Any)?,
                '*' => self.parse_star()?,
                '[' if !self.found_unclosed_class => self.parse_class()?,
                '{' => self.push_alternate()?,
                '}' => self.pop_alternate()?,
                ',' => self.parse_comma()?,
                '\\' => self.parse_backslash()?,
                c => self.push_token(Token::Literal(c))?,
            }
        }
        Ok(())
    }

    fn push_alternate(&mut self) -> Result<(), Error> {
        // Mark where this alternation starts
        self.alternates_stack.push(self.branches.len());
        // Start first branch of this alternation
        self.branches.push(Tokens::default());
        Ok(())
    }

    fn pop_alternate(&mut self) -> Result<(), Error> {
        let Some(start) = self.alternates_stack.pop() else {
            return Err(self.error(ErrorKind::UnopenedAlternates));
        };
        // Collect all branches since start into Alternates token
        let alts = Token::Alternates(self.branches.drain(start..).collect());
        self.push_token(alts)?;
        Ok(())
    }
```

```
    fn parse_comma(&mut self) -> Result<(), Error> {        // Comma is special
only inside alternation        if self.alternates_stack.is_empty() {
self.push_token(Token::Literal(','))        } else {            // Start new
branch at same nesting level            Ok(self.branches.push(Tokens::default()))
}    }}
```

The `alternates_stack` and `branches` work together: when `{` is encountered, the current branch count is pushed to the stack. When `}` is seen, all branches since that index are collected into an `Alternates` token.

---

# Quick Reference

## Token Types

| Token | Pattern | Meaning |
|---|---|---|
| `Literal(c)` | `a`, `b`, `/` | Match exact character |
| `Any` | `?` | Match any single character |
| `ZeroOrMore` | `*` | Match zero or more characters |
| `RecursivePrefix` | `**/` | Match any directory prefix |
| `RecursiveSuffix` | `/**` | Match any directory suffix |
| `RecursiveZeroOrMore` | `/**/` | Match one or more directories |
| `Class` | `[a-z]` | Match character in class |
| `Alternates` | `{a,b}` | Match any alternative |

## Match Strategy Selection

```
Pattern              → Strategy
──────────────────────────────────────────
**/Makefile        → BasenameLiteral
foo/bar            → Literal
*.rs               → Extension
build/**           → Prefix
**/test.js         → Suffix (component=true)
**/*.test.js       → RequiredExtension
**/*_test/**/*.c   → Regex
```

## Key Builder Options

| Option | Default | Effect |
|---|---|---|
| `case_insensitive` | `false` | Enable case-insensitive matching |
| `literal_separator` | `false` | Require `/` to match path separator |
| `backslash_escape` | platform | Enable `\` as escape character |
| `empty_alternates` | `false` | Allow `{,a}` to match empty string |
| `allow_unclosed_class` | `false` | Treat `[` without `]` as literal |

# Ignore Crate Overview: Code Companion

Reference code for the Ignore Crate Overview lecture. Sections correspond to the lecture document.

## Section 1: The Crate's Public Interface

```
/*!
The ignore crate provides a fast recursive directory iterator that respects
various filters such as globs, file types and `.gitignore` files. The precise
matching rules and precedence is explained in the documentation for
`WalkBuilder`.

Secondarily, this crate exposes gitignore and file type matchers for use cases
that demand more fine-grained control.
*/

// Simple usage example from the documentation
use ignore::Walk;

for result in Walk::new("./") {
    match result {
        Ok(entry) => println!("{}", entry.path().display()),
        Err(err) => println!("ERROR: {}", err),
    }
}

// Builder pattern for customization
use ignore::WalkBuilder;

for result in WalkBuilder::new("./").hidden(false).build() {
    println!("{:?}", result);
}
```

```
// Re-exporting key types at the crate root for a clean public API
pub use crate::walk::{
    DirEntry, ParallelVisitor, ParallelVisitorBuilder, Walk, WalkBuilder,
    WalkParallel, WalkState,
};
```

The `pub use` pattern allows users to write `ignore::Walk` instead of `ignore::walk::Walk`, hiding internal module organization from the public API.

---

## Section 2: Module Organization

```
// Private modules - implementation details
mod default_types;  // Built-in file type definitions
mod dir;            // Directory entry handling
mod pathutil;       // Path manipulation utilities
mod walk;           // Core walking implementation (types re-exported above)

// Public modules - stable API for fine-grained control
pub mod gitignore;  // Gitignore file parsing and matching
pub mod overrides;  // User-specified override patterns
pub mod types;      // File type matching (e.g., "rust", "python")
```

The `pub mod` vs `mod` distinction determines what users can import directly. Private modules can be refactored freely; public modules are part of the semver contract.

---

## Section 3: The Error Enum Architecture

```rust
/// Represents an error that can occur when parsing a gitignore file.
#[derive(Debug)]
pub enum Error {
    /// A collection of "soft" errors. These occur when adding an ignore
    /// file partially succeeded.
    Partial(Vec<Error>),

    /// An error associated with a specific line number.
    WithLineNumber {
        line: u64,
        err: Box<Error>,  // Boxed to avoid infinite size
    },

    /// An error associated with a particular file path.
    WithPath {
        path: PathBuf,
        err: Box<Error>,
    },

    /// An error associated with a particular directory depth when recursively
    /// walking a directory.
    WithDepth {
        depth: usize,
        err: Box<Error>,
    },

    /// An error that occurs when a file loop is detected when traversing
    /// symbolic links.
    Loop {
        ancestor: PathBuf,
        child: PathBuf,
    },

    /// An error that occurs when doing I/O, such as reading an ignore file.
    Io(std::io::Error),

    /// An error that occurs when trying to parse a glob.
    Glob {
        glob: Option<String>,  // Original user-provided pattern
        err: String,
    },

    /// A type selection for a file type that is not defined.
    UnrecognizedFileType(String),

    /// A user specified file type definition could not be parsed.
    InvalidDefinition,
}
```

The wrapper variants (`WithLineNumber`, `WithPath`, `WithDepth`) create a nested structure that accumulates context as errors propagate upward.

## Section 4: Implementing Clone for Error

```
impl Clone for Error {
    fn clone(&self) -> Error {
        match *self {
            // Recursive variants clone their inner errors
            Error::Partial(ref errs) => Error::Partial(errs.clone()),
            Error::WithLineNumber { line, ref err } => {
                Error::WithLineNumber { line, err: err.clone() }
            }
            // ... other wrapper variants follow the same pattern ...

            // std::io::Error doesn't implement Clone, so we must rebuild it
            Error::Io(ref err) => match err.raw_os_error() {
                // If there's a raw OS error code, recreate from that
                Some(e) => Error::Io(std::io::Error::from_raw_os_error(e)),
                // Otherwise, create new error with same kind and string message
                None => {
                    Error::Io(std::io::Error::new(err.kind(), err.to_string()))
                }
            },

            // Simple variants clone directly
            Error::Glob { ref glob, ref err } => {
                Error::Glob { glob: glob.clone(), err: err.clone() }
            }
            Error::InvalidDefinition => Error::InvalidDefinition,
            // ...
        }
    }
}
```

The IO error cloning is a pragmatic workaround. Using `raw_os_error()` preserves the most useful information; falling back to `kind()` and `to_string()` captures the rest.

## Section 5: Error Inspection Methods

```rust
impl Error {
    /// Returns true if this is a partial error.
    pub fn is_partial(&self) -> bool {
        match *self {
            Error::Partial(_) => true,
            // Recursively check through context wrappers
            Error::WithLineNumber { ref err, .. } => err.is_partial(),
            Error::WithPath { ref err, .. } => err.is_partial(),
            Error::WithDepth { ref err, .. } => err.is_partial(),
            _ => false,
        }
    }

    /// Returns true if this error is exclusively an I/O error.
    pub fn is_io(&self) -> bool {
        match *self {
            // Special case: single-element Partial containing an IO error
            Error::Partial(ref errs) => errs.len() == 1 && errs[0].is_io(),
            // Unwrap context layers
            Error::WithLineNumber { ref err, .. } => err.is_io(),
            Error::WithPath { ref err, .. } => err.is_io(),
            Error::WithDepth { ref err, .. } => err.is_io(),
            Error::Loop { .. } => false,  // Loops are not IO errors
            Error::Io(_) => true,
            Error::Glob { .. } => false,
            Error::UnrecognizedFileType(_) => false,
            Error::InvalidDefinition => false,
        }
    }

    /// Inspect the original std::io::Error if there is one.
    /// Returns borrowed reference - use into_io_error() for ownership.
    pub fn io_error(&self) -> Option<&std::io::Error> {
        match *self {
            Error::Partial(ref errs) => {
                if errs.len() == 1 {
                    errs[0].io_error()  // Delegate to inner error
                } else {
                    None
                }
            }
            Error::WithLineNumber { ref err, .. } => err.io_error(),
            Error::WithPath { ref err, .. } => err.io_error(),
            Error::WithDepth { ref err, .. } => err.io_error(),
            Error::Io(ref err) => Some(err),  // Found it!
            _ => None,
        }
    }

    /// Consumes self to return owned std::io::Error.
    pub fn into_io_error(self) -> Option<std::io::Error> {
        match self {
            Error::Io(err) => Some(err),
            Error::WithPath { err, .. } => err.into_io_error(),
```

```
            // ... similar pattern for other wrappers            _ => None,
    }    }}
```

The borrowing (`io_error`) vs consuming (`into_io_error`) pattern lets callers choose based on whether they need ownership of the underlying error.

---

## Section 6: Error Context Builders

```
impl Error {
    /// Turn an error into a tagged error with the given file path.
    fn with_path<P: AsRef<Path>>(self, path: P) -> Error {
        Error::WithPath {
            path: path.as_ref().to_path_buf(),
            err: Box::new(self),  // Wrap self in Box
        }
    }

    /// Turn an error into a tagged error with the given depth.
    fn with_depth(self, depth: usize) -> Error {
        Error::WithDepth { depth, err: Box::new(self) }
    }

    /// Turn an error into a tagged error with path and line number.
    /// If path is empty, omit the path wrapper.
    fn tagged<P: AsRef<Path>>(self, path: P, lineno: u64) -> Error {
        let errline =
            Error::WithLineNumber { line: lineno, err: Box::new(self) };
        // Handle edge case: empty path shouldn't appear in error message
        if path.as_ref().as_os_str().is_empty() {
            return errline;
        }
        errline.with_path(path)  // Fluent chaining
    }

    /// Build an error from a walkdir error.
    fn from_walkdir(err: walkdir::Error) -> Error {
        let depth = err.depth();
        // Check for filesystem loop (special case)
        if let (Some(anc), Some(child)) = (err.loop_ancestor(), err.path()) {
            return Error::WithDepth {
                depth,
                err: Box::new(Error::Loop {
                    ancestor: anc.to_path_buf(),
                    child: child.to_path_buf(),
                }),
            };
        }
        // Convert to IO error with path context
        let path = err.path().map(|p| p.to_path_buf());
        let mut ig_err = Error::Io(std::io::Error::from(err));
        if let Some(path) = path {
            ig_err = Error::WithPath { path, err: Box::new(ig_err) };
        }
        ig_err
    }
}
```

These methods enable fluent error construction: `Error::Glob{..}.tagged(path, line)` wraps the glob error with location context in a single expression.

## Section 7: Standard Trait Implementations

```
impl std::error::Error for Error {
    #[allow(deprecated)]  // description() is deprecated but still required
    fn description(&self) -> &str {
        match *self {
            Error::Partial(_) => "partial error",
            // Delegate through wrappers to get the real description
            Error::WithLineNumber { ref err, .. } => err.description(),
            Error::WithPath { ref err, .. } => err.description(),
            Error::WithDepth { ref err, .. } => err.description(),
            Error::Loop { .. } => "file system loop found",
            Error::Io(ref err) => err.description(),
            Error::Glob { ref err, .. } => err,
            Error::UnrecognizedFileType(_) => "unrecognized file type",
            Error::InvalidDefinition => "invalid definition",
        }
    }
}

impl std::fmt::Display for Error {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match *self {
            // Join partial errors with newlines
            Error::Partial(ref errs) => {
                let msgs: Vec<String> =
                    errs.iter().map(|err| err.to_string()).collect();
                write!(f, "{}", msgs.join("\n"))
            }
            // Prepend line number
            Error::WithLineNumber { line, ref err } => {
                write!(f, "line {}: {}", line, err)
            }
            // Prepend path
            Error::WithPath { ref path, ref err } => {
                write!(f, "{}: {}", path.display(), err)
            }
            // Depth is metadata, not displayed
            Error::WithDepth { ref err, .. } => err.fmt(f),
            Error::Loop { ref ancestor, ref child } => write!(
                f,
                "File system loop found: {} points to an ancestor {}",
                child.display(),
                ancestor.display()
            ),
            Error::Io(ref err) => err.fmt(f),
            Error::Glob { glob: None, ref err } => write!(f, "{}", err),
            Error::Glob { glob: Some(ref glob), ref err } => {
                write!(f, "error parsing glob '{}': {}", glob, err)
            }
            // ...
        }
    }
}

// Enables Error::Io(io_error) construction via .into()
```

```
impl From<std::io::Error> for Error {    fn from(err: std::io::Error) -> Error {
Error::Io(err)    }}
```

The `Display` implementation recursively formats wrapped errors, producing messages like `"path/.gitignore: line 42: invalid glob"`.

---

## Section 8: The Match Enum

```rust
/// The result of a glob match.
///
/// The type parameter `T` typically refers to a type that provides more
/// information about a particular match.
#[derive(Clone, Debug)]
pub enum Match<T> {
    /// The path didn't match any glob.
    None,
    /// The highest precedent glob matched indicates the path should be ignored.
    Ignore(T),
    /// The highest precedent glob matched indicates the path should be
whitelisted.
    Whitelist(T),
}

impl<T> Match<T> {
    /// Returns true if the match result didn't match any globs.
    pub fn is_none(&self) -> bool {
        matches!(*self, Match::None)
    }

    /// Returns true if the match result implies the path should be ignored.
    pub fn is_ignore(&self) -> bool {
        matches!(*self, Match::Ignore(_))
    }

    /// Inverts the match so that Ignore becomes Whitelist and vice versa.
    pub fn invert(self) -> Match<T> {
        match self {
            Match::None => Match::None,
            Match::Ignore(t) => Match::Whitelist(t),
            Match::Whitelist(t) => Match::Ignore(t),
        }
    }

    /// Apply a function to the value inside this match.
    pub fn map<U, F: FnOnce(T) -> U>(self, f: F) -> Match<U> {
        match self {
            Match::None => Match::None,
            Match::Ignore(t) => Match::Ignore(f(t)),
            Match::Whitelist(t) => Match::Whitelist(f(t)),
        }
    }

    /// Return self if not none, otherwise return other.
    pub fn or(self, other: Self) -> Self {
        if self.is_none() { other } else { self }
    }
}
```

The `Match<T>` enum is generic over the metadata type, allowing different matchers to attach different information (glob pattern, file source, etc.) to their match results.

---

## Quick Reference

### Error Variant Summary

| Variant | Purpose |
|---|---|
| `Partial(Vec<Error>)` | Multiple errors, some operations may have succeeded |
| `WithLineNumber { line, err }` | Adds line number context |
| `WithPath { path, err }` | Adds file path context |
| `WithDepth { depth, err }` | Adds directory depth context |
| `Loop { ancestor, child }` | Filesystem loop via symlinks |
| `Io(std::io::Error)` | Underlying IO error |
| `Glob { glob, err }` | Invalid glob pattern |
| `UnrecognizedFileType(String)` | Unknown file type requested |
| `InvalidDefinition` | Malformed type definition |

### Match API

| Method | Returns | Purpose |
|---|---|---|
| `is_none()` | `bool` | Check if no match occurred |

| | | |
|---|---|---|
| `is_whitelist()` | `bool` | Check if path should be included |
| `invert()` | `Match<T>` | Swap ignore/whitelist |
| `map(f)` | `Match<U>` | Transform contained value |
| `or(other)` | `Match<T>` | Use other if self is None |
| `inner()` | `Option<&T>` | Access contained value |

## Public Exports

```
// From crate root
pub use walk::{DirEntry, ParallelVisitor, ParallelVisitorBuilder,
               Walk, WalkBuilder, WalkParallel, WalkState};

// Public modules for fine-grained control
pub mod gitignore;   // Gitignore parsing
pub mod overrides;   // Override patterns
pub mod types;       // File type matching
```

# Gitignore Parsing: Code Companion

Reference code for the Gitignore Parsing lecture. Sections correspond to the lecture document.

---

## Section 1: The Glob Data Structure

```rust
/// Glob represents a single glob in a gitignore file.
///
/// This is used to report information about the highest precedent glob that
/// matched in one or more gitignore files.
#[derive(Clone, Debug)]
pub struct Glob {
    /// The file path that this glob was extracted from.
    from: Option<PathBuf>,
    /// The original glob string (what user wrote).
    original: String,
    /// The actual glob string used to convert to a regex (transformed).
    actual: String,
    /// Whether this is a whitelisted glob or not (prefixed with !).
    is_whitelist: bool,
    /// Whether this glob should only match directories or not (suffixed with /).
    is_only_dir: bool,
}

impl Glob {
    /// Returns the file path that defined this glob.
    pub fn from(&self) -> Option<&Path> {
        self.from.as_ref().map(|p| &**p)
    }

    /// The original glob as it was defined in a gitignore file.
    pub fn original(&self) -> &str {
        &self.original
    }

    /// The actual glob that was compiled to respect gitignore semantics.
    pub fn actual(&self) -> &str {
        &self.actual
    }

    /// Whether this was a whitelisted glob or not.
    pub fn is_whitelist(&self) -> bool {
        self.is_whitelist
    }

    /// Whether this glob must match a directory or not.
    pub fn is_only_dir(&self) -> bool {
        self.is_only_dir
    }

    /// Returns true if and only if this glob has a `**/` prefix.
    fn has_doublestar_prefix(&self) -> bool {
        self.actual.starts_with("**/") || self.actual == "**"
    }
}
```

The `original` vs `actual` distinction preserves user input for diagnostics while storing the transformed pattern for matching. For example, `foo/` becomes `foo` in `actual` with `is_only_dir` set to `true`.

---

## Section 2: The Gitignore Matcher

```
/// Gitignore is a matcher for the globs in one or more gitignore files
/// in the same directory.
#[derive(Clone, Debug)]
pub struct Gitignore {
    /// The compiled glob set from the globset crate.
    set: GlobSet,
    /// The root directory for relative matching.
    root: PathBuf,
    /// All globs with their metadata.
    globs: Vec<Glob>,
    /// Count of ignore patterns (for optimization decisions).
    num_ignores: u64,
    /// Count of whitelist patterns (! prefixed).
    num_whitelists: u64,
    /// Object pool for scratch vectors during matching.
    matches: Option<Arc<Pool<Vec<usize>>>>,
}

impl Gitignore {
    /// Creates a new empty gitignore matcher that never matches anything.
    pub fn empty() -> Gitignore {
        Gitignore {
            set: GlobSet::empty(),
            root: PathBuf::from(""),
            globs: vec![],
            num_ignores: 0,
            num_whitelists: 0,
            matches: None,
        }
    }

    /// Returns true if and only if this gitignore has zero globs.
    pub fn is_empty(&self) -> bool {
        self.set.is_empty()
    }

    /// Returns the total number of globs.
    pub fn len(&self) -> usize {
        self.set.len()
    }
}
```

The `matches` field uses `Arc<Pool<Vec<usize>>>` for thread-safe scratch buffer reuse. The pool returns vectors that track which pattern indices matched, avoiding allocation on every match operation.

---

## Section 3: Path Stripping and Relative Matching

```
impl Gitignore {
    /// Strips the given path such that it's suitable for matching with this
    /// gitignore matcher.
    fn strip<'a, P: 'a + AsRef<Path> + ?Sized>(
        &'a self,
        path: &'a P,
    ) -> &'a Path {
        let mut path = path.as_ref();

        // A leading ./ is completely superfluous. We also strip it from
        // our gitignore root path, so we need to strip it from our candidate
        // path too.
        if let Some(p) = strip_prefix("./", path) {
            path = p;
        }

        // Strip any common prefix between the candidate path and the root
        // of the gitignore, to make sure we get relative matching right.
        // BUT, a file name might not have any directory components to it,
        // in which case, we don't want to accidentally strip any part of the
        // file name.
        //
        // As an additional special case, if the root is just `.`, then we
        // shouldn't try to strip anything, e.g., when path begins with a `.`.
        if self.root != Path::new(".") && !is_file_name(path) {
            if let Some(p) = strip_prefix(&self.root, path) {
                path = p;
                // If we're left with a leading slash, get rid of it.
                if let Some(p) = strip_prefix("/", path) {
                    path = p;
                }
            }
        }
        path
    }
}
```

The `is_file_name` check prevents stripping part of a bare filename that happens to match the root prefix. The lifetime `'a` ensures the returned path slice borrows from the input.

## Section 4: The Matching Algorithm

```
impl Gitignore {
    /// Returns whether the given path matched a pattern in this gitignore.
    pub fn matched<P: AsRef<Path>>(
        &self,
        path: P,
        is_dir: bool,
    ) -> Match<&Glob> {
        if self.is_empty() {
            return Match::None;
        }
        self.matched_stripped(self.strip(path.as_ref()), is_dir)
    }

    /// Like matched, but takes a path that has already been stripped.
    fn matched_stripped<P: AsRef<Path>>(
        &self,
        path: P,
        is_dir: bool,
    ) -> Match<&Glob> {
        if self.is_empty() {
            return Match::None;
        }
        let path = path.as_ref();

        // Get a scratch vector from the pool.
        let mut matches = self.matches.as_ref().unwrap().get();
        let candidate = Candidate::new(path);

        // Fill the vector with indices of all matching patterns.
        self.set.matches_candidate_into(&candidate, &mut *matches);

        // Iterate in REVERSE order: last pattern wins (highest precedence).
        for &i in matches.iter().rev() {
            let glob = &self.globs[i];
            // Directory-only patterns must match directories.
            if !glob.is_only_dir() || is_dir {
                return if glob.is_whitelist() {
                    Match::Whitelist(glob)
                } else {
                    Match::Ignore(glob)
                };
            }
        }
        Match::None
    }
}
```

The reverse iteration implements gitignore's "last pattern wins" semantics. The `is_dir` parameter is filesystem knowledge that the glob matcher can't determine from the pattern alone.

---

## Section 5: The Builder Pattern

```rust
/// Builds a matcher for a single set of globs from a .gitignore file.
#[derive(Clone, Debug)]
pub struct GitignoreBuilder {
    builder: GlobSetBuilder,
    root: PathBuf,
    globs: Vec<Glob>,
    case_insensitive: bool,
    /// Defaults to true to match git's permissive behavior.
    allow_unclosed_class: bool,
}

impl GitignoreBuilder {
    /// Create a new builder for a gitignore file.
    pub fn new<P: AsRef<Path>>(root: P) -> GitignoreBuilder {
        let root = root.as_ref();
        GitignoreBuilder {
            builder: GlobSetBuilder::new(),
            // Strip ./ prefix during construction, not on every match.
            root: strip_prefix("./", root).unwrap_or(root).to_path_buf(),
            globs: vec![],
            case_insensitive: false,
            allow_unclosed_class: true,  // Match git's behavior
        }
    }

    /// Builds a new matcher from the globs added so far.
    pub fn build(&self) -> Result<Gitignore, Error> {
        let nignore = self.globs.iter().filter(|g| !g.is_whitelist()).count();
        let nwhite = self.globs.iter().filter(|g| g.is_whitelist()).count();
        let set = self
            .builder
            .build()
            .map_err(|err| Error::Glob { glob: None, err: err.to_string() })?;
        Ok(Gitignore {
            set,
            root: self.root.clone(),
            globs: self.globs.clone(),
            num_ignores: nignore as u64,
            num_whitelists: nwhite as u64,
            matches: Some(Arc::new(Pool::new(|| vec![]))),
        })
    }
}
```

The builder wraps a `GlobSetBuilder` internally, maintaining the abstraction layer. The pool is created at build time with a closure that produces empty vectors on demand.

## Section 6: Parsing Individual Lines

```
impl GitignoreBuilder {
    /// Add a line from a gitignore file to this builder.
    pub fn add_line(
        &mut self,
        from: Option<PathBuf>,
        mut line: &str,
    ) -> Result<&mut GitignoreBuilder, Error> {
        // Skip comments.
        if line.starts_with("#") {
            return Ok(self);
        }
        // Trim trailing whitespace unless escaped.
        if !line.ends_with("\\ ") {
            line = line.trim_right();
        }
        // Skip blank lines.
        if line.is_empty() {
            return Ok(self);
        }

        let mut glob = Glob {
            from,
            original: line.to_string(),
            actual: String::new(),
            is_whitelist: false,
            is_only_dir: false,
        };
        let mut is_absolute = false;

        // Handle escaped ! or # at start (literal match).
        if line.starts_with("\\!") || line.starts_with("\\#") {
            line = &line[1..];
            is_absolute = line.chars().nth(0) == Some('/');
        } else {
            // Check for whitelist prefix.
            if line.starts_with("!") {
                glob.is_whitelist = true;
                line = &line[1..];
            }
            // Leading slash means anchor to root.
            if line.starts_with("/") {
                line = &line[1..];
                is_absolute = true;
            }
        }

        // Trailing slash means directory-only match.
        if line.as_bytes().last() == Some(&b'/') {
            glob.is_only_dir = true;
            line = &line[..line.len() - 1];
            // Handle escaped trailing slash.
            if line.as_bytes().last() == Some(&b'\\') {
                line = &line[..line.len() - 1];
            }
```

```
        }        glob.actual = line.to_string();        // Patterns without /
match anywhere (add **/ prefix).        if !is_absolute && !line.chars().any(|c|
c == '/') {            if !glob.has_doublestar_prefix() {
glob.actual = format!("**/{}", glob.actual);            }        }        // Fix
/** to not match the directory itself.        if glob.actual.ends_with("/**") {
glob.actual = format!("{}/*", glob.actual);        }        // Build the actual
glob with gitignore-specific options.        let parsed =
GlobBuilder::new(&glob.actual)            .literal_separator(true)      // *
doesn't match /            .case_insensitive(self.case_insensitive)
.backslash_escape(true)       // \ escapes special chars
.allow_unclosed_class(self.allow_unclosed_class)            .build()
.map_err(|err| Error::Glob {               glob: Some(glob.original.clone()),
err: err.kind().to_string(),            })?;        self.builder.add(parsed);
self.globs.push(glob);        Ok(self)    }}
```

The transformation order matters: whitelist detection happens before slash stripping, and `**/` prefixing happens after both. The `literal_separator(true)` ensures `*` won't match path separators, matching gitignore semantics.

## Quick Reference

### Pattern Transformations

| Input Pattern | Transformed Pattern | Semantics |
|---|---|---|
| `foo` | `**/foo` | Match anywhere in tree |
| `/foo` | `foo` | Match only at root |
| `foo/` | `foo` + `is_only_dir=true` | Match directories only |
| `!foo` | `**/foo` + `is_whitelist=true` | Un-ignore pattern |
| `\!foo` | `**/!foo` | Literal `!` in name |
| `foo/**` | `foo/**/*` | Match contents, not directory |

### Match Result Types

```rust
pub enum Match<T> {
    None,           // No pattern matched
    Ignore(T),      // Should be ignored
    Whitelist(T),   // Explicitly un-ignored
}
```

## Key Builder Options

| Option | Default | Purpose |
|---|---|---|
| `case_insensitive` | `false` | Case-insensitive matching |
| `allow_unclosed_class` | `true` | Treat `[abc` as literal (git compat) |

## Precedence Rules

1. Last matching pattern wins
2. Whitelist (`!`) can override earlier ignores
3. Directory-only patterns (`foo/`) only match if `is_dir=true`
4. Patterns in child directories override parent patterns

# ripgrep crates/ignore/src/types.rs: Code Companion

Reference code for the File Type Definitions lecture. Sections correspond to the lecture document.

---

## Section 1: The Builder Pattern for Type Matching

```rust
/// TypesBuilder builds a type matcher from a set of file type definitions and
/// a set of file type selections.
pub struct TypesBuilder {
    // HashMap for O(1) lookup of type definitions by name
    types: HashMap<String, FileTypeDef>,
    // Vec preserves insertion order - later selections have higher precedence
    selections: Vec<Selection<()>>,
}

impl TypesBuilder {
    /// Create a new builder for a file type matcher.
    ///
    /// The builder contains *no* type definitions to start with.
    pub fn new() -> TypesBuilder {
        TypesBuilder { types: HashMap::new(), selections: vec![] }
    }

    /// Build the current set of file type definitions *and* selections into
    /// a file type matcher.
    pub fn build(&self) -> Result<Types, Error> {
        let defs = self.definitions();
        // Precompute whether any positive selections exist
        let has_selected = self.selections.iter().any(|s| !s.is_negated());

        let mut selections = vec![];
        let mut glob_to_selection = vec![];
        let mut build_set = GlobSetBuilder::new();

        // Phase 2: Resolve selections to actual definitions
        for (isel, selection) in self.selections.iter().enumerate() {
            let def = match self.types.get(selection.name()) {
                Some(def) => def.clone(),
                None => {
                    // Fail if user selected a type that doesn't exist
                    let name = selection.name().to_string();
                    return Err(Error::UnrecognizedFileType(name));
                }
            };
            // ... builds glob set and mappings ...
        }
        // ... returns compiled Types ...
    }
}
```

The two-phase design allows definitions and selections to be added in any order. Validation only occurs during `build()`, when all information is available.

## Section 2: The Selection Enum and Generic Design

```rust
/// Indicates the type of a selection for a particular file type.
#[derive(Clone, Debug)]
enum Selection<T> {
    Select(String, T),  // Include files matching this type
    Negate(String, T),  // Exclude files matching this type
}

impl<T> Selection<T> {
    fn is_negated(&self) -> bool {
        match *self {
            Selection::Select(..) => false,
            Selection::Negate(..) => true,
        }
    }

    fn name(&self) -> &str {
        match *self {
            Selection::Select(ref name, _) => name,
            Selection::Negate(ref name, _) => name,
        }
    }

    /// Transform Selection<T> to Selection<U> by applying f to the inner value
    fn map<U, F: FnOnce(T) -> U>(self, f: F) -> Selection<U> {
        match self {
            Selection::Select(name, inner) => {
                Selection::Select(name, f(inner))
            }
            Selection::Negate(name, inner) => {
                Selection::Negate(name, f(inner))
            }
        }
    }

    fn inner(&self) -> &T {
        match *self {
            Selection::Select(_, ref inner) => inner,
            Selection::Negate(_, ref inner) => inner,
        }
    }
}
```

During building, `T` is `()` (unit type—no data). After building, `T` is `FileTypeDef`. The `map` method transforms between these states while preserving the Select/Negate distinction.

## Section 3: The Types Matcher Structure

```rust
/// Types is a file type matcher.
#[derive(Clone, Debug)]
pub struct Types {
    /// All of the file type definitions, sorted lexicographically by name.
    defs: Vec<FileTypeDef>,
    /// All of the selections made by the user.
    selections: Vec<Selection<FileTypeDef>>,
    /// Whether there is at least one Selection::Select in our selections.
    /// When this is true, a Match::None is converted to Match::Ignore.
    has_selected: bool,
    /// A mapping from glob index in the set to two indices. The first is an
    /// index into `selections` and the second is an index into the
    /// corresponding file type definition's list of globs.
    glob_to_selection: Vec<(usize, usize)>,
    /// The set of all glob selections, used for actual matching.
    set: GlobSet,
    /// Temporary storage for globs that match - thread-safe pool.
    matches: Arc<Pool<Vec<usize>>>,
}

impl Types {
    /// Creates a new file type matcher that never matches any path and
    /// contains no file type definitions.
    pub fn empty() -> Types {
        Types {
            defs: vec![],
            selections: vec![],
            has_selected: false,
            glob_to_selection: vec![],
            set: GlobSetBuilder::new().build().unwrap(),
            // Pool creates Vec<usize> on demand for each thread
            matches: Arc::new(Pool::new(|| vec![])),
        }
    }
}
```

The `glob_to_selection` vector enables reverse lookup: given a glob match index, find which selection and which glob within that selection matched.

## Section 4: The Glob Wrapper Type

```rust
/// The lifetime ``a` refers to the lifetime of the underlying file type
/// definition, which corresponds to the lifetime of the file type matcher.
#[derive(Clone, Debug)]
pub struct Glob<'a>(GlobInner<'a>);

#[derive(Clone, Debug)]
enum GlobInner<'a> {
    /// No glob matched, but the file path should still be ignored.
    UnmatchedIgnore,
    /// A glob matched.
    Matched {
        /// The file type definition which provided the glob.
        def: &'a FileTypeDef,
    },
}

impl<'a> Glob<'a> {
    fn unmatched() -> Glob<'a> {
        Glob(GlobInner::UnmatchedIgnore)
    }

    /// Return the file type definition that matched, if one exists.
    pub fn file_type_def(&self) -> Option<&FileTypeDef> {
        match self {
            Glob(GlobInner::UnmatchedIgnore) => None,
            Glob(GlobInner::Matched { def, .. }) => Some(def),
        }
    }
}
```

The inner enum is private; callers only see the `file_type_def()` method. The lifetime `'a` ties the returned reference to the `Types` matcher's lifetime.

---

## Section 5: The Matching Logic

```
impl Types {
    /// Returns a match for the given path against this file type matcher.
    pub fn matched<'a, P: AsRef<Path>>(
        &'a self,
        path: P,
        is_dir: bool,
    ) -> Match<Glob<'a>> {
        // File types don't apply to directories
        if is_dir || self.set.is_empty() {
            return Match::None;
        }
        // Extract just the file name for matching
        let name = match file_name(path.as_ref()) {
            Some(name) => name,
            None if self.has_selected => {
                return Match::Ignore(Glob::unmatched());
            }
            None => {
                return Match::None;
            }
        };
        // Get a reusable Vec from the pool
        let mut matches = self.matches.get();
        self.set.matches_into(name, &mut *matches);

        // The highest precedent match is the last one
        if let Some(&i) = matches.last() {
            let (isel, _) = self.glob_to_selection[i];
            let sel = &self.selections[isel];
            let glob = Glob(GlobInner::Matched { def: sel.inner() });
            return if sel.is_negated() {
                Match::Ignore(glob)
            } else {
                Match::Whitelist(glob)
            };
        }
        // No match: ignore if types were selected, else no opinion
        if self.has_selected {
            Match::Ignore(Glob::unmatched())
        } else {
            Match: None
        }
    }
}
```

The `matches.last()` call implements precedence: later selections override earlier ones. The pool-based allocation (`self.matches.get()`) avoids per-file heap allocations.

# Section 6: Building Type Definitions

```
impl TypesBuilder {
    /// Add a new file type definition.
    pub fn add(&mut self, name: &str, glob: &str) -> Result<(), Error> {
        // Validate: "all" is reserved, names must be alphanumeric
        if name == "all" || !name.chars().all(|c| c.is_alphanumeric()) {
            return Err(Error::InvalidDefinition);
        }
        let (key, glob) = (name.to_string(), glob.to_string());
        // entry() avoids double lookup for "exists?" then "insert"
        self.types
            .entry(key)
            .or_insert_with(|| FileTypeDef {
                name: name.to_string(),
                globs: vec![],
            })
            .globs
            .push(glob);
        Ok(())
    }

    /// Add a new file type definition specified in string form.
    /// Formats: `{name}:{glob}` or `{name}:include:{type1,type2,...}`
    pub fn add_def(&mut self, def: &str) -> Result<(), Error> {
        let parts: Vec<&str> = def.split(':').collect();
        match parts.len() {
            2 => {
                // Basic format: name:glob
                let name = parts[0];
                let glob = parts[1];
                if name.is_empty() || glob.is_empty() {
                    return Err(Error::InvalidDefinition);
                }
                self.add(name, glob)
            }
            3 => {
                // Include format: name:include:type1,type2
                let name = parts[0];
                let types_string = parts[2];
                if name.is_empty()
                    || parts[1] != "include"
                    || types_string.is_empty()
                {
                    return Err(Error::InvalidDefinition);
                }
                let types = types_string.split(',');
                // Fail fast if any referenced type doesn't exist
                if types.clone().any(|t| !self.types.contains_key(t)) {
                    return Err(Error::InvalidDefinition);
                }
                // Copy globs from included types (evaluated immediately)
                for type_name in types {
                    let globs =
                        self.types.get(type_name).unwrap().globs.clone();
                    for glob in globs {
```

```
                        self.add(name, &glob)?;                    }
}               Ok(())              }             _ =>
Err(Error::InvalidDefinition),          }     }}
```

The include format copies globs at definition time, not at match time. This means the composite type is frozen when defined.

---

## Quick Reference

### File Type Definition Structure

```
pub struct FileTypeDef {
    name: String,          // e.g., "rust"
    globs: Vec<String>,    // e.g., ["*.rs"]
}
```

### Key Methods

| Method | Purpose |
|---|---|
| `TypesBuilder::new()` | Create empty builder |
| `builder.add_defaults()` | Load built-in type definitions |
| `builder.add(name, glob)` | Add single glob to type |
| `builder.add_def(def)` | Parse and add string definition |
| `builder.select(name)` | Include files matching type |
| `builder.negate(name)` | Exclude files matching type |
| `builder.build()` | Compile to immutable `Types` |
| `types.matched(path, is_dir)` | Check if path matches selections |

## Match Results

| Result | Meaning |
|---|---|
| `Match::Whitelist(glob)` | File matches a selected type |
| `Match::Ignore(glob)` | File matches negated type OR doesn't match any selected type |
| `Match::None` | No selections active, no filtering |

## Definition String Formats

```
rust:*.rs              # Basic: name:glob
web:include:html,css   # Composite: name:include:types
```

# Parallel Directory Walking: Code Companion

Reference code for the Parallel Directory Walking lecture. Sections correspond to the lecture document.

---

## Section 1: The Directory Entry Abstraction

```rust
/// A directory entry with a possible error attached.
///
/// The error typically refers to a problem parsing ignore files in a
/// particular directory.
#[derive(Clone, Debug)]
pub struct DirEntry {
    dent: DirEntryInner,
    err: Option<Error>,  // Error attachment pattern - errors travel with entries
}

/// DirEntryInner is the implementation of DirEntry.
///
/// It specifically represents three distinct sources of directory entries:
///
/// 1. From the walkdir crate.
/// 2. Special entries that represent things like stdin.
/// 3. From a path.
#[derive(Clone, Debug)]
enum DirEntryInner {
    Stdin,                       // Pseudo-entry for stdin input
    Walkdir(walkdir::DirEntry), // From sequential iteration
    Raw(DirEntryRaw),           // From parallel iteration
}

impl DirEntryInner {
    fn path(&self) -> &Path {
        use self::DirEntryInner::*;
        match *self {
            Stdin => Path::new("<stdin>"),
            Walkdir(ref x) => x.path(),
            Raw(ref x) => x.path(),
        }
    }

    fn metadata(&self) -> Result<Metadata, Error> {
        use self::DirEntryInner::*;
        match *self {
            Stdin => {
                // Stdin has no real metadata - synthesize an error
                let err = Error::Io(io::Error::new(
                    io::ErrorKind::Other,
                    "<stdin> has no metadata",
                ));
                Err(err.with_path("<stdin>"))
            }
            Walkdir(ref x) => x.metadata().map_err(|err| {
                Error::Io(io::Error::from(err)).with_path(x.path())
            }),
            Raw(ref x) => x.metadata(),
        }
    }

    fn is_dir(&self) -> bool {
        self.file_type().map(|ft| ft.is_dir()).unwrap_or(false)
```

```
    }}
```

The three-variant enum provides a unified interface regardless of entry source. The `err` field enables the "error attachment" pattern where parsing errors don't abort traversal.

---

## Section 2: Platform-Specific Raw Entries

```rust
/// DirEntryRaw is essentially copied from the walkdir crate so that we can
/// build `DirEntry`s from whole cloth in the parallel iterator.
#[derive(Clone)]
struct DirEntryRaw {
    path: PathBuf,
    ty: FileType,
    follow_link: bool,
    depth: usize,
    #[cfg(unix)]
    ino: u64,              // Unix: inode comes cheap from readdir
    #[cfg(windows)]
    metadata: fs::Metadata, // Windows: full metadata comes free
}

impl DirEntryRaw {
    // Platform-specific construction from directory entry
    #[cfg(unix)]
    fn from_entry_os(
        depth: usize,
        ent: &fs::DirEntry,
        ty: fs::FileType,
    ) -> Result<DirEntryRaw, Error> {
        use std::os::unix::fs::DirEntryExt;

        Ok(DirEntryRaw {
            path: ent.path(),
            ty,
            follow_link: false,
            depth,
            ino: ent.ino(),  // Cheap on Unix - no stat required
        })
    }

    #[cfg(windows)]
    fn from_entry_os(
        depth: usize,
        ent: &fs::DirEntry,
        ty: fs::FileType,
    ) -> Result<DirEntryRaw, Error> {
        let md = ent.metadata().map_err(|err| {
            let err = Error::Io(io::Error::from(err)).with_path(ent.path());
            Error::WithDepth { depth, err: Box::new(err) }
        })?;
        Ok(DirEntryRaw {
            path: ent.path(),
            ty,
            follow_link: false,
            depth,
            metadata: md,   // Free on Windows during directory reading
        })
    }

    // Platform-specific metadata access
    #[cfg(windows)]
```

```
    fn metadata_internal(&self) -> Result<fs::Metadata, Error> {        if
self.follow_link {            fs::metadata(&self.path)  // Follow symlink - need
fresh stat        } else {            Ok(self.metadata.clone()) // Use cached
metadata        }        .map_err(|err|
Error::Io(io::Error::from(err)).with_path(&self.path))    }    #
[cfg(not(windows))]    fn metadata_internal(&self) -> Result<fs::Metadata, Error>
{        if self.follow_link {            fs::metadata(&self.path)        } else
{            fs::symlink_metadata(&self.path)  // Don't follow symlinks        }
.map_err(|err| Error::Io(io::Error::from(err)).with_path(&self.path))    }}
```

The `#[cfg(unix)]` and `#[cfg(windows)]` attributes compile completely different struct layouts and implementations. This avoids runtime branching and stores only what's cheap to obtain on each platform.

## Section 3: WalkBuilder and the Builder Pattern

```rust
#[derive(Clone)]
pub struct WalkBuilder {
    paths: Vec<PathBuf>,
    ig_builder: IgnoreBuilder,
    max_depth: Option<usize>,
    min_depth: Option<usize>,
    max_filesize: Option<u64>,
    follow_links: bool,
    same_file_system: bool,
    sorter: Option<Sorter>,
    threads: usize,
    skip: Option<Arc<Handle>>,
    filter: Option<Filter>,
    /// Lazy initialization - only query CWD if needed, and only once
    global_gitignores_relative_to:
        OnceLock<Result<PathBuf, Arc<std::io::Error>>>,
}

impl WalkBuilder {
    /// Create a new builder for a recursive directory iterator
    pub fn new<P: AsRef<Path>>(path: P) -> WalkBuilder {
        WalkBuilder {
            paths: vec![path.as_ref().to_path_buf()],
            ig_builder: IgnoreBuilder::new(),
            max_depth: None,
            // ... other defaults
            threads: 0,  // 0 means auto-detect
            global_gitignores_relative_to: OnceLock::new(),
        }
    }

    /// Build a new `Walk` iterator (sequential)
    pub fn build(&self) -> Walk {
        // ... configuration applied to walkdir
    }

    /// Build a new `WalkParallel` iterator
    /// Note: returns a type that does NOT implement Iterator
    pub fn build_parallel(&self) -> WalkParallel {
        let ig_root = self
            .get_or_set_current_dir()
            .map(|cwd| self.ig_builder.build_with_cwd(Some(cwd.to_path_buf())))
            .unwrap_or_else(|| self.ig_builder.build());
        WalkParallel {
            paths: self.paths.clone().into_iter(),
            ig_root,
            // ... other configuration
        }
    }

    /// Lazy CWD initialization - called once, result cached
    fn get_or_set_current_dir(&self) -> Option<&Path> {
        let result = self.global_gitignores_relative_to.get_or_init(|| {
            let result = std::env::current_dir().map_err(Arc::new);
```

```
            match result {                Ok(ref path) => {
log::trace!(                          "automatically discovered CWD: {}",
path.display()                    );                }                Err(ref err)
=> {                    log::debug!(                          "failed to find CWD \
(global gitignores will be ignored): {err}"                    );
}            }            result        });
result.as_ref().ok().map(|path| &**path)    }}
```

The `OnceLock` ensures the potentially expensive CWD lookup happens at most once. The builder can produce two fundamentally different types: `Walk` (implements `Iterator`) and `WalkParallel` (callback-based).

---

## Section 4: The Single-Threaded Walk Iterator

```
pub struct Walk {
    its: std::vec::IntoIter<(PathBuf, Option<WalkEventIter>)>,
    it: Option<WalkEventIter>,
    ig_root: Ignore,
    ig: Ignore,           // Current ignore state - changes as we traverse
    max_filesize: Option<u64>,
    skip: Option<Arc<Handle>>,
    filter: Option<Filter>,
}

impl Walk {
    fn skip_entry(&self, ent: &DirEntry) -> Result<bool, Error> {
        if ent.depth() == 0 {
            return Ok(false);  // Never skip root entries
        }
        // CRITICAL: Trivial skipping BEFORE expensive filesystem operations
        // This prevents unnecessary file downloads on remote filesystems
        if should_skip_entry(&self.ig, ent) {
            return Ok(true);
        }
        if let Some(ref stdout) = self.skip {
            if path_equals(ent, stdout)? {
                return Ok(true);
            }
        }
        // File size check requires metadata - do it last
        if self.max_filesize.is_some() && !ent.is_dir() {
            return Ok(skip_filesize(
                self.max_filesize.unwrap(),
                ent.path(),
                &ent.metadata().ok(),
            ));
        }
        Ok(false)
    }
}

impl Iterator for Walk {
    type Item = Result<DirEntry, Error>;

    fn next(&mut self) -> Option<Result<DirEntry, Error>> {
        loop {
            let ev = match self.it.as_mut().and_then(|it| it.next()) {
                Some(ev) => ev,
                None => {
                    // Move to next root path
                    match self.its.next() {
                        None => return None,
                        Some((_, None)) => {
                            return Some(Ok(DirEntry::new_stdin()));
                        }
                        Some((path, Some(it))) => {
                            self.it = Some(it);
                            if path.is_dir() {
```

```
                                // Add parent ignore rules for new root
let (ig, err) = self.ig_root.add_parents(path);
self.ig = ig;                                   if let Some(err) = err {
return Some(Err(err));                                  }
}                       }                       }                   continue;
}           };          match ev {                  Err(err) => return
Some(Err(Error::from_walkdir(err))),                Ok(WalkEvent::Exit) => {
// Pop ignore rules when leaving directory                      self.ig =
self.ig.parent().unwrap();                  }
Ok(WalkEvent::Dir(ent)) => {                    // Push new ignore rules when
entering directory                      let (igtmp, err) =
self.ig.add_child(ent.path());                      self.ig = igtmp;
// ... return directory entry                   }
Ok(WalkEvent::File(ent)) => {                   // ... handle file entry
}           }       }   }}
```

The `ig` field maintains the current ignore matcher state. It grows when entering directories (via `add_child`) and shrinks when exiting (via `parent()`). The `skip_entry` method orders checks from cheapest to most expensive.

## Section 5: WalkEvent and Directory Structure

```rust
/// Transforms flat walkdir stream into structured enter/exit events
struct WalkEventIter {
    depth: usize,
    it: walkdir::IntoIter,
    next: Option<Result<walkdir::DirEntry, walkdir::Error>>,  // Lookahead buffer
}

#[derive(Debug)]
enum WalkEvent {
    Dir(walkdir::DirEntry),   // Entering a directory
    File(walkdir::DirEntry),  // A file (non-directory)
    Exit,                     // Leaving a directory
}

impl Iterator for WalkEventIter {
    type Item = walkdir::Result<WalkEvent>;

    fn next(&mut self) -> Option<walkdir::Result<WalkEvent>> {
        // Take from lookahead buffer, or get next from underlying iterator
        let dent = self.next.take().or_else(|| self.it.next());

        let depth = match dent {
            None => 0,
            Some(Ok(ref dent)) => dent.depth(),
            Some(Err(ref err)) => err.depth(),
        };

        // Key insight: if depth decreased, we've finished a directory
        if depth < self.depth {
            self.depth -= 1;
            self.next = dent;  // Stash for next call
            return Some(Ok(WalkEvent::Exit));  // Synthesize exit event
        }

        self.depth = depth;
        match dent {
            None => None,
            Some(Err(err)) => Some(Err(err)),
            Some(Ok(dent)) => {
                if walkdir_is_dir(&dent) {
                    self.depth += 1;
                    Some(Ok(WalkEvent::Dir(dent)))
                } else {
                    Some(Ok(WalkEvent::File(dent)))
                }
            }
        }
    }
}
```

The lookahead-and-stash pattern: when depth decreases, we save the current entry in `self.next`, emit a synthetic `Exit` event, and return the stashed entry on the subsequent call. This transforms the flat stream into a structured tree traversal.

---

## Section 6: WalkState and Parallel Visitor Traits

```rust
/// Control flow for parallel traversal
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
pub enum WalkState {
    Continue,  // Keep walking
    Skip,      // Don't descend into this directory
    Quit,      // Stop all workers as soon as possible
}

/// Builder creates per-thread visitors - no sharing needed
pub trait ParallelVisitorBuilder<'s> {
    fn build(&mut self) -> Box<dyn ParallelVisitor + 's>;
}

/// Each thread gets its own visitor instance
pub trait ParallelVisitor: Send {
    fn visit(&mut self, entry: Result<DirEntry, Error>) -> WalkState;
}

// Convenience wrapper for closure-based visitors
struct FnBuilder<F> {
    builder: F,
}

impl<'s, F: FnMut() -> FnVisitor<'s>> ParallelVisitorBuilder<'s>
    for FnBuilder<F>
{
    fn build(&mut self) -> Box<dyn ParallelVisitor + 's> {
        let visitor = (self.builder)();  // Call factory to create visitor
        Box::new(FnVisitorImp { visitor })
    }
}

type FnVisitor<'s> =
    Box<dyn FnMut(Result<DirEntry, Error>) -> WalkState + Send + 's>;
```

The `ParallelVisitorBuilder` creates separate visitors for each thread, enabling lock-free accumulation of results in thread-local storage. The `'s` lifetime parameter allows visitors to borrow from their environment.

# Section 7: The Work-Stealing Stack

```rust
/// A work-stealing stack
#[derive(Debug)]
struct Stack {
    index: usize,                       // This thread's index
    deque: Deque<Message>,              // Thread-local work stack
    stealers: Arc<[Stealer<Message>]>, // Access to other threads' work
}

impl Stack {
    /// Create stacks for all threads, distribute initial work
    fn new_for_each_thread(threads: usize, init: Vec<Message>) -> Vec<Stack> {
        // LIFO ensures depth-first traversal - critical for memory usage
        let deques: Vec<Deque<Message>> =
            std::iter::repeat_with(Deque::new_lifo).take(threads).collect();

        let stealers = Arc::<[Stealer<Message>]>::from(
            deques.iter().map(Deque::stealer).collect::<Vec<_>>(),
        );

        let stacks: Vec<Stack> = deques
            .into_iter()
            .enumerate()
            .map(|(index, deque)| Stack {
                index,
                deque,
                stealers: stealers.clone(),
            })
            .collect();

        // Round-robin distribution of initial work
        init.into_iter()
            .rev()  // Reverse to cancel LIFO effect
            .zip(stacks.iter().cycle())
            .for_each(|(m, s)| s.push(m));

        stacks
    }

    fn pop(&self) -> Option<Message> {
        // Try local first, then steal from others
        self.deque.pop().or_else(|| self.steal())
    }

    fn steal(&self) -> Option<Message> {
        // Fairness: steal from index+1, index+2, ..., wrap to 0, 1, ...
        let (left, right) = self.stealers.split_at(self.index);
        let right = &right[1..];  // Don't steal from ourselves

        right
            .iter()
            .chain(left.iter())
            .map(|s| s.steal_batch_and_pop(&self.deque))
            .find_map(|s| s.success())
    }
```

```
}
```

The `crossbeam_deque` crate provides the work-stealing primitives. Using `new_lifo()` ensures depth-first traversal, which dramatically reduces memory usage on wide directory trees. The steal operation takes a batch to amortize synchronization overhead.

---

## Section 8: Worker Thread Implementation

```rust
struct Worker<'s> {
    visitor: Box<dyn ParallelVisitor + 's>,
    stack: Stack,
    quit_now: Arc<AtomicBool>,           // Global quit signal
    active_workers: Arc<AtomicUsize>,    // Termination detection
    max_depth: Option<usize>,
    // ... other configuration
}

impl<'s> Worker<'s> {
    fn run(mut self) {
        while let Some(work) = self.get_work() {
            if let WalkState::Quit = self.run_one(work) {
                self.quit_now();
            }
        }
    }

    fn get_work(&mut self) -> Option<Work> {
        let mut value = self.recv();
        loop {
            // Priority channel simulation: quit beats all other messages
            if self.is_quit_now() {
                value = Some(Message::Quit)
            }
            match value {
                Some(Message::Work(work)) => return Some(work),
                Some(Message::Quit) => {
                    // Propagate quit to wake other threads (domino effect)
                    self.send_quit();
                    return None;
                }
                None => {
                    // No work available - check if we're done
                    if self.deactivate_worker() == 0 {
                        // All workers idle + all deques empty = done
                        self.send_quit();
                        return None;
                    }
                    // Wait for work to appear
                    loop {
                        if let Some(v) = self.recv() {
                            self.activate_worker();
                            value = Some(v);
                            break;
                        }
                        // Sleep to avoid CPU spin
                        std::thread::sleep(std::time::Duration::from_millis(1));
                    }
                }
            }
        }
    }
```

```
    /// Deactivates a worker and returns count of still-active workers    fn
deactivate_worker(&self) -> usize {        self.active_workers.fetch_sub(1,
AtomicOrdering::Acquire) - 1    }    fn activate_worker(&self) {
self.active_workers.fetch_add(1, AtomicOrdering::Release);    }}
```

Termination detection uses an atomic counter: when a worker finds no work, it decrements `active_workers`. If this returns 0, all workers are simultaneously idle with empty deques, meaning work is truly exhausted. The `Acquire`/`Release` ordering ensures proper synchronization.

---

## Section 9: Directory Processing and Work Generation

```
struct Work {
    dent: DirEntry,
    ignore: Ignore,           // Ignore rules for this directory's parents
    root_device: Option<u64>, // For same_file_system check
}

impl Work {
    fn read_dir(&mut self) -> Result<fs::ReadDir, Error> {
        let readdir = match fs::read_dir(self.dent.path()) {
            Ok(readdir) => readdir,
            Err(err) => {
                return Err(Error::from(err)
                    .with_path(self.dent.path())
                    .with_depth(self.dent.depth()));
            }
        };
        // Add ignore rules from this directory's .gitignore etc
        let (ig, err) = self.ignore.add_child(self.dent.path());
        self.ignore = ig;
        self.dent.err = err;  // Attach parsing error to entry
        Ok(readdir)
    }
}

impl<'s> Worker<'s> {
    fn generate_work(
        &mut self,
        ig: &Ignore,
        depth: usize,
        root_device: Option<u64>,
        result: Result<fs::DirEntry, io::Error>,
    ) -> WalkState {
        let fs_dent = match result {
            Ok(fs_dent) => fs_dent,
            Err(err) => {
                return
self.visitor.visit(Err(Error::from(err).with_depth(depth)));
            }
        };

        let mut dent = match DirEntryRaw::from_entry(depth, &fs_dent) {
            Ok(dent) => DirEntry::new_raw(dent, None),
            Err(err) => return self.visitor.visit(Err(err)),
        };

        // Handle symlinks - potentially resolve and check for loops
        if self.follow_links && dent.file_type().map_or(false, |ft|
ft.is_symlink()) {
            // ... symlink resolution code
            if dent.is_dir() {
                if let Err(err) = check_symlink_loop(ig, dent.path(), depth) {
                    return self.visitor.visit(Err(err));
                }
            }
```

```
        }        // CRITICAL: Ignore check BEFORE expensive operations        if
should_skip_entry(ig, &dent) {            return WalkState::Continue;        }
// Additional checks: stdout skip, filesize, filter        // ...        // Push
work for later processing        self.send(Work { dent, ignore: ig.clone(),
root_device });        WalkState::Continue    }}
```

Each directory entry goes through: raw entry construction, symlink handling (with loop detection), ignore matching, and finally work submission. The ignore check happens early to avoid expensive metadata operations on files that will be skipped anyway.

---

## Quick Reference

### Key Types

| Type | Purpose |
|---|---|
| `DirEntry` | Unified directory entry with attached error |
| `DirEntryInner` | Three-variant enum: `Stdin`, `Walkdir`, `Raw` |
| `DirEntryRaw` | Platform-specific raw entry for parallel iteration |
| `WalkBuilder` | Configurable builder for both iterator types |
| `Walk` | Sequential iterator (implements `Iterator`) |
| `WalkParallel` | Parallel traversal (callback-based, not `Iterator`) |
| `WalkState` | Control flow: `Continue`, `Skip`, `Quit` |
| `Stack` | Work-stealing deque wrapper |
| `Worker` | Per-thread traversal worker |
| `Work` | Unit of work: entry + ignore rules + device |

## Platform Differences

```
Unix:
  - DirEntryRaw stores: path, type, depth, inode
  - Inode from readdir (cheap)
  - Metadata requires stat call

Windows:
  - DirEntryRaw stores: path, type, depth, full metadata
  - Metadata from directory read (free)
  - No inode concept
```

## Traversal Order Priority

```
1. Glob overrides (whitelist/ignore)
2. Ignore files (.ignore > .gitignore > .git/info/exclude > global)
3. File type matcher
4. Hidden file check
5. File size limit
6. Yielded to caller
```

## Work-Stealing Algorithm

```
Local pop (LIFO) → Steal from neighbors (round-robin)
                          ↓
                   steal_batch_and_pop
                          ↓
                   Depth-first traversal
```

# ripgrep crates/cli/src/lib.rs: Code Companion

Reference code for the CLI Utilities Overview lecture. Sections correspond to the lecture document.

---

## Section 1: The Crate's Documentation-First Design

```
/*!
This crate provides common routines used in command line applications, with a
focus on routines useful for search oriented applications. As a utility
library, there is no central type or function. However, a key focus of this
crate is to improve failure modes and provide user friendly error messages
when things go wrong.

To the best extent possible, everything in this crate works on Windows, macOS
and Linux.


# Standard I/O

[`is_readable_stdin`] determines whether stdin can be usefully read from...

# Coloring and buffering

The [`stdout`], [`stdout_buffered_block`] and [`stdout_buffered_line`] routines
are alternative constructors for [`StandardStream`]...

# Escaping

The [`escape`](crate::escape()), [`escape_os`], [`unescape`] and
[`unescape_os`] routines provide a user friendly way of dealing with UTF-8
encoded strings that can express arbitrary bytes...
*/

#![deny(missing_docs)]  // Compile-time enforcement: all public items must be
documented
```

The `#![deny(missing_docs)]` attribute turns missing documentation into a compile error rather than a warning. The `!` indicates this is an inner attribute applying to the entire crate.

## Section 2: Module Organization and Re-exports

```
// Private module declarations - implementation details hidden from users
mod decompress;
mod escape;
mod hostname;
mod human;
mod pattern;
mod process;
mod wtr;

// Carefully curated public API via re-exports
pub use crate::{
    // Decompression utilities - grouped together
    decompress::{
        DecompressionMatcher, DecompressionMatcherBuilder,
        DecompressionReader, DecompressionReaderBuilder, resolve_binary,
    },
    // Escape sequence handling
    escape::{escape, escape_os, unescape, unescape_os},
    // System hostname retrieval
    hostname::hostname,
    // Human-readable size parsing (e.g., "2M" -> bytes)
    human::{ParseSizeError, parse_human_readable_size},
    // Pattern parsing with good error messages
    pattern::{
        InvalidPatternError, pattern_from_bytes, pattern_from_os,
        patterns_from_path, patterns_from_reader, patterns_from_stdin,
    },
    // Subprocess execution with stderr capture
    process::{CommandError, CommandReader, CommandReaderBuilder},
    // Terminal-aware output streams
    wtr::{
        StandardStream, stdout, stdout_buffered_block, stdout_buffered_line,
    },
};
```

The `pub use crate::module::Item` pattern creates a flat public namespace. Users import `grep_cli::unescape` rather than navigating the internal module structure.

## Section 3: Heuristic Stdin Detection

```rust
/// Returns true if and only if stdin is believed to be readable.
///
/// Note that this isn't perfect and essentially corresponds to a heuristic.
/// When things are unclear (such as if an error occurs during introspection to
/// determine whether stdin is readable), this prefers to return `false`. That
/// means it's possible for an end user to pipe something into your program and
/// have this return `false` and thus potentially lead to ignoring the user's
/// stdin data. While not ideal, this is perhaps better than falsely assuming
/// stdin is readable, which would result in blocking forever on reading stdin.
pub fn is_readable_stdin() -> bool {
    use std::io::IsTerminal;

    // ... platform-specific imp() functions defined here ...

    // Two-step check:
    // 1. If stdin IS a terminal, user is typing - no piped data
    // 2. If stdin is NOT a terminal, check what kind of input it is
    !std::io::stdin().is_terminal() && imp()
}
```

The `IsTerminal` trait (stabilized in Rust 1.70) provides the first filter. The `imp()` function provides platform-specific refinement when stdin is not a terminal.

## Section 4: Platform-Specific Implementation with Conditional Compilation

```rust
#[cfg(unix)]  // Only compiled on Unix-like systems
fn imp() -> bool {
    use std::{
        fs::File,
        os::{fd::AsFd, unix::fs::FileTypeExt},  // Unix-specific extensions
    };

    let stdin = std::io::stdin();
    // Clone the file descriptor to get owned access
    let fd = match stdin.as_fd().try_clone_to_owned() {
        Ok(fd) => fd,
        Err(err) => {
            log::debug!(
                "for heuristic stdin detection on Unix, \
                 could not clone stdin file descriptor \
                 (thus assuming stdin is not readable): {err}",
            );
            return false;  // Defensive: don't crash, just return false
        }
    };

    // Convert file descriptor to File to access metadata
    let file = File::from(fd);
    let md = match file.metadata() {
        Ok(md) => md,
        Err(err) => {
            log::debug!(
                "for heuristic stdin detection on Unix, \
                 could not get file metadata for stdin \
                 (thus assuming stdin is not readable): {err}",
            );
            return false;
        }
    };

    // Check file type using Unix-specific extension trait
    let ft = md.file_type();
    let is_file = ft.is_file();      // Regular file (e.g., < file.txt)
    let is_fifo = ft.is_fifo();      // Named pipe (e.g., mkfifo)
    let is_socket = ft.is_socket();  // Unix socket
    let is_readable = is_file || is_fifo || is_socket;

    // Detailed logging for debugging heuristic decisions
    log::debug!(
        "for heuristic stdin detection on Unix, \
         found that \
         is_file={is_file}, is_fifo={is_fifo} and is_socket={is_socket}, \
         and thus concluded that is_stdin_readable={is_readable}",
    );
    is_readable
}
```

The `FileTypeExt` trait provides Unix-specific methods like `is_fifo()` and `is_socket()` that aren't available in the cross-platform `std::fs::FileType`.

---

## Section 5: The Windows Implementation Contrast

```
#[cfg(windows)]  // Only compiled on Windows
fn imp() -> bool {
    // Windows uses handles, not file descriptors
    let stdin = winapi_util::HandleRef::stdin();
    let typ = match winapi_util::file::typ(stdin) {
        Ok(typ) => typ,
        Err(err) => {
            log::debug!(
                "for heuristic stdin detection on Windows, \
                 could not get file type of stdin \
                 (thus assuming stdin is not readable): {err}",
            );
            return false;
        }
    };

    // Windows file type categories differ from Unix
    let is_disk = typ.is_disk();  // Analogous to Unix is_file()
    let is_pipe = typ.is_pipe();  // Analogous to Unix is_fifo()
    let is_readable = is_disk || is_pipe;

    log::debug!(
        "for heuristic stdin detection on Windows, \
         found that is_disk={is_disk} and is_pipe={is_pipe}, \
         and thus concluded that is_stdin_readable={is_readable}",
    );
    is_readable
}

#[cfg(not(any(unix, windows)))]  // Fallback for other platforms
fn imp() -> bool {
    log::debug!("on non-{{Unix,Windows}}, assuming stdin is not readable");
    false  // Conservative default
}
```

The `winapi_util` crate provides safe Rust wrappers around Windows API calls. The fallback ensures compilation succeeds on any platform Rust supports.

---

## Section 6: Deprecation Patterns and API Evolution

```rust
/// Returns true if and only if stdin is believed to be connected to a tty
/// or a console.
///
/// Note that this is now just a wrapper around
/// [`std::io::IsTerminal`](https://doc.rust-
lang.org/std/io/trait.IsTerminal.html).
/// Callers should prefer using the `IsTerminal` trait directly. This routine
/// is deprecated and will be removed in the next semver incompatible release.
#[deprecated(since = "0.1.10", note = "use std::io::IsTerminal instead")]
pub fn is_tty_stdin() -> bool {
    use std::io::IsTerminal;
    std::io::stdin().is_terminal()  // Delegate to standard library
}

#[deprecated(since = "0.1.10", note = "use std::io::IsTerminal instead")]
pub fn is_tty_stdout() -> bool {
    use std::io::IsTerminal;
    std::io::stdout().is_terminal()
}

#[deprecated(since = "0.1.10", note = "use std::io::IsTerminal instead")]
pub fn is_tty_stderr() -> bool {
    use std::io::IsTerminal;
    std::io::stderr().is_terminal()
}
```

The `#[deprecated]` attribute generates compiler warnings with the specified message. Functions remain functional but delegate to the replacement API, ensuring identical behavior.

---

# Quick Reference

## Public API Overview

| Category | Items | Purpose |
|---|---|---|
| Decompression | `DecompressionReader`, `DecompressionMatcher` | Read compressed files via external tools |
| Escape Handling | `escape`, `unescape`, `escape_os`, `unescape_os` | Convert between literal and escaped byte strings |

| | | |
|---|---|---|
| Pattern Parsing | `pattern_from_bytes`, `patterns_from_path` | Parse regex patterns with good error messages |
| Process I/O | `CommandReader`, `CommandError` | Execute processes with stderr capture |
| Terminal Output | `StandardStream`, `stdout` | TTY-aware buffered colored output |
| Utilities | `hostname`, `parse_human_readable_size` | System info and size parsing |

## Conditional Compilation Attributes

```
#[cfg(unix)]                        // Unix-only code
#[cfg(windows)]                     // Windows-only code
#[cfg(not(any(unix, windows)))]     // Fallback for other platforms
```

## Key Traits Used

| Trait | Source | Purpose |
|---|---|---|
| `IsTerminal` | `std::io` | Check if stream is connected to TTY |
| `AsFd` | `std::os::fd` | Access file descriptor on Unix |
| `FileTypeExt` | `std::os::unix::fs` | Unix-specific file type checks |

## Deprecation Pattern

```
#[deprecated(
    since = "version",      // When deprecation occurred
    note = "migration hint" // How to update code
)]
pub fn old_function() -> T {
    new_function()  // Delegate to replacement
}
```

# Decompression Support: Code Companion

Reference code for the Decompression Support lecture. Sections correspond to the lecture document.

---

## Section 1: The Two-Layer Builder Architecture

```rust
/// A builder for a matcher that determines which files get decompressed.
#[derive(Clone, Debug)]
pub struct DecompressionMatcherBuilder {
    /// The commands for each matching glob.
    commands: Vec<DecompressionCommand>,
    /// Whether to include the default matching rules.
    defaults: bool,
}

/// A representation of a single command for decompressing data
/// out-of-process.
#[derive(Clone, Debug)]
struct DecompressionCommand {
    /// The glob that matches this command.
    glob: String,
    /// The command or binary name - stored as absolute path for security
    bin: PathBuf,
    /// The arguments to invoke with the command.
    args: Vec<OsString>,
}

/// Configures and builds a streaming reader for decompressing data.
#[derive(Clone, Debug, Default)]
pub struct DecompressionReaderBuilder {
    // The compiled matcher - expensive to build, cheap to query
    matcher: DecompressionMatcher,
    // Delegates process spawning details to CommandReaderBuilder
    command_builder: CommandReaderBuilder,
}
```

The two-layer architecture separates concerns: `DecompressionMatcherBuilder` handles glob-to-command mapping, while `DecompressionReaderBuilder` handles process execution. Note that `DecompressionCommand` stores the binary as a `PathBuf` (absolute path) rather than a program name string.

## Section 2: Building the Matcher with Layered Defaults

```
pub fn build(&self) -> Result<DecompressionMatcher, CommandError> {
    // Conditionally load defaults based on configuration flag
    let defaults = if !self.defaults {
        vec![]
    } else {
        default_decompression_commands()
    };

    let mut glob_builder = GlobSetBuilder::new();
    let mut commands = vec![];

    // Chain defaults with user commands - user commands come AFTER,
    // so they take precedence when multiple globs match
    for decomp_cmd in defaults.iter().chain(&self.commands) {
        // Compile each glob pattern
        let glob = Glob::new(&decomp_cmd.glob).map_err(|err| {
            CommandError::io(io::Error::new(io::ErrorKind::Other, err))
        })?;
        glob_builder.add(glob);
        // Build parallel vector - index in globs matches index in commands
        commands.push(decomp_cmd.clone());
    }

    let globs = glob_builder.build().map_err(|err| {
        CommandError::io(io::Error::new(io::ErrorKind::Other, err))
    })?;

    Ok(DecompressionMatcher { globs, commands })
}

/// Toggle whether defaults are included
pub fn defaults(&mut self, yes: bool) -> &mut DecompressionMatcherBuilder {
    self.defaults = yes;
    self
}
```

The `chain()` call ensures user-provided commands are added after defaults. The parallel construction of `globs` and `commands` vectors maintains index correspondence for later lookups.

## Section 3: The Associate Methods and Security Considerations

```rust
/// Silent version - ignores failures (useful for optional tools)
pub fn associate<P, I, A>(
    &mut self,
    glob: &str,
    program: P,
    args: I,
) -> &mut DecompressionMatcherBuilder
where
    P: AsRef<OsStr>,
    I: IntoIterator<Item = A>,
    A: AsRef<OsStr>,
{
    // Deliberately ignore errors - if the binary doesn't exist,
    // the association is simply not added
    let _ = self.try_associate(glob, program, args);
    self
}

/// Explicit error handling version - returns Result
pub fn try_associate<P, I, A>(
    &mut self,
    glob: &str,
    program: P,
    args: I,
) -> Result<&mut DecompressionMatcherBuilder, CommandError>
where
    P: AsRef<OsStr>,
    I: IntoIterator<Item = A>,
    A: AsRef<OsStr>,
{
    let glob = glob.to_string();
    // SECURITY: Resolve to absolute path NOW, not at execution time
    // This prevents Windows CWD search vulnerability
    let bin = try_resolve_binary(Path::new(program.as_ref()))?;
    let args =
        args.into_iter().map(|a| a.as_ref().to_os_string()).collect();
    self.commands.push(DecompressionCommand { glob, bin, args });
    Ok(self)
}
```

The `try_resolve_binary()` call at configuration time is the key security measure. By resolving against PATH early, we store an absolute path that won't trigger Windows' dangerous CWD search behavior when the command is later executed.

---

## Section 4: Command Lookup and the GlobSet Integration

```rust
/// A matcher for determining how to decompress files.
#[derive(Clone, Debug)]
pub struct DecompressionMatcher {
    /// Optimized structure for matching against many globs simultaneously
    globs: GlobSet,
    /// Parallel vector - index corresponds to glob index
    commands: Vec<DecompressionCommand>,
}

impl DecompressionMatcher {
    /// Return a Command if we know how to decompress this file
    pub fn command<P: AsRef<Path>>(&self, path: P) -> Option<Command> {
        // matches() returns ALL matching glob indices
        // next_back() takes the LAST one (most recently added)
        if let Some(i) = self.globs.matches(path).into_iter().next_back() {
            let decomp_cmd = &self.commands[i];
            // Build the Command from our stored configuration
            let mut cmd = Command::new(&decomp_cmd.bin);
            cmd.args(&decomp_cmd.args);
            return Some(cmd);
        }
        None
    }

    /// Cheaper check - just need to know if ANY glob matches
    pub fn has_command<P: AsRef<Path>>(&self, path: P) -> bool {
        // is_match can short-circuit; matches() must find all
        self.globs.is_match(path)
    }
}
```

The `next_back()` call implements the precedence rule. `GlobSet::matches()` returns indices in the order globs were added, so taking the last match gives priority to user customizations over defaults.

---

## Section 5: The DecompressionReaderBuilder and Graceful Degradation

```rust
impl DecompressionReaderBuilder {
    pub fn build<P: AsRef<Path>>(
        &self,
        path: P,
    ) -> Result<DecompressionReader, CommandError> {
        let path = path.as_ref();

        // Path 1: No matching command -> passthru reader
        let Some(mut cmd) = self.matcher.command(path) else {
            return DecompressionReader::new_passthru(path);
        };

        // Add the file path as final argument to the decompression command
        cmd.arg(path);

        match self.command_builder.build(&mut cmd) {
            // Path 2: Command spawned successfully
            Ok(cmd_reader) => Ok(DecompressionReader { rdr: Ok(cmd_reader) }),
            // Path 3: Spawn failed -> log and fallback to passthru
            Err(err) => {
                log::debug!(
                    "{}: error spawning command '{:?}': {} \
                     (falling back to uncompressed reader)",
                    path.display(),
                    cmd,
                    err,
                );
                DecompressionReader::new_passthru(path)
            }
        }
    }

    /// Control async stderr reading to prevent deadlocks
    pub fn async_stderr(
        &mut self,
        yes: bool,
    ) -> &mut DecompressionReaderBuilder {
        self.command_builder.async_stderr(yes);
        self
    }
}
```

The three-path logic ensures ripgrep never fails catastrophically due to missing decompression tools. Errors are logged at debug level, allowing users to diagnose issues while keeping normal output clean.

## Section 6: The DecompressionReader and Unified I/O

```rust
/// A streaming reader for decompressing the contents of a file.
#[derive(Debug)]
pub struct DecompressionReader {
    // Result used as Either: Ok = decompressing, Err = raw file
    rdr: Result<CommandReader, File>,
}

impl DecompressionReader {
    /// Convenience constructor using default matcher
    pub fn new<P: AsRef<Path>>(
        path: P,
    ) -> Result<DecompressionReader, CommandError> {
        DecompressionReaderBuilder::new().build(path)
    }

    /// Create a reader that just passes through file contents
    fn new_passthru(path: &Path) -> Result<DecompressionReader, CommandError> {
        let file = File::open(path)?;
        // Err variant holds the File for passthru mode
        Ok(DecompressionReader { rdr: Err(file) })
    }

    /// Clean up child process resources
    pub fn close(&mut self) -> io::Result<()> {
        match self.rdr {
            Ok(ref mut rdr) => rdr.close(),
            Err(_) => Ok(()),  // File needs no special cleanup
        }
    }
}

/// Unified Read implementation - caller doesn't know which mode is active
impl io::Read for DecompressionReader {
    fn read(&mut self, buf: &mut [u8]) -> io::Result<usize> {
        match self.rdr {
            Ok(ref mut rdr) => rdr.read(buf),   // Read from process stdout
            Err(ref mut rdr) => rdr.read(buf),  // Read from file directly
        }
    }
}
```

The `Result<CommandReader, File>` type is an unconventional use of `Result` as a sum type (like `Either`). Both variants implement `io::Read`, so the `Read` implementation simply delegates to whichever is present.

## Section 7: Binary Resolution and Default Commands

```
fn try_resolve_binary<P: AsRef<Path>>(
    prog: P,
) -> Result<PathBuf, CommandError> {
    use std::env;

    fn is_exe(path: &Path) -> bool {
        let Ok(md) = path.metadata() else { return false };
        !md.is_dir()
    }

    let prog = prog.as_ref();
    // Already absolute? Return as-is
    if prog.is_absolute() {
        return Ok(prog.to_path_buf());
    }

    // Search PATH environment variable
    let Some(syspaths) = env::var_os("PATH") else {
        let msg = "system PATH environment variable not found";
        return Err(CommandError::io(io::Error::new(
            io::ErrorKind::Other,
            msg,
        )));
    };

    for syspath in env::split_paths(&syspaths) {
        if syspath.as_os_str().is_empty() {
            continue;
        }
        let abs_prog = syspath.join(prog);
        if is_exe(&abs_prog) {
            return Ok(abs_prog.to_path_buf());
        }
        // Windows: try .com and .exe extensions
        if abs_prog.extension().is_none() {
            for extension in ["com", "exe"] {
                let abs_prog = abs_prog.with_extension(extension);
                if is_exe(&abs_prog) {
                    return Ok(abs_prog.to_path_buf());
                }
            }
        }
    }

    let msg = format!("{}: could not find executable in PATH", prog.display());
    return Err(CommandError::io(io::Error::new(io::ErrorKind::Other, msg)));
}

fn default_decompression_commands() -> Vec<DecompressionCommand> {
    const ARGS_GZIP: &[&str] = &["gzip", "-d", "-c"];
    const ARGS_BZIP: &[&str] = &["bzip2", "-d", "-c"];
    const ARGS_XZ: &[&str] = &["xz", "-d", "-c"];
    // ... more compression formats ...
```

```
    fn add(glob: &str, args: &[&str], cmds: &mut Vec<DecompressionCommand>) {
// Silently skip if binary not found in PATH        let bin = match
resolve_binary(Path::new(args[0])) {           Ok(bin) => bin,
Err(err) => {               log::debug!("{}", err);               return;  //
Don't add rule for missing tools           }        };
cmds.push(DecompressionCommand {            glob: glob.to_string(),
bin,            args: args.iter().skip(1)  // Skip program name
.map(|s| OsStr::new(s).to_os_string())               .collect(),        });    }
let mut cmds = vec![];    add("*.gz", ARGS_GZIP, &mut cmds);    add("*.tgz",
ARGS_GZIP, &mut cmds);    // ... register all supported formats ...    cmds}
```

The `-d -c` flags are standard across decompression tools: `-d` for decompress, `-c` for output to stdout. The `add()` helper silently skips unavailable tools, ensuring the matcher only contains commands that can actually run.

---

## Quick Reference

### Supported Compression Formats

| Extension | Command | Notes |
|---|---|---|
| `.gz`, `.tgz` | `gzip -d -c` | Most common |
| `.bz2`, `.tbz2` | `bzip2 -d -c` | Higher compression |
| `.xz`, `.txz` | `xz -d -c` | Modern, high ratio |
| `.lz4` | `lz4 -d -c` | Fast decompression |
| `.lzma` | `xz --format=lzma -d -c` | Legacy format |
| `.br` | `brotli -d -c` | Web-optimized |
| `.zst`, `.zstd` | `zstd -q -d -c` | Fast, modern |
| `.Z` | `uncompress -c` | Unix compress |

### Key Types

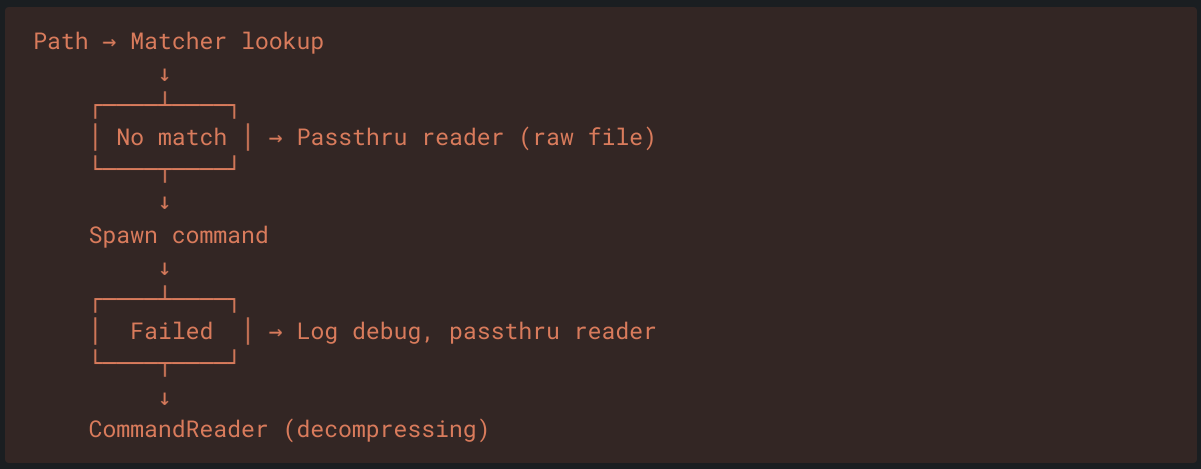
```rust
// Builder for glob-to-command mapping
DecompressionMatcherBuilder::new()
    .defaults(true)                    // Include built-in rules
    .associate("*.custom", "tool", ["-d"])  // Add custom rule
    .build()?                          // -> DecompressionMatcher

// Compiled matcher (reuse for many files)
DecompressionMatcher::command(&path)   // -> Option<Command>
DecompressionMatcher::has_command(&path)  // -> bool

// Reader builder (reuse matcher)
DecompressionReaderBuilder::new()
    .matcher(matcher)
    .async_stderr(true)
    .build(&path)?                     // -> DecompressionReader

// Convenience (builds matcher each time)
DecompressionReader::new(&path)?       // -> DecompressionReader
```

## Graceful Degradation Flow



# ripgrep crates/cli/src/process.rs: Code Companion

Reference code for the Process Management lecture. Sections correspond to the lecture document.

---

## Section 1: The Custom Error Type Strategy

```rust
/// An error that can occur while running a command and reading its output.
///
/// This error can be seamlessly converted to an `io::Error` via a `From`
/// implementation.
#[derive(Debug)]
pub struct CommandError {
    kind: CommandErrorKind,  // Private field - users can't match on variants
}

#[derive(Debug)]
enum CommandErrorKind {
    Io(io::Error),           // System-level I/O failures
    Stderr(Vec<u8>),         // Process stderr output (raw bytes, may not be UTF-
8)
}

impl CommandError {
    /// Create an error from an I/O error.
    pub(crate) fn io(ioerr: io::Error) -> CommandError {
        CommandError { kind: CommandErrorKind::Io(ioerr) }
    }

    /// Create an error from the contents of stderr (which may be empty).
    pub(crate) fn stderr(bytes: Vec<u8>) -> CommandError {
        CommandError { kind: CommandErrorKind::Stderr(bytes) }
    }

    /// Returns true if and only if this error has empty data from stderr.
    pub(crate) fn is_empty(&self) -> bool {
        match self.kind {
            CommandErrorKind::Stderr(ref bytes) => bytes.is_empty(),
            _ => false,
        }
    }
}

impl std::error::Error for CommandError {}
```

The `pub(crate)` constructors limit creation to within the crate while the struct itself is public. The `is_empty` method will be important for distinguishing expected pipe errors from real failures.

## Section 2: Error Display Formatting for Human Readability

```rust
impl std::fmt::Display for CommandError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self.kind {
            // I/O errors delegate to their own Display implementation
            CommandErrorKind::Io(ref e) => e.fmt(f),
            CommandErrorKind::Stderr(ref bytes) => {
                // Lossy conversion handles non-UTF-8 stderr gracefully
                let msg = String::from_utf8_lossy(bytes);
                if msg.trim().is_empty() {
                    // Explicit message when process wrote nothing useful
                    write!(f, "<stderr is empty>")
                } else {
                    // Visual box separates external tool output from ripgrep
messages
                    let div = "-".repeat(79);
                    write!(
                        f,
                        "\n{div}\n{msg}\n{div}",
                        div = div,
                        msg = msg.trim()  // Clean up whitespace for readability
                    )
                }
            }
        }
    }
}
```

The `from_utf8_lossy` function replaces invalid UTF-8 sequences with the Unicode replacement character, ensuring the error message is always displayable even if the external tool wrote binary data to stderr.

---

## Section 3: Bidirectional Error Conversion

```rust
// Converting io::Error → CommandError (wrapping)
impl From<io::Error> for CommandError {
    fn from(ioerr: io::Error) -> CommandError {
        CommandError { kind: CommandErrorKind::Io(ioerr) }
    }
}

// Converting CommandError → io::Error (unwrapping or wrapping)
impl From<CommandError> for io::Error {
    fn from(cmderr: CommandError) -> io::Error {
        match cmderr.kind {
            // I/O errors unwrap cleanly - no information lost
            CommandErrorKind::Io(ioerr) => ioerr,
            // Stderr errors become "Other" with the CommandError as source
            CommandErrorKind::Stderr(_) => {
                io::Error::new(io::ErrorKind::Other, cmderr)
            }
        }
    }
}
```

These implementations enable the `?` operator to work seamlessly. A function returning `Result<T, io::Error>` can call functions returning `Result<T, CommandError>` and vice versa.

---

## Section 4: The Builder Pattern for Command Configuration

```rust
/// Configures and builds a streaming reader for process output.
#[derive(Clone, Debug, Default)]
pub struct CommandReaderBuilder {
    async_stderr: bool,
}

impl CommandReaderBuilder {
    /// Create a new builder with the default configuration.
    pub fn new() -> CommandReaderBuilder {
        CommandReaderBuilder::default()
    }

    /// Build a new streaming reader for the given command's output.
    pub fn build(
        &self,
        command: &mut process::Command,
    ) -> Result<CommandReader, CommandError> {
        // Override stdout/stderr to piped - we need to capture both
        let mut child = command
            .stdout(process::Stdio::piped())
            .stderr(process::Stdio::piped())
            .spawn()?;  // ? converts io::Error to CommandError via From

        // Take ownership of stderr from the child process
        let stderr = if self.async_stderr {
            StderrReader::r#async(child.stderr.take().unwrap())
        } else {
            StderrReader::sync(child.stderr.take().unwrap())
        };
        Ok(CommandReader { child, stderr, eof: false })
    }
}
```

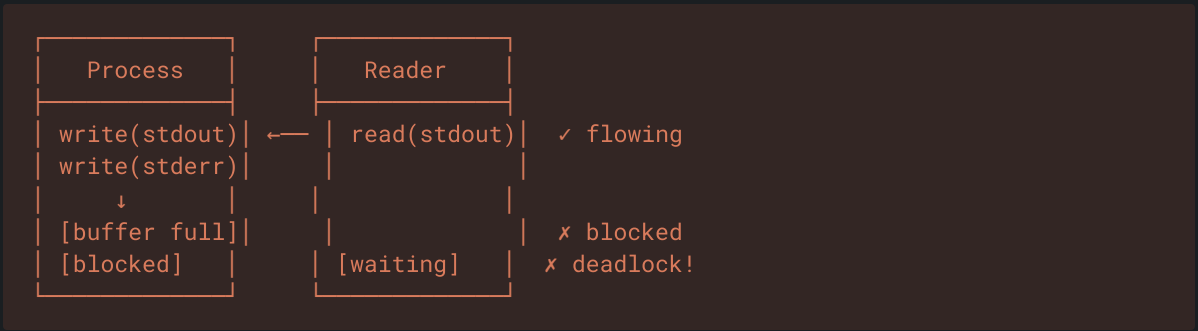
The `take()` calls return `Option<ChildStderr>`, removing the handle from the child and giving ownership to `StderrReader`. The `unwrap()` is safe because we just set stderr to `piped()`.

---

## Section 5: Understanding the Deadlock Problem

```rust
impl CommandReaderBuilder {
    /// When enabled, the reader will asynchronously read the contents of the
    /// command's stderr output. When disabled, stderr is only read after the
    /// stdout stream has been exhausted (or if the process quits with an error
    /// code).
    ///
    /// Note that when enabled, this may require launching an additional
    /// thread in order to read stderr. This is done so that the process being
    /// executed is never blocked from writing to stdout or stderr. If this is
    /// disabled, then it is possible for the process to fill up the stderr
    /// buffer and deadlock.
    ///
    /// This is enabled by default.
    pub fn async_stderr(&mut self, yes: bool) -> &mut CommandReaderBuilder {
        self.async_stderr = yes;
        self
    }
}
```

The deadlock scenario:




## Section 6: The CommandReader Core Structure

```rust
/// A streaming reader for a command's output.
#[derive(Debug)]
pub struct CommandReader {
    child: process::Child,     // Owns the spawned process (including stdout)
    stderr: StderrReader,      // Handles stderr via sync or async reading
    /// This is set to true once 'read' returns zero bytes. When this isn't
    /// set and we close the reader, then we anticipate a pipe error when
    /// reaping the child process and silence it.
    eof: bool,
}

impl CommandReader {
    /// Create a new streaming reader for the given command using the default
    /// configuration.
    pub fn new(
        cmd: &mut process::Command,
    ) -> Result<CommandReader, CommandError> {
        CommandReaderBuilder::new().build(cmd)
    }
}
```

The `Child` struct from `std::process` contains `stdout: Option<ChildStdout>`. The `Option` allows taking ownership of stdout later (via `take()`), which is how the close protocol works.

---

## Section 7: The Close Protocol

```
impl CommandReader {
    /// Closes the CommandReader, freeing any resources used by its underlying
    /// child process. If the child process exits with a nonzero exit code, the
    /// returned Err value will include its stderr.
    pub fn close(&mut self) -> io::Result<()> {
        // Step 1: Take stdout (if already taken, close() was called before)
        let stdout = match self.child.stdout.take() {
            None => return Ok(()),  // Idempotent: already closed
            Some(stdout) => stdout,
        };

        // Step 2: Drop stdout to signal EOF to child process
        drop(stdout);

        // Step 3: Wait for child and check exit status
        if self.child.wait()?.success() {
            Ok(())
        } else {
            // Step 4: On failure, collect stderr for error message
            let err = self.stderr.read_to_end();

            // Step 5: Handle the early-close case
            // If we didn't read to EOF and stderr is empty,
            // the "failure" is just a broken pipe - expected behavior
            if !self.eof && err.is_empty() {
                return Ok(());
            }
            Err(io::Error::from(err))
        }
    }
}
```

The `drop(stdout)` closes the pipe, which causes the child process to receive EOF on its output. Well-behaved processes will exit cleanly when this happens.

## Section 8: The Drop Safety Net

```
impl Drop for CommandReader {
    fn drop(&mut self) {
        if let Err(error) = self.close() {
            // Log warning instead of panicking - Drop should never panic
            log::warn!("{}", error);
        }
    }
}
```

The `Drop` implementation ensures resource cleanup even if the caller forgets to call `close()`. Using `log::warn!` instead of panicking follows Rust's guideline that `Drop` implementations should not panic.

---

## Section 9: The Read Implementation

```rust
impl io::Read for CommandReader {
    fn read(&mut self, buf: &mut [u8]) -> io::Result<usize> {
        // Handle already-closed reader
        let stdout = match self.child.stdout {
            None => return Ok(0),  // Signals EOF to caller
            Some(ref mut stdout) => stdout,
        };

        // Delegate to the underlying stdout pipe
        let nread = stdout.read(buf)?;

        if nread == 0 {
            // Reached natural EOF - set flag and trigger cleanup
            self.eof = true;
            self.close().map(|_| 0)  // Return 0 to signal EOF
        } else {
            Ok(nread)
        }
    }
}
```

The `ref mut` pattern borrows stdout mutably without taking ownership. This is necessary because we need to read from it while keeping it inside the `Option` for later cleanup.

---

## Section 10: Async vs Sync Stderr Handling

```rust
/// A reader that encapsulates the asynchronous or synchronous reading of
/// stderr.
#[derive(Debug)]
enum StderrReader {
    Async(Option<std::thread::JoinHandle<CommandError>>),
    Sync(process::ChildStderr),
}

impl StderrReader {
    /// Create a reader for stderr that reads contents asynchronously.
    fn r#async(mut stderr: process::ChildStderr) -> StderrReader {
        // Spawn thread that reads stderr and converts to CommandError
        let handle =
            std::thread::spawn(move || stderr_to_command_error(&mut stderr));
        StderrReader::Async(Some(handle))
    }

    /// Create a reader for stderr that reads contents synchronously.
    fn sync(stderr: process::ChildStderr) -> StderrReader {
        StderrReader::Sync(stderr)
    }

    /// Consumes all of stderr on to the heap and returns it as an error.
    fn read_to_end(&mut self) -> CommandError {
        match *self {
            StderrReader::Async(ref mut handle) => {
                let handle = handle
                    .take()
                    .expect("read_to_end cannot be called more than once");
                // join() blocks until thread completes, returns its result
                handle.join().expect("stderr reading thread does not panic")
            }
            StderrReader::Sync(ref mut stderr) => {
                stderr_to_command_error(stderr)
            }
        }
    }
}

fn stderr_to_command_error(stderr: &mut process::ChildStderr) -> CommandError {
    let mut bytes = vec![];
    match stderr.read_to_end(&mut bytes) {
        Ok(_) => CommandError::stderr(bytes),
        Err(err) => CommandError::io(err),
    }
}
```

The `r#async` syntax is a raw identifier, necessary because `async` is a reserved keyword in Rust. The `move` keyword in the closure transfers ownership of `stderr` to the spawned thread.

# Quick Reference

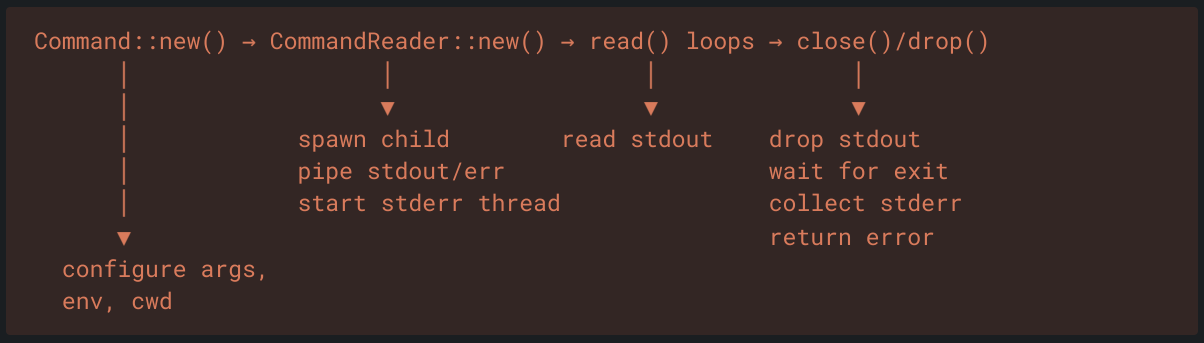
## Type Summary

| Type | Purpose |
|---|---|
| `CommandError` | Error type capturing I/O errors or stderr content |
| `CommandReaderBuilder` | Configures process reading behavior |
| `CommandReader` | Streaming reader implementing `Read` trait |
| `StderrReader` | Internal enum for sync/async stderr handling |

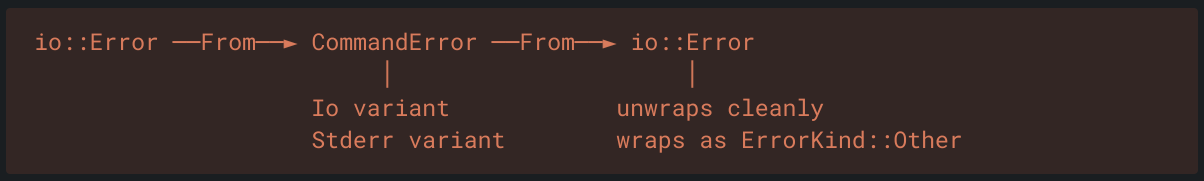
## Key Methods

| Method | Returns | Purpose |
|---|---|---|
| `CommandReader::new()` | `Result<CommandReader, CommandError>` | Spawn process with defaults |
| `CommandReader::close()` | `io::Result<()>` | Clean up process, return stderr on failure |
| `CommandReaderBuilder::build()` | `Result<CommandReader, CommandError>` | Spawn with custom config |
| `CommandReaderBuilder::async_stderr()` | `&mut Self` | Enable/disable async stderr reading |

## Process Lifecycle




## Error Conversion Flow



# ripgrep crates/core/main.rs: Code Companion

Reference code for the Application Entry Point lecture. Sections correspond to the lecture document.

## Section 1: The Allocator Override

```
// Since Rust no longer uses jemalloc by default, ripgrep will, by default,
// use the system allocator. On Linux, this would normally be glibc's
// allocator, which is pretty good. In particular, ripgrep does not have a
// particularly allocation heavy workload, so there really isn't much
// difference (for ripgrep's purposes) between glibc's allocator and jemalloc.
//
// However, when ripgrep is built with musl, this means ripgrep will use musl's
// allocator, which appears to be substantially worse. (musl's goal is not to
// have the fastest version of everything. Its goal is to be small and amenable
// to static compilation.) Even though ripgrep isn't particularly allocation
// heavy, musl's allocator appears to slow down ripgrep quite a bit. Therefore,
// when building with musl, we use jemalloc.
//
// We don't unconditionally use jemalloc because it can be nice to use the
// system's default allocator by default. Moreover, jemalloc seems to increase
// compilation times by a bit.
//
// Moreover, we only do this on 64-bit systems since jemalloc doesn't support
// i686.

// Conditional compilation: only include this code for musl + 64-bit targets
#[cfg(all(target_env = "musl", target_pointer_width = "64"))]
#[global_allocator]  // Replaces the default allocator for ALL heap allocations
static ALLOC: tikv_jemallocator::Jemalloc = tikv_jemallocator::Jemalloc;
```

The `#[cfg(...)]` attribute uses boolean logic: `all()` requires both conditions. The `#[global_allocator]` attribute can only appear once per binary and completely replaces Rust's default allocation strategy.

## Section 2: The Main Function and Exit Code Philosophy

```
use std::{io::Write, process::ExitCode};

/// Then, as it was, then again it will be.
fn main() -> ExitCode {
    match run(flags::parse()) {
        Ok(code) => code,
        Err(err) => {
            // Look for a broken pipe error. In this case, we generally want
            // to exit "gracefully" with a success exit code. This matches
            // existing Unix convention.
            for cause in err.chain() {
                // Walk the error chain looking for an io::Error
                if let Some(ioerr) = cause.downcast_ref::<std::io::Error>() {
                    if ioerr.kind() == std::io::ErrorKind::BrokenPipe {
                        return ExitCode::from(0);  // Graceful exit on pipe close
                    }
                }
            }
            // {:#} prints the full error chain with "caused by:" formatting
            eprintln_locked!("{:#}", err);
            ExitCode::from(2)  // Error exit code
        }
    }
}
```

The `err.chain()` method comes from the `anyhow` crate and iterates through the entire chain of wrapped errors. The `downcast_ref::<T>()` method attempts to cast a trait object back to a concrete type, returning `Some(&T)` on success.

## Section 3: The Run Function and Mode Dispatch

```
/// The main entry point for ripgrep.
fn run(result: crate::flags::ParseResult<HiArgs>) -> anyhow::Result<ExitCode> {
    use crate::flags::{Mode, ParseResult};

    // Three-way match on parse result: error, special mode, or success
    let args = match result {
        ParseResult::Err(err) => return Err(err),
        ParseResult::Special(mode) => return special(mode),  // Help, version
        ParseResult::Ok(args) => args,
    };

    // Mode dispatch: determines what ripgrep actually does
    let matched = match args.mode() {
        // Short-circuit if pattern can never match anything
        Mode::Search(_) if !args.matches_possible() => false,
        // Single-threaded search (threads == 1)
        Mode::Search(mode) if args.threads() == 1 => search(&args, mode)?,
        // Multi-threaded search (default)
        Mode::Search(mode) => search_parallel(&args, mode)?,
        // File listing modes
        Mode::Files if args.threads() == 1 => files(&args)?,
        Mode::Files => files_parallel(&args)?,
        // Utility modes return directly
        Mode::Types => return types(&args),
        Mode::Generate(mode) => return generate(mode),
    };

    // Exit code calculation: 0 = match found, 1 = no match, 2 = error
    Ok(if matched && (args.quiet() || !messages::errored()) {
        ExitCode::from(0)
    } else if messages::errored() {
        ExitCode::from(2)
    } else {
        ExitCode::from(1)
    })
}
```

The match guard syntax (`if args.threads() == 1`) allows conditional branching within pattern arms. The `messages::errored()` function provides global error state tracking without threading error counts through every function.

## Section 4: Single-Threaded Search Architecture

```rust
/// The top-level entry point for single-threaded search.
fn search(args: &HiArgs, mode: SearchMode) -> anyhow::Result<bool> {
    let started_at = std::time::Instant::now();

    // Build the search infrastructure using HiArgs builder methods
    let haystack_builder = args.haystack_builder();
    let unsorted = args
        .walk_builder()?
        .build()
        .filter_map(|result| haystack_builder.build_from_result(result));
    let haystacks = args.sort(unsorted);  // Optional sorting

    let mut matched = false;
    let mut searched = false;
    let mut stats = args.stats();  // Option<Stats> - None if not requested

    // Compose the search worker from matcher, searcher, and printer
    let mut searcher = args.search_worker(
        args.matcher()?,
        args.searcher()?,
        args.printer(mode, args.stdout()),
    )?;

    for haystack in haystacks {
        searched = true;
        let search_result = match searcher.search(&haystack) {
            Ok(search_result) => search_result,
            // Broken pipe means graceful termination
            Err(err) if err.kind() == std::io::ErrorKind::BrokenPipe => break,
            Err(err) => {
                err_message!("{}: {}", haystack.path().display(), err);
                continue;  // Log error but keep searching other files
            }
        };
        matched = matched || search_result.has_match();

        // Accumulate stats only if requested (Option pattern)
        if let Some(ref mut stats) = stats {
            *stats += search_result.stats().unwrap();
        }

        // Support --quiet: stop after first match
        if matched && args.quit_after_match() {
            break;
        }
    }

    // Helpful error when ignore rules filter everything
    if args.has_implicit_path() && !searched {
        eprint_nothing_searched();
    }

    // Print stats if requested
    if let Some(ref stats) = stats {
```

```
        let wtr = searcher.printer().get_mut();        let _ = print_stats(mode,
 stats, started_at, wtr);    }    Ok(matched)}
```

The `filter_map` combinator transforms and filters in one step—`None` values are discarded. The `if let Some(ref mut stats)` pattern provides zero-cost optional feature handling.

---

## Section 5: Parallel Search and Thread Coordination

```
/// The top-level entry point for multi-threaded search.
fn search_parallel(args: &HiArgs, mode: SearchMode) -> anyhow::Result<bool> {
    use std::sync::atomic::{AtomicBool, Ordering};

    let started_at = std::time::Instant::now();
    let haystack_builder = args.haystack_builder();
    let bufwtr = args.buffer_writer();  // Thread-safe buffered writer

    // Shared state wrapped in appropriate synchronization primitives
    let stats = args.stats().map(std::sync::Mutex::new);
    let matched = AtomicBool::new(false);
    let searched = AtomicBool::new(false);

    // Create a prototype searcher that will be cloned per-thread
    let mut searcher = args.search_worker(
        args.matcher()?,
        args.searcher()?,
        args.printer(mode, bufwtr.buffer()),
    )?;

    // Parallel directory walk with worker closure
    args.walk_builder()?.build_parallel().run(|| {
        // Capture shared state by reference
        let bufwtr = &bufwtr;
        let stats = &stats;
        let matched = &matched;
        let searched = &searched;
        let haystack_builder = &haystack_builder;
        let mut searcher = searcher.clone();  // Each thread gets its own clone

        // Return a boxed closure that processes one directory entry
        Box::new(move |result| {
            let haystack = match haystack_builder.build_from_result(result) {
                Some(haystack) => haystack,
                None => return WalkState::Continue,
            };

            searched.store(true, Ordering::SeqCst);
            searcher.printer().get_mut().clear();  // Clear buffer for reuse

            let search_result = match searcher.search(&haystack) {
                Ok(search_result) => search_result,
                Err(err) => {
                    err_message!("{}: {}", haystack.path().display(), err);
                    return WalkState::Continue;
                }
            };

            if search_result.has_match() {
                matched.store(true, Ordering::SeqCst);
            }

            // Stats protected by mutex for thread-safe accumulation
            if let Some(ref locked_stats) = *stats {
```

```
                let mut stats = locked_stats.lock().unwrap();
*stats += search_result.stats().unwrap();            }            // Print
through buffer writer (handles synchronization)            if let Err(err) =
bufwtr.print(searcher.printer().get_mut()) {                if err.kind() ==
std::io::ErrorKind::BrokenPipe {                    return WalkState::Quit;  //
Stop all threads                }                err_message!("{}: {}",
haystack.path().display(), err);            }            // Check if we should
stop (--quiet mode)            if matched.load(Ordering::SeqCst) &&
args.quit_after_match() {                WalkState::Quit            } else {
WalkState::Continue            }        })    });    // Post-processing (runs
after all threads complete)    if args.has_implicit_path() &&
!searched.load(Ordering::SeqCst) {        eprint_nothing_searched();    }    if
let Some(ref locked_stats) = stats {        let stats =
locked_stats.lock().unwrap();        let mut wtr = searcher.printer().get_mut();
let _ = print_stats(mode, &stats, started_at, &mut wtr);        let _ =
bufwtr.print(&mut wtr);    }    Ok(matched.load(Ordering::SeqCst))}
```

The `WalkState` enum controls parallel traversal: `Continue` proceeds normally, `Quit` signals all threads to stop. `SeqCst` ordering provides the strongest memory ordering guarantees, ensuring visibility across all threads.

---

## Section 6: File Listing Modes

```
/// Single-threaded file listing
fn files(args: &HiArgs) -> anyhow::Result<bool> {
    let haystack_builder = args.haystack_builder();
    let unsorted = args
        .walk_builder()?
        .build()
        .filter_map(|result| haystack_builder.build_from_result(result));
    let haystacks = args.sort(unsorted);

    let mut matched = false;
    let mut path_printer = args.path_printer_builder().build(args.stdout());

    for haystack in haystacks {
        matched = true;
        if args.quit_after_match() {
            break;
        }
        if let Err(err) = path_printer.write(haystack.path()) {
            if err.kind() == std::io::ErrorKind::BrokenPipe {
                break;
            }
            return Err(err.into());
        }
    }
    Ok(matched)
}

/// Multi-threaded file listing with channel-based output
fn files_parallel(args: &HiArgs) -> anyhow::Result<bool> {
    use std::{
        sync::{atomic::{AtomicBool, Ordering}, mpsc},
        thread,
    };

    let haystack_builder = args.haystack_builder();
    let mut path_printer = args.path_printer_builder().build(args.stdout());
    let matched = AtomicBool::new(false);

    // Channel for sending paths from workers to printer thread
    let (tx, rx) = mpsc::channel::<crate::haystack::Haystack>();

    // Single printing thread prevents write tearing
    let print_thread = thread::spawn(move || -> std::io::Result<()> {
        for haystack in rx.iter() {
            path_printer.write(haystack.path())?;
        }
        Ok(())
    });

    args.walk_builder()?.build_parallel().run(|| {
        let haystack_builder = &haystack_builder;
        let matched = &matched;
        let tx = tx.clone();  // Clone sender for each worker
```

```
        Box::new(move |result| {            let haystack = match
haystack_builder.build_from_result(result) {                Some(haystack) =>
haystack,                None => return WalkState::Continue,            };
matched.store(true, Ordering::SeqCst);            if args.quit_after_match() {
WalkState::Quit            } else {                match tx.send(haystack) {
Ok(_) => WalkState::Continue,                    Err(_) => WalkState::Quit,  //
Receiver dropped                }            }        })    });    drop(tx);  //
Close channel, allowing print_thread to finish    if let Err(err) =
print_thread.join().unwrap() {        if err.kind() !=
std::io::ErrorKind::BrokenPipe {            return Err(err.into());        }    }
Ok(matched.load(Ordering::SeqCst))}
```

The `drop(tx)` is critical: it closes the sending end of the channel, allowing the receiver's `iter()` to terminate. Without this, the print thread would block forever waiting for more messages.

---

## Section 7: Utility Modes and Statistics

```
/// The top-level entry point for `--type-list`.
fn types(args: &HiArgs) -> anyhow::Result<ExitCode> {
    let mut count = 0;
    let mut stdout = args.stdout();
    for def in args.types().definitions() {
        count += 1;
        stdout.write_all(def.name().as_bytes())?;
        stdout.write_all(b": ")?;

        let mut first = true;
        for glob in def.globs() {
            if !first {
                stdout.write_all(b", ")?;
            }
            stdout.write_all(glob.as_bytes())?;
            first = false;
        }
        stdout.write_all(b"\n")?;
    }
    // Exit 1 if no types defined, 0 otherwise
    Ok(ExitCode::from(if count == 0 { 1 } else { 0 }))
}

/// Generate shell completions and man pages
fn generate(mode: crate::flags::GenerateMode) -> anyhow::Result<ExitCode> {
    use crate::flags::GenerateMode;

    let output = match mode {
        GenerateMode::Man => flags::generate_man_page(),
        GenerateMode::CompleteBash => flags::generate_complete_bash(),
        GenerateMode::CompleteZsh => flags::generate_complete_zsh(),
        GenerateMode::CompleteFish => flags::generate_complete_fish(),
        GenerateMode::CompletePowerShell =>
flags::generate_complete_powershell(),
    };
    writeln!(std::io::stdout(), "{}", output.trim_end())?;
    Ok(ExitCode::from(0))
}

/// Print statistics in text or JSON format
fn print_stats<W: Write>(
    mode: SearchMode,
    stats: &grep::printer::Stats,
    started: std::time::Instant,
    mut wtr: W,
) -> std::io::Result<()> {
    let elapsed = std::time::Instant::now().duration_since(started);

    if matches!(mode, SearchMode::JSON) {
        serde_json::to_writer(
            &mut wtr,
            &serde_json::json!({
                "type": "summary",
                "data": {
```

```
                    "stats": stats,                    "elapsed_total": {
"secs": elapsed.as_secs(),                        "nanos":
elapsed.subsec_nanos(),                       "human": format!("{:0.6}s",
elapsed.as_secs_f64()),                    },                }            }),
)?;        write!(wtr, "\n")    } else {        write!(wtr, "{matches}
matches{lines} matched lines{searches_with_match} files contained
matches{searches} files searched{bytes_printed} bytes printed{bytes_searched}
bytes searched{search_time:0.6} seconds spent searching{process_time:0.6} seconds
total",            matches = stats.matches(),            lines =
stats.matched_lines(),            // ... remaining fields        )    }}
```

The `matches!` macro is a concise way to test if a value matches a pattern without binding. The generic `W: Write` bound allows the same function to write to any destination implementing `Write`.

---

## Quick Reference

### Exit Codes

| Code | Meaning |
|---|---|
| 0 | Matches found (or broken pipe) |
| 1 | No matches found |
| 2 | Error occurred |

### Mode Dispatch

| Mode | Single-Threaded | Multi-Threaded |
|---|---|---|
| Search | `search()` | `search_parallel()` |
| Files | `files()` | `files_parallel()` |
| Types | `types()` | N/A |

| | | |
|---|---|---|
| Generate | `generate()` | N/A |

## Thread Synchronization Primitives

```
// Atomic boolean for simple flags
use std::sync::atomic::{AtomicBool, Ordering};
let flag = AtomicBool::new(false);
flag.store(true, Ordering::SeqCst);
flag.load(Ordering::SeqCst);

// Mutex for complex data requiring exclusive access
use std::sync::Mutex;
let data = Mutex::new(Stats::new());
let mut guard = data.lock().unwrap();
*guard += new_stats;

// Channel for producer-consumer patterns
use std::sync::mpsc;
let (tx, rx) = mpsc::channel();
tx.send(item)?;
for item in rx.iter() { /* ... */ }
```

## WalkState Control Flow

```
use ignore::WalkState;

// In parallel walk closure:
WalkState::Continue  // Keep processing files
WalkState::Quit      // Signal all threads to stop
WalkState::Skip      // Skip current directory (not used here)
```

# ripgrep crates/core/flags/mod.rs: Code Companion

Reference code for the Flag System Architecture lecture. Sections correspond to the lecture document.

---

## Section 1: The Module's Public Contract

```rust
/*!
Defines ripgrep's command line interface.

This modules deals with everything involving ripgrep's flags and positional
arguments. This includes generating shell completions, `--help` output and even
ripgrep's man page. It's also responsible for parsing and validating every
flag (including reading ripgrep's config file), and manages the contact points
between these flags and ripgrep's cast of supporting libraries.
*/
use std::{
    ffi::OsString,
    fmt::Debug,
    panic::{RefUnwindSafe, UnwindSafe},
};

// Re-exports organized by purpose: completions, documentation, and parsing
pub(crate) use crate::flags::{
    // Shell completion generators for different shells
    complete::{
        bash::generate as generate_complete_bash,
        fish::generate as generate_complete_fish,
        powershell::generate as generate_complete_powershell,
        zsh::generate as generate_complete_zsh,
    },
    // Documentation generators for help text, man pages, and version info
    doc::{
        help::{
            generate_long as generate_help_long,
            generate_short as generate_help_short,
        },
        man::generate as generate_man_page,
        version::{
            generate_long as generate_version_long,
            generate_pcre2 as generate_version_pcre2,
            generate_short as generate_version_short,
        },
    },
    // High-level parsed arguments - what the rest of ripgrep uses
    hiargs::HiArgs,
    // Low-level argument types for different operational modes
    lowargs::{GenerateMode, Mode, SearchMode, SpecialMode},
    // The parsing infrastructure
    parse::{ParseResult, parse},
};

// Submodule declarations - each handles a single responsibility
mod complete;   // Shell completion generation
mod config;     // Configuration file reading
mod defs;       // Actual flag definitions
mod doc;        // Documentation generation
mod hiargs;     // High-level argument representation
mod lowargs;    // Low-level argument representation
mod parse;      // Command-line parsing
```

The `pub(crate)` visibility makes these types available throughout the `core` crate while hiding them from external consumers. The re-exports with renamed identifiers (e.g., `generate as generate_complete_bash`) create a clean, descriptive API.

---

## Section 2: The Flag Trait as Architectural Contract

```rust
/// A trait that encapsulates the definition of an optional flag for ripgrep.
///
/// This trait is meant to be used via dynamic dispatch. Namely, the `defs`
/// module provides a single global slice of `&dyn Flag` values correspondings
/// to all of the flags in ripgrep.
trait Flag: Debug + Send + Sync + UnwindSafe + RefUnwindSafe + 'static {
    /// Returns true if this flag is a switch. When a flag is a switch, the
    /// CLI parser will not look for a value after the flag is seen.
    fn is_switch(&self) -> bool;

    // ... name methods ...

    /// Returns the category of this flag.
    fn doc_category(&self) -> Category;

    // ... documentation methods ...

    /// Given the parsed value (which might just be a switch), this should
    /// update the state in `args` based on the value given for this flag.
    ///
    /// By convention, implementations should generally not try to "do"
    /// anything other than validate the value given. For example, the
    /// implementation for `--hostname-bin` should not try to resolve the
    /// hostname to use by running the binary provided.
    fn update(
        &self,
        value: FlagValue,
        args: &mut crate::flags::lowargs::LowArgs,
    ) -> anyhow::Result<()>;
}
```

The trait bounds tell an important story: `Debug` enables introspection, `Send + Sync` allows potential parallel processing, `UnwindSafe + RefUnwindSafe` ensure panic safety, and `'static` guarantees flag definitions outlive any references to them.

---

## Section 3: Name Resolution and Aliases

```
trait Flag: /* bounds */ {
    /// A short single byte name for this flag. This returns `None` by default,
    /// which signifies that the flag has no short name.
    ///
    /// The byte returned must be an ASCII codepoint that is a `.` or is
    /// alpha-numeric.
    fn name_short(&self) -> Option<u8> {
        None  // Default: no short name
    }

    /// Returns the long name of this flag. All flags must have a "long" name.
    ///
    /// The long name must be at least 2 bytes, and all of its bytes must be
    /// ASCII codepoints that are either `-` or alpha-numeric.
    fn name_long(&self) -> &'static str;

    /// Returns a list of aliases for this flag.
    ///
    /// By default, an empty slice is returned.
    fn aliases(&self) -> &'static [&'static str] {
        &[]  // Default: no aliases
    }

    /// Returns a negated name for this flag. The negation of a flag is
    /// intended to have the opposite meaning of a flag or to otherwise turn
    /// something "off" or revert it to its default behavior.
    ///
    /// Negated flags are not listed in their own section in the `-h/--help`
    /// output or man page. Instead, they are automatically mentioned at the
    /// end of the documentation section of the flag they negated.
    fn name_negated(&self) -> Option<&'static str> {
        None  // Default: no negation
    }
}
```

Note that `name_long` has no default implementation—it's the only required naming method. The constraint that short names be `u8` (not `char`) enforces ASCII-only short flags, which is a deliberate design choice for portability.

## Section 4: Documentation as First-Class Data

```
trait Flag: /* bounds */ {
    /// Returns the variable name describing the type of value this flag
    /// accepts. This should always be set for non-switch flags and never set
    /// for switch flags.
    ///
    /// For example, the `--max-count` flag has its variable name set to `NUM`.
    ///
    /// The convention is to capitalize variable names.
    fn doc_variable(&self) -> Option<&'static str> {
        None
    }

    /// Returns the category of this flag.
    ///
    /// Every flag must have a single category. Categories are used to organize
    /// flags in the generated documentation.
    fn doc_category(&self) -> Category;

    /// A (very) short documentation string describing what this flag does.
    ///
    /// This may sacrifice "proper English" in order to be as terse as
    /// possible. Generally, we try to ensure that `rg -h` doesn't have any
    /// lines that exceed 79 columns.
    fn doc_short(&self) -> &'static str;

    /// A (possibly very) longer documentation string describing in full
    /// detail what this flag does. This should be in mandoc/mdoc format.
    fn doc_long(&self) -> &'static str;

    /// If this is a non-switch flag that accepts a small set of specific
    /// values, then this should list them.
    fn doc_choices(&self) -> &'static [&'static str] {
        &[]
    }
}
```

The `'static` lifetime on all return types ensures documentation strings are compile-time constants, enabling zero-cost access during help generation. The mandoc format requirement for `doc_long` enables direct man page generation.

## Section 5: The Category System for Organization

```rust
/// The category that a flag belongs to.
///
/// Categories are used to organize flags into "logical" groups in the
/// generated documentation.
#[derive(Clone, Copy, Debug, Eq, Hash, PartialEq, PartialOrd, Ord)]
enum Category {
    /// Flags related to how ripgrep reads its input.
    Input,
    /// Flags related to the operation of the search itself.
    Search,
    /// Flags related to how ripgrep filters haystacks.
    Filter,
    /// Flags related to how ripgrep shows its search results.
    Output,
    /// Flags related to changing ripgrep's output at a more fundamental level.
    OutputModes,
    /// Flags related to logging behavior.
    Logging,
    /// Other behaviors not related to ripgrep's core functionality.
    OtherBehaviors,
}

impl Category {
    /// Returns a string representation of this category.
    ///
    /// This string is the name of the variable used in various templates for
    /// generated documentation. This name can be used for interpolation.
    fn as_str(&self) -> &'static str {
        match *self {
            Category::Input => "input",
            Category::Search => "search",
            Category::Filter => "filter",
            Category::Output => "output",
            Category::OutputModes => "output-modes",
            Category::Logging => "logging",
            Category::OtherBehaviors => "other-behaviors",
        }
    }
}
```

The `Ord` and `PartialOrd` derives use the enum variant declaration order for comparison, making `Input` sort before `Search`, which sorts before `Filter`, and so on. This determines presentation order in documentation.

---

## Section 6: Shell Completion Support

```rust
/// The kind of argument a flag accepts, to be used for shell completions.
#[derive(Clone, Copy, Debug)]
enum CompletionType {
    /// No special category. is_switch() and doc_choices() may apply.
    Other,
    /// A path to a file.
    Filename,
    /// A command in $PATH.
    Executable,
    /// The name of a file type, as used by e.g. --type.
    Filetype,
    /// The name of an encoding_rs encoding, as used by --encoding.
    Encoding,
}

// In the Flag trait:
trait Flag: /* bounds */ {
    fn completion_type(&self) -> CompletionType {
        CompletionType::Other  // Default for most flags
    }
}
```

Each variant maps to shell-specific completion behavior. For example, `Filename` triggers file path completion, while `Executable` searches `$PATH`. The default `Other` falls back to `doc_choices()` if available.

---

## Section 7: The FlagValue Type

```
/// Represents a value parsed from the command line.
///
/// This doesn't include the corresponding flag, but values come in one of
/// two forms: a switch (on or off) or an arbitrary value.
#[derive(Debug)]
enum FlagValue {
    /// A flag that is either on or off.
    Switch(bool),
    /// A flag that comes with an arbitrary user value.
    Value(OsString),
}

impl FlagValue {
    /// Return the yes or no value of this switch.
    ///
    /// If this flag value is not a switch, then this panics.
    fn unwrap_switch(self) -> bool {
        match self {
            FlagValue::Switch(yes) => yes,
            FlagValue::Value(_) => {
                unreachable!("got flag value but expected switch")
            }
        }
    }

    /// Return the user provided value of this flag.
    ///
    /// If this flag is a switch, then this panics.
    fn unwrap_value(self) -> OsString {
        match self {
            FlagValue::Switch(_) => {
                unreachable!("got switch but expected flag value")
            }
            FlagValue::Value(v) => v,
        }
    }
}
```

Using `OsString` instead of `String` is crucial—it allows flag values to contain arbitrary bytes, which matters on Unix where paths can contain non-UTF-8 sequences. The `unreachable!` macro in the unwrap methods indicates programmer error rather than user error.

## Quick Reference

### Flag Trait Method Summary

| Method | Required | Returns | Purpose |
|---|---|---|---|
| `is_switch` | Yes | `bool` | Determines if flag takes a value |
| `name_short` | No | `Option<u8>` | Single-character alias |
| `name_long` | Yes | `&'static str` | Primary flag name |
| `aliases` | No | `&'static [&'static str]` | Alternative names |
| `name_negated` | No | `Option<&'static str>` | `--no-*` form |
| `doc_variable` | No | `Option<&'static str>` | Value placeholder (e.g., `NUM`) |
| `doc_category` | Yes | `Category` | Documentation grouping |
| `doc_short` | Yes | `&'static str` | Brief `-h` description |
| `doc_long` | Yes | `&'static str` | Full `--help`/man description |
| `doc_choices` | No | `&'static [&'static str]` | Valid values list |
| `completion_type` | No | `CompletionType` | Shell completion hint |
| `update` | Yes | `anyhow::Result<()>` | Apply flag to args |

## Trait Bounds on Flag

```
Flag: Debug + Send + Sync + UnwindSafe + RefUnwindSafe + 'static
```

## Architecture Flow

```
Command Line Input
        ↓
    parse::parse()
        ↓
    LowArgs (raw flag values)
        ↓
    HiArgs (validated, high-level)
        ↓
    ripgrep execution
```

# Low-Level Arguments: Code Companion

Reference code for the Low-Level Arguments lecture. Sections correspond to the lecture document.

---

## Section 1: The Anatomy of Low-Level Arguments

```rust
/// A collection of "low level" arguments.
///
/// The "low level" here is meant to constrain this type to be as close to the
/// actual CLI flags and arguments as possible.
#[derive(Debug, Default)]
pub(crate) struct LowArgs {
    // Essential arguments.
    pub(crate) special: Option<SpecialMode>,
    pub(crate) mode: Mode,
    pub(crate) positional: Vec<OsString>,
    pub(crate) patterns: Vec<PatternSource>,

    // Everything else, sorted lexicographically.
    pub(crate) binary: BinaryMode,
    pub(crate) boundary: Option<BoundaryMode>,
    pub(crate) buffer: BufferMode,
    pub(crate) byte_offset: bool,
    pub(crate) case: CaseMode,
    pub(crate) color: ColorChoice,
    pub(crate) colors: Vec<UserColorSpec>,
    pub(crate) column: Option<bool>,
    pub(crate) context: ContextMode,
    // ... many more fields follow
    pub(crate) encoding: EncodingMode,
    pub(crate) engine: EngineChoice,
    pub(crate) fixed_strings: bool,
    pub(crate) follow: bool,
    pub(crate) globs: Vec<String>,
    pub(crate) hidden: bool,
    pub(crate) max_count: Option<u64>,
    pub(crate) multiline: bool,
    pub(crate) threads: Option<usize>,
    pub(crate) type_changes: Vec<TypeChange>,
    pub(crate) unrestricted: usize,
    pub(crate) vimgrep: bool,
    pub(crate) with_filename: Option<bool>,
}
```

The `pub(crate)` visibility allows any module within the crate to read/write fields directly, while hiding them from external consumers. The `Default` derive enables incremental construction during parsing.

## Section 2: Special Modes and Short-Circuiting

```rust
/// A "special" mode that supercedes everything else.
///
/// When one of these modes is present, it overrides everything else and causes
/// ripgrep to short-circuit.
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
pub(crate) enum SpecialMode {
    /// Condensed help output (-h flag)
    HelpShort,
    /// Verbose help output (--help flag)
    HelpLong,
    /// Condensed version (ripgrep x.y.z)
    VersionShort,
    /// Verbose version with build features
    VersionLong,
    /// PCRE2's version information
    VersionPCRE2,
}
```

The enum is `Copy` because it's small and frequently passed around. The distinction between Short/Long variants enables Unix-style `-h` vs `--help` behavior.

---

## Section 3: Operational Modes and Override Semantics

```rust
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
pub(crate) enum Mode {
    /// ripgrep will execute a search of some kind.
    Search(SearchMode),
    /// Show files that *would* be searched without searching.
    Files,
    /// List all file type definitions configured.
    Types,
    /// Generate man pages, completions, etc.
    Generate(GenerateMode),
}

impl Default for Mode {
    fn default() -> Mode {
        Mode::Search(SearchMode::Standard)
    }
}

impl Mode {
    /// Update this mode while implementing override semantics.
    pub(crate) fn update(&mut self, new: Mode) {
        match *self {
            // Search mode can be overridden by anything
            Mode::Search(_) => *self = new,
            _ => {
                // Non-search modes can override each other,
                // but search modes cannot override non-search modes.
                // Example: `--files -l` stays as Mode::Files
                if !matches!(*self, Mode::Search(_)) {
                    *self = new;
                }
            }
        }
    }
}
```

The `update` method's logic ensures explicit mode flags (like `--files`) take precedence over implicit search mode flags from config files.

## Section 4: Search Mode Variations

```rust
/// The kind of search that ripgrep is going to perform.
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
pub(crate) enum SearchMode {
    /// Default mode - print matching lines
    /// No explicit flag; use --no-json to reset to this
    Standard,
    /// Show files containing at least one match (-l)
    FilesWithMatches,
    /// Show files that don't contain any matches (--files-without-match)
    FilesWithoutMatch,
    /// Show files with match count per file (-c)
    Count,
    /// Show files with total match count (--count-matches)
    CountMatches,
    /// Print matches in JSON lines format (--json)
    JSON,
}
```

`Count` vs `CountMatches`: a file with one line containing three matches contributes 1 to `Count` but 3 to `CountMatches`.

---

## Section 5: Binary Data Handling

```rust
/// Indicates how ripgrep should treat binary data.
#[derive(Debug, Default, Eq, PartialEq)]
pub(crate) enum BinaryMode {
    /// Context-dependent: explicit files get SearchAndSuppress,
    /// discovered files get aggressive binary skipping.
    #[default]
    Auto,

    /// Search binary files but suppress matches and warn.
    /// NUL bytes replaced with line terminators to prevent
    /// memory exhaustion from long binary "lines".
    SearchAndSuppress,

    /// Treat all files as plain text. No skipping, no NUL replacement.
    AsText,
}
```

The `#[default]` attribute on `Auto` eliminates the need for a separate `Default` impl. The NUL-to-newline replacement in `SearchAndSuppress` is a crucial memory safety heuristic.

---

## Section 6: Context Mode and Precedence

```rust
/// Line context options for output.
#[derive(Debug, Eq, PartialEq)]
pub(crate) enum ContextMode {
    /// Print all lines (unbounded context)
    Passthru,
    /// Show specific number of lines before/after matches
    Limited(ContextModeLimited),
}

/// Tracks before/after/both separately for correct precedence.
#[derive(Debug, Default, Eq, PartialEq)]
pub(crate) struct ContextModeLimited {
    before: Option<usize>,  // -B flag
    after: Option<usize>,   // -A flag
    both: Option<usize>,    // -C flag
}

impl ContextModeLimited {
    /// Returns (before, after) with correct precedence:
    /// -A/-B always override -C regardless of order.
    pub(crate) fn get(&self) -> (usize, usize) {
        // Start with -C value or (0, 0)
        let (mut before, mut after) =
            self.both.map(|lines| (lines, lines)).unwrap_or((0, 0));

        // -A and -B always win over -C
        if let Some(lines) = self.before {
            before = lines;
        }
        if let Some(lines) = self.after {
            after = lines;
        }
        (before, after)
    }
}
```

The three-field design with deferred resolution in `get()` enables order-independent flag processing where `-A`/`-B` always override `-C`.

## Quick Reference

### Mode Hierarchy

```
SpecialMode (highest priority - short-circuits everything)
    └── HelpShort, HelpLong, VersionShort, VersionLong, VersionPCRE2

Mode (operational mode)
    ├── Search(SearchMode) - default, can be overridden
    ├── Files - overrides search, can be overridden by other non-search
    ├── Types - overrides search
    └── Generate(GenerateMode) - overrides search
```

## Key Types Summary

| Type | Purpose | Default |
|---|---|---|
| `LowArgs` | All CLI arguments | `Default::default()` |
| `SpecialMode` | Short-circuit operations | `None` |
| `Mode` | What ripgrep does | `Search(Standard)` |
| `SearchMode` | How to report matches | `Standard` |
| `BinaryMode` | Binary file handling | `Auto` |
| `ContextMode` | Lines around matches | `Limited(0, 0)` |
| `ColorChoice` | Output coloring | `Auto` |
| `EncodingMode` | Text encoding | `Auto` |
| `EngineChoice` | Regex engine | `Default` |

## Visibility Pattern

```rust
pub(crate) struct LowArgs {
    pub(crate) field: Type,  // Crate-internal access only
}
```

All fields use `pub(crate)` - accessible within the crate, hidden from external consumers.

# ripgrep crates/core/flags/hiargs.rs: Code Companion

Reference code for the High-Level Arguments lecture. Sections correspond to the lecture document.

---

## Section 1: The HiArgs Structure and Its Philosophy

```rust
/// A high level representation of CLI arguments.
///
/// The distinction between low and high level arguments is somewhat arbitrary
/// and wishy washy. The main idea here is that high level arguments generally
/// require all of CLI parsing to be finished.
#[derive(Debug)]
pub(crate) struct HiArgs {
    // Direct copies from LowArgs - simple values that pass through unchanged
    binary: BinaryDetection,
    boundary: Option<BoundaryMode>,
    buffer: BufferMode,
    byte_offset: bool,
    case: CaseMode,

    // Computed from multiple low-level arguments
    color: ColorChoice,              // Depends on terminal detection + user flag
    colors: grep::printer::ColorSpecs,
    column: bool,                    // Derived from --column OR --vimgrep
    heading: bool,                   // Complex logic involving terminal +
vimgrep

    // Fully-constructed objects from external crates
    globs: ignore::overrides::Override,  // Compiled glob matcher
    types: ignore::types::Types,         // File type matcher
    mmap_choice: grep::searcher::MmapChoice,
    hyperlink_config: grep::printer::HyperlinkConfig,

    // Derived state affecting search behavior
    quit_after_match: bool,          // Depends on both stats and quiet
    threads: usize,                  // Computed from sort mode, path count,
available CPUs
    with_filename: bool,             // Complex default based on vimgrep and path
count

    // ... over 80 fields total
    cwd: PathBuf,
    is_terminal_stdout: bool,
    line_number: bool,
    paths: Paths,
    patterns: Patterns,
    // ... many more
}
```

The field groupings show the progression from raw values to computed configuration. Fields like `globs` and `types` are fully constructed matchers ready for use during search.

---

## Section 2: The Conversion Process and State Management

```
impl HiArgs {
    pub(crate) fn from_low_args(mut low: LowArgs) -> anyhow::Result<HiArgs> {
        // Defensive assertion - special modes should never reach this point
        assert_eq!(None, low.special, "special mode demands short-circuiting");

        // Validate sorting mode support early
        if let Some(ref sort) = low.sort {
            sort.supported()?;
        }

        // Mode adjustments for flag interactions
        match low.mode {
            Mode::Search(ref mut mode) => match *mode {
                // -v --count-matches becomes -v --count
                // (counting individual matches makes no sense when inverting)
                SearchMode::CountMatches if low.invert_match => {
                    *mode = SearchMode::Count;
                }
                // -o --count becomes --count-matches
                SearchMode::Count if low.only_matching => {
                    *mode = SearchMode::CountMatches;
                }
                _ => {}
            },
            _ => {}
        }

        // Initialize shared state for conversion functions
        let mut state = State::new()?;

        // Order matters: patterns must be read before paths
        let patterns = Patterns::from_low_args(&mut state, &mut low)?;
        let paths = Paths::from_low_args(&mut state, &patterns, &mut low)?;

        // Build complex objects using resolved state
        let binary = BinaryDetection::from_low_args(&state, &low);
        let colors = take_color_specs(&mut state, &mut low);
        // ... more conversions
    }
}

/// State shared across all low->high argument conversions.
#[derive(Debug)]
struct State {
    /// Terminal detection affects color and heading defaults
    is_terminal_stdout: bool,
    /// Prevents double-reading stdin (patterns vs search input)
    stdin_consumed: bool,
    /// Cached for glob matching and path resolution
    cwd: PathBuf,
}

impl State {
    fn new() -> anyhow::Result<State> {
```

```
        use std::io::IsTerminal;        let cwd = current_dir()?;
log::debug!("read CWD from environment: {}", cwd.display());        Ok(State {
is_terminal_stdout: std::io::stdout().is_terminal(),            stdin_consumed:
false,            cwd,        })    }}
```

The `State` struct acts as a coordination mechanism—mutations by one conversion function (like setting `stdin_consumed = true`) affect subsequent conversions.

---

## Section 3: Pattern Collection and Deduplication

```
/// The disjunction of patterns to search for.
#[derive(Debug)]
struct Patterns {
    patterns: Vec<String>,
}

impl Patterns {
    fn from_low_args(
        state: &mut State,
        low: &mut LowArgs,
    ) -> anyhow::Result<Patterns> {
        // Non-search modes don't need patterns
        if !matches!(low.mode, Mode::Search(_)) {
            return Ok(Patterns { patterns: vec![] });
        }

        // Common case: first positional is the pattern (rg foo)
        if low.patterns.is_empty() {
            anyhow::ensure!(
                !low.positional.is_empty(),
                "ripgrep requires at least one pattern to execute a search"
            );
            let ospat = low.positional.remove(0);
            let Ok(pat) = ospat.into_string() else {
                anyhow::bail!("pattern given is not valid UTF-8")
            };
            return Ok(Patterns { patterns: vec![pat] });
        }

        // Deduplication with HashSet - prevents regex engine slowdowns
        let mut seen = HashSet::new();
        let mut patterns = Vec::with_capacity(low.patterns.len());
        let mut add = |pat: String| {
            if !seen.contains(&pat) {
                seen.insert(pat.clone());
                patterns.push(pat);
            }
        };

        // Collect from all sources: -e/--regexp and -f/--file
        for source in low.patterns.drain(..) {
            match source {
                PatternSource::Regexp(pat) => add(pat),
                PatternSource::File(path) => {
                    if path == Path::new("-") {
                        // Stdin guard: ensure we haven't already consumed it
                        anyhow::ensure!(
                            !state.stdin_consumed,
                            "error reading -f/--file from stdin: stdin \
                             has already been consumed"
                        );
                        for pat in grep::cli::patterns_from_stdin()? {
                            add(pat);
                        }
```

```
                        // Mark stdin as consumed for later path handling
state.stdin_consumed = true;                    } else {
for pat in grep::cli::patterns_from_path(&path)? {
add(pat);                           }                   }               }
}       }       Ok(Patterns { patterns })    }}
```

The `seen` HashSet adds memory overhead but prevents pathological performance when users accidentally pass thousands of duplicate patterns.

---

## Section 4: Path Resolution and Implicit Behavior

```
/// The collection of paths to search.
#[derive(Debug)]
struct Paths {
    paths: Vec<PathBuf>,
    /// True when ripgrep "guessed" to search CWD (affects path printing)
    has_implicit_path: bool,
    /// Enables single-file optimizations
    is_one_file: bool,
}

impl Paths {
    fn from_low_args(
        state: &mut State,
        _: &Patterns,  // Required to ensure patterns consumed first
        low: &mut LowArgs,
    ) -> anyhow::Result<Paths> {
        let mut paths = Vec::with_capacity(low.positional.len());
        for osarg in low.positional.drain(..) {
            let path = PathBuf::from(osarg);
            if state.stdin_consumed && path == Path::new("-") {
                anyhow::bail!(
                    "error: attempted to read patterns from stdin \
                     while also searching stdin",
                );
            }
            paths.push(path);
        }

        if !paths.is_empty() {
            // Note: !is_dir() rather than is_file() - handles device files,
pipes
            // See: https://github.com/BurntSushi/ripgrep/issues/2736
            let is_one_file = paths.len() == 1
                && (paths[0] == Path::new("-") || !paths[0].is_dir());
            return Ok(Paths { paths, has_implicit_path: false, is_one_file });
        }

        // Heuristic: determine if we should search stdin or CWD
        let is_readable_stdin = grep::cli::is_readable_stdin();
        let use_cwd = !is_readable_stdin
            || state.stdin_consumed
            || !matches!(low.mode, Mode::Search(_));

        let (path, is_one_file) = if use_cwd {
            log::debug!("heuristic chose to search ./");
            (PathBuf::from("./"), false)
        } else {
            log::debug!("heuristic chose to search stdin");
            (PathBuf::from("-"), true)
        };

        Ok(Paths { paths: vec![path], has_implicit_path: true, is_one_file })
    }
}
```

The `has_implicit_path` field determines whether paths are printed with `./` prefix—matching GNU grep's behavior for script compatibility.

---

## Section 5: Binary Detection Strategy

```rust
/// Dual binary detection modes: one for explicit files, one for discovered
files.
#[derive(Debug)]
struct BinaryDetection {
    /// Used when user explicitly names a file
    explicit: grep::searcher::BinaryDetection,
    /// Used when file is discovered during directory traversal
    implicit: grep::searcher::BinaryDetection,
}

impl BinaryDetection {
    fn from_low_args(_: &State, low: &LowArgs) -> BinaryDetection {
        // -a/--text or --null-data disables all binary detection
        let none = matches!(low.binary, BinaryMode::AsText) || low.null_data;
        // --binary mode: search but suppress binary content in output
        let convert = matches!(low.binary, BinaryMode::SearchAndSuppress);

        // Explicit files: never quit early (would be filtering user's request)
        let explicit = if none {
            grep::searcher::BinaryDetection::none()
        } else {
            // Convert NUL bytes but don't quit - user asked for this file
            grep::searcher::BinaryDetection::convert(b'\x00')
        };

        // Implicit files: can quit early (user didn't specifically request them)
        let implicit = if none {
            grep::searcher::BinaryDetection::none()
        } else if convert {
            grep::searcher::BinaryDetection::convert(b'\x00')
        } else {
            // Default: quit on binary to skip non-text files quickly
            grep::searcher::BinaryDetection::quit(b'\x00')
        };

        BinaryDetection { explicit, implicit }
    }

    /// Returns true when binary detection is completely disabled.
    pub(crate) fn is_none(&self) -> bool {
        let none = grep::searcher::BinaryDetection::none();
        self.explicit == none && self.implicit == none
    }
}
```

The philosophy: explicit requests should never be filtered, but implicit discovery can use aggressive heuristics for performance.

# Section 6: The Builder Pattern Ecosystem in Action

```rust
impl HiArgs {
    /// Builds a "standard" grep printer with all configured options.
    fn printer_standard<W: termcolor::WriteColor>(
        &self,
        wtr: W,
    ) -> grep::printer::Standard<W> {
        let mut builder = grep::printer::StandardBuilder::new();
        builder
            // Output formatting
            .byte_offset(self.byte_offset)
            .column(self.column)
            .heading(self.heading)
            .only_matching(self.only_matching)
            .trim_ascii(self.trim)
            // Colors and hyperlinks
            .color_specs(self.colors.clone())
            .hyperlink(self.hyperlink_config.clone())
            // Column/line limits
            .max_columns_preview(self.max_columns_preview)
            .max_columns(self.max_columns)
            // Path handling
            .path(self.with_filename)
            .path_terminator(self.path_terminator.clone())
            .separator_path(self.path_separator.clone())
            // Vimgrep compatibility
            .per_match_one_line(true)
            .per_match(self.vimgrep)
            // Replacement and context
            .replacement(self.replace.clone().map(|r| r.into()))
            .separator_context(self.context_separator.clone().into_bytes())
            .separator_field_context(
                self.field_context_separator.clone().into_bytes(),
            )
            .separator_field_match(
                self.field_match_separator.clone().into_bytes(),
            )
            .stats(self.stats.is_some());

        // Single-threaded: printer handles file separators
        // Multi-threaded: buffer writer handles them (has global visibility)
        if self.threads == 1 {
            builder.separator_search(self.file_separator.clone());
        }
        builder.build(wtr)
    }

    /// Creates a directory walker configured from all relevant flags.
    pub(crate) fn walk_builder(&self) -> anyhow::Result<ignore::WalkBuilder> {
        let mut builder = ignore::WalkBuilder::new(&self.paths.paths[0]);
        for path in self.paths.paths.iter().skip(1) {
            builder.add(path);
        }

        builder
```

```
            .max_depth(self.max_depth)             .follow_links(self.follow)
 .max_filesize(self.max_filesize)             .threads(self.threads)
 .same_file_system(self.one_file_system)             .overrides(self.globs.clone())
 .types(self.types.clone())             .hidden(!self.hidden)
 .parents(!self.no_ignore_parent)             .ignore(!self.no_ignore_dot)
 .git_global(!self.no_ignore_vcs && !self.no_ignore_global)
 .git_ignore(!self.no_ignore_vcs)             .git_exclude(!self.no_ignore_vcs &&
 !self.no_ignore_exclude)             .require_git(!self.no_require_git);
 Ok(builder)    }}
```

Each builder method maps HiArgs fields to library configuration. The builder pattern allows extensive configuration without unwieldy constructor signatures.

## Quick Reference

### Key Type Relationships

| Low-Level (`LowArgs`) | High-Level (`HiArgs`) | Notes |
|---|---|---|
| `Vec<PatternSource>` | `Patterns` | Deduplicated, unified from all sources |
| `Vec<OsString>` (positional) | `Paths` | With implicit path detection |
| `BinaryMode` | `BinaryDetection` | Split into explicit/implicit modes |
| `Vec<ColorSpec>` | `ColorSpecs` | Compiled into single config |
| `Vec<TypeChange>` | `ignore::types::Types` | Fully built type matcher |
| `Vec<String>` (globs) | `ignore::overrides::Override` | Compiled glob matcher |

### State Flow During Conversion

```
LowArgs + State::new()
    │
    ├─► Patterns::from_low_args()  ──► may set stdin_consumed = true
    │
    ├─► Paths::from_low_args()     ──► checks stdin_consumed
    │
    ├─► BinaryDetection::from_low_args()
    │
    ├─► types(), globs(), stats()  ──► build external crate objects
    │
    └─► HiArgs { ... }             ──► fully resolved configuration
```

## Computed Field Derivations

```
// Threading decision
threads = if sorting || one_file { 1 }
          else { user_specified || min(available_cpus, 12) }

// Filename display
with_filename = user_specified || vimgrep || !is_one_file

// Line numbers
line_number = user_specified || (terminal && !stdin_only) || column || vimgrep

// Early exit optimization
quit_after_match = stats.is_none() && quiet
```

# ripgrep crates/core/search.rs: Code Companion

Reference code for the Search Orchestration lecture. Sections correspond to the lecture document.

---

## Section 1: The Configuration Foundation

```rust
/// The configuration for the search worker.
///
/// Among a few other things, the configuration primarily controls the way we
/// show search results to users at a very high level.
#[derive(Clone, Debug)]
struct Config {
    // External command to preprocess files before searching
    preprocessor: Option<std::path::PathBuf>,
    // Glob patterns to select which files get preprocessed
    preprocessor_globs: ignore::overrides::Override,
    // Whether to automatically decompress archives
    search_zip: bool,
    // Binary detection for files discovered during directory walking
    binary_implicit: grep::searcher::BinaryDetection,
    // Binary detection for files explicitly named by the user
    binary_explicit: grep::searcher::BinaryDetection,
}

impl Default for Config {
    fn default() -> Config {
        Config {
            preprocessor: None,
            preprocessor_globs: ignore::overrides::Override::empty(),
            search_zip: false,
            // By default, no binary detection - search everything
            binary_implicit: grep::searcher::BinaryDetection::none(),
            binary_explicit: grep::searcher::BinaryDetection::none(),
        }
    }
}
```

The `Override` type from the `ignore` crate provides gitignore-style glob matching. The two binary detection fields allow different policies based on how files were discovered.

## Section 2: The Builder Pattern in Action

```
/// A builder for configuring and constructing a search worker.
#[derive(Clone, Debug)]
pub(crate) struct SearchWorkerBuilder {
    config: Config,
    command_builder: grep::cli::CommandReaderBuilder,
}

impl SearchWorkerBuilder {
    /// Create a new builder for configuring and constructing a search worker.
    pub(crate) fn new() -> SearchWorkerBuilder {
        let mut command_builder = grep::cli::CommandReaderBuilder::new();
        // Async stderr prevents deadlocks when preprocessors write errors
        command_builder.async_stderr(true);

        SearchWorkerBuilder { config: Config::default(), command_builder }
    }

    /// Create a new search worker using the given searcher, matcher and
    /// printer.
    pub(crate) fn build<W: WriteColor>(
        &self,
        matcher: PatternMatcher,
        searcher: grep::searcher::Searcher,
        printer: Printer<W>,
    ) -> SearchWorker<W> {
        let config = self.config.clone();
        let command_builder = self.command_builder.clone();
        // Only create decomp_builder when search_zip is enabled
        // This avoids work like resolving decompression binary paths
        let decomp_builder = config.search_zip.then(|| {
            let mut decomp_builder =
                grep::cli::DecompressionReaderBuilder::new();
            decomp_builder.async_stderr(true);
            decomp_builder
        });
        SearchWorker {
            config,
            command_builder,
            decomp_builder,
            matcher,
            searcher,
            printer,
        }
    }
}
```

The `then()` method on `bool` is an elegant way to conditionally create `Option<T>` values. It returns `Some(closure_result)` if true, `None` if false.

---

## Section 3: Configuring the Preprocessor Pipeline

```rust
/// Set the path to a preprocessor command.
///
/// When this is set, instead of searching files directly, the given
/// command will be run with the file path as the first argument, and the
/// output of that command will be searched instead.
pub(crate) fn preprocessor(
    &mut self,
    cmd: Option<std::path::PathBuf>,
) -> anyhow::Result<&mut SearchWorkerBuilder> {
    if let Some(ref prog) = cmd {
        // Resolve the binary path now, fail fast if it doesn't exist
        let bin = grep::cli::resolve_binary(prog)?;
        self.config.preprocessor = Some(bin);
    } else {
        self.config.preprocessor = None;
    }
    // Returns Result to allow early error detection
    Ok(self)
}

/// Set the globs for determining which files should be run through the
/// preprocessor. By default, with no globs and a preprocessor specified,
/// every file is run through the preprocessor.
pub(crate) fn preprocessor_globs(
    &mut self,
    globs: ignore::overrides::Override,
) -> &mut SearchWorkerBuilder {
    self.config.preprocessor_globs = globs;
    self
}

/// Returns true if and only if the given file path should be run through
/// the preprocessor.
fn should_preprocess(&self, path: &Path) -> bool {
    if !self.config.preprocessor.is_some() {
        return false;
    }
    if self.config.preprocessor_globs.is_empty() {
        // No globs means preprocess everything
        return true;
    }
    // Glob match: file is preprocessed unless it's explicitly ignored
    !self.config.preprocessor_globs.matched(path, false).is_ignore()
}
```

The `resolve_binary` function handles platform-specific concerns like finding executables with implicit `.exe` extensions on Windows.

## Section 4: The PatternMatcher and Printer Enums

```rust
/// The pattern matcher used by a search worker.
#[derive(Clone, Debug)]
pub(crate) enum PatternMatcher {
    RustRegex(grep::regex::RegexMatcher),
    // Only included when pcre2 feature is enabled
    #[cfg(feature = "pcre2")]
    PCRE2(grep::pcre2::RegexMatcher),
}

/// The printer used by a search worker.
///
/// The `W` type parameter refers to the type of the underlying writer.
#[derive(Clone, Debug)]
pub(crate) enum Printer<W> {
    /// Use the standard printer, which supports the classic grep-like format.
    Standard(grep::printer::Standard<W>),
    /// Use the summary printer, which supports aggregate displays of search
    /// results.
    Summary(grep::printer::Summary<W>),
    /// A JSON printer, which emits results in the JSON Lines format.
    JSON(grep::printer::JSON<W>),
}

impl<W: WriteColor> Printer<W> {
    /// Return a mutable reference to the underlying printer's writer.
    pub(crate) fn get_mut(&mut self) -> &mut W {
        match *self {
            Printer::Standard(ref mut p) => p.get_mut(),
            Printer::Summary(ref mut p) => p.get_mut(),
            Printer::JSON(ref mut p) => p.get_mut(),
        }
    }
}
```

The `WriteColor` trait bound (from `termcolor`) ensures the writer supports colored output. This abstracts over stdout, files, and test buffers.

---

## Section 5: The SearchWorker Structure

```rust
/// A worker for executing searches.
///
/// It is intended for a single worker to execute many searches, and is
/// generally intended to be used from a single thread. When searching using
/// multiple threads, it is better to create a new worker for each thread.
#[derive(Clone, Debug)]
pub(crate) struct SearchWorker<W> {
    config: Config,
    command_builder: grep::cli::CommandReaderBuilder,
    /// This is `None` when `search_zip` is not enabled, since in this case it
    /// can never be used. We do this because building the reader can sometimes
    /// do non-trivial work (like resolving the paths of decompression binaries
    /// on Windows).
    decomp_builder: Option<grep::cli::DecompressionReaderBuilder>,
    matcher: PatternMatcher,
    searcher: grep::searcher::Searcher,
    printer: Printer<W>,
}
```

The `Clone` derive enables creating per-thread worker instances for parallel search. Each thread gets its own worker to avoid contention.

---

## Section 6: The Main Search Entry Point

```
impl<W: WriteColor> SearchWorker<W> {
    /// Execute a search over the given haystack.
    pub(crate) fn search(
        &mut self,
        haystack: &crate::haystack::Haystack,
    ) -> io::Result<SearchResult> {
        // Select binary detection based on how the file was discovered
        let bin = if haystack.is_explicit() {
            self.config.binary_explicit.clone()
        } else {
            self.config.binary_implicit.clone()
        };
        let path = haystack.path();
        log::trace!("{}: binary detection: {:?}", path.display(), bin);

        // Configure the searcher with the chosen detection strategy
        self.searcher.set_binary_detection(bin);

        // Decision tree: how should we read this file?
        if haystack.is_stdin() {
            self.search_reader(path, &mut io::stdin().lock())
        } else if self.should_preprocess(path) {
            self.search_preprocessor(path)
        } else if self.should_decompress(path) {
            self.search_decompress(path)
        } else {
            self.search_path(path)
        }
    }

    /// Returns true if and only if the given file path should be
    /// decompressed before searching.
    fn should_decompress(&self, path: &Path) -> bool {
        self.decomp_builder.as_ref().is_some_and(|decomp_builder| {
            decomp_builder.get_matcher().has_command(path)
        })
    }
}
```

The `is_some_and` method combines `Option::is_some` with a predicate check in one step, avoiding unnecessary unwrapping.

## Section 7: The Preprocessor Search Path

```rust
/// Search the given file path by first asking the preprocessor for the
/// data to search instead of opening the path directly.
fn search_preprocessor(
    &mut self,
    path: &Path,
) -> io::Result<SearchResult> {
    use std::{fs::File, process::Stdio};

    let bin = self.config.preprocessor.as_ref().unwrap();
    let mut cmd = std::process::Command::new(bin);
    // Pass file path as argument, also provide file content via stdin
    cmd.arg(path).stdin(Stdio::from(File::open(path)?));

    // Build a reader that captures stdout from the command
    let mut rdr = self.command_builder.build(&mut cmd).map_err(|err| {
        io::Error::new(
            io::ErrorKind::Other,
            format!(
                "preprocessor command could not start: '{cmd:?}': {err}",
            ),
        )
    })?;

    // Search the command's output, not the original file
    let result = self.search_reader(path, &mut rdr).map_err(|err| {
        io::Error::new(
            io::ErrorKind::Other,
            format!("preprocessor command failed: '{cmd:?}': {err}"),
        )
    });

    // Important: close the reader to wait for the process to exit
    let close_result = rdr.close();
    let search_result = result?;
    close_result?;
    Ok(search_result)
}
```

The error handling wraps original errors with context about which command failed. The `close()` call ensures the subprocess is properly reaped.

## Section 8: Decompression and Direct File Search

```
/// Attempt to decompress the data at the given file path and search the
/// result. If the given file path isn't recognized as a compressed file,
/// then search it without doing any decompression.
fn search_decompress(&mut self, path: &Path) -> io::Result<SearchResult> {
    let Some(ref decomp_builder) = self.decomp_builder else {
        // No decompression builder - fall back to direct search
        return self.search_path(path);
    };
    let mut rdr = decomp_builder.build(path)?;
    let result = self.search_reader(path, &mut rdr);
    let close_result = rdr.close();
    let search_result = result?;
    close_result?;
    Ok(search_result)
}

/// Search the contents of the given file path.
fn search_path(&mut self, path: &Path) -> io::Result<SearchResult> {
    use self::PatternMatcher::*;

    let (searcher, printer) = (&mut self.searcher, &mut self.printer);
    // Dispatch to the appropriate regex engine
    match self.matcher {
        RustRegex(ref m) => search_path(m, searcher, printer, path),
        #[cfg(feature = "pcre2")]
        PCRE2(ref m) => search_path(m, searcher, printer, path),
    }
}
```

The `let-else` pattern (`let Some(...) = ... else { return ... }`) provides clean early returns for `Option` values.

## Section 9: The Generic Search Functions

```
/// Search the contents of the given file path using the given matcher,
/// searcher and printer.
fn search_path<M: Matcher, W: WriteColor>(
    matcher: M,
    searcher: &mut grep::searcher::Searcher,
    printer: &mut Printer<W>,
    path: &Path,
) -> io::Result<SearchResult> {
    match *printer {
        Printer::Standard(ref mut p) => {
            // Create a sink that connects the printer to the path
            let mut sink = p.sink_with_path(&matcher, path);
            // Execute the search, writing results to the sink
            searcher.search_path(&matcher, path, &mut sink)?;
            Ok(SearchResult {
                has_match: sink.has_match(),
                stats: sink.stats().map(|s| s.clone()),
            })
        }
        Printer::Summary(ref mut p) => {
            let mut sink = p.sink_with_path(&matcher, path);
            searcher.search_path(&matcher, path, &mut sink)?;
            Ok(SearchResult {
                has_match: sink.has_match(),
                stats: sink.stats().map(|s| s.clone()),
            })
        }
        Printer::JSON(ref mut p) => {
            let mut sink = p.sink_with_path(&matcher, path);
            searcher.search_path(&matcher, path, &mut sink)?;
            Ok(SearchResult {
                has_match: sink.has_match(),
                // JSON printer always has stats
                stats: Some(sink.stats().clone()),
            })
        }
    }
}
```

The `Matcher` trait bound allows any regex engine that implements the grep crate's matching interface. The sink pattern connects the searcher to the printer.

## Section 10: The SearchResult Type

```rust
/// The result of executing a search.
///
/// Generally speaking, the "result" of a search is sent to a printer, which
/// writes results to an underlying writer such as stdout or a file. However,
/// every search also has some aggregate statistics or meta data that may be
/// useful to higher level routines.
#[derive(Clone, Debug, Default)]
pub(crate) struct SearchResult {
    has_match: bool,
    stats: Option<grep::printer::Stats>,
}

impl SearchResult {
    /// Whether the search found a match or not.
    pub(crate) fn has_match(&self) -> bool {
        self.has_match
    }

    /// Return aggregate search statistics for a single search, if available.
    ///
    /// It can be expensive to compute statistics, so these are only present
    /// if explicitly enabled in the printer provided by the caller.
    pub(crate) fn stats(&self) -> Option<&grep::printer::Stats> {
        self.stats.as_ref()
    }
}
```

Statistics are optional because computing them has performance cost. The caller decides whether to enable stats collection in the printer configuration.

---

# Quick Reference

## Search Decision Flow

```
search(haystack)
    |
    ├─ stdin? ──────────────► search_reader(stdin)
    |
    ├─ preprocessor? ───────► search_preprocessor ──► search_reader
    |
    ├─ compressed? ─────────► search_decompress ──► search_reader
    |
    └─ regular file ────────► search_path (may use mmap)
```

## Key Types

| Type | Purpose |
|---|---|
| `Config` | High-level search settings |
| `SearchWorkerBuilder` | Fluent API for worker construction |
| `SearchWorker<W>` | Executes searches, holds all components |
| `PatternMatcher` | Abstracts over regex engines |
| `Printer<W>` | Abstracts over output formats |
| `SearchResult` | Match status and optional statistics |

## Binary Detection Strategies

| Strategy | Used For | Typical Setting |
|---|---|---|
| `binary_implicit` | Files found via directory walking | `quit` (skip binary files) |
| `binary_explicit` | Files named by user | `none` (search everything) |

## Trait Bounds

```
// Pattern matching abstraction
M: Matcher

// Colored output writer
W: WriteColor

// Generic readable input
R: io::Read
```

# ripgrep crates/core/haystack.rs: Code Companion

Reference code for the Haystack Discovery lecture. Sections correspond to the lecture document.

---

## Section 1: The Builder Pattern at Its Simplest

```rust
/// A builder for constructing things to search over.
#[derive(Clone, Debug)]
pub(crate) struct HaystackBuilder {
    strip_dot_prefix: bool,  // The single configuration option
}

impl HaystackBuilder {
    /// Return a new haystack builder with a default configuration.
    pub(crate) fn new() -> HaystackBuilder {
        HaystackBuilder { strip_dot_prefix: false }
    }

    /// When enabled, if the haystack's file path starts with `./` then it is
    /// stripped.
    ///
    /// This is useful when implicitly searching the current working directory.
    pub(crate) fn strip_dot_prefix(
        &mut self,
        yes: bool,
    ) -> &mut HaystackBuilder {  // Returns &mut Self for method chaining
        self.strip_dot_prefix = yes;
        self
    }
}
```

The `pub(crate)` visibility restricts these types to the `core` crate. The `strip_dot_prefix` method returns `&mut Self`, enabling the fluent builder pattern: `builder.strip_dot_prefix(true).build(entry)`.

---

## Section 2: Constructing from Fallible Results

```rust
/// Create a new haystack from a possibly missing directory entry.
///
/// If the directory entry isn't present, then the corresponding error is
/// logged if messages have been configured. Otherwise, if the directory
/// entry is deemed searchable, then it is returned as a haystack.
pub(crate) fn build_from_result(
    &self,
    result: Result<ignore::DirEntry, ignore::Error>,  // From directory traversal
) -> Option<Haystack> {
    match result {
        Ok(dent) => self.build(dent),  // Delegate to main build logic
        Err(err) => {
            err_message!("{err}");     // Log error, don't propagate
            None                       // Signal "nothing to search"
        }
    }
}
```

The `err_message!` macro is defined elsewhere in ripgrep's core crate. It handles conditional error output based on configuration. Returning `Option<Haystack>` lets callers use `filter_map` to seamlessly skip errors during iteration.

---

## Section 3: The Filtering Decision Tree

```
/// Create a new haystack using this builder's configuration.
///
/// If a directory entry could not be created or should otherwise not be
/// searched, then this returns `None` after emitting any relevant log
/// messages.
fn build(&self, dent: ignore::DirEntry) -> Option<Haystack> {
    // Wrap immediately to use Haystack's helper methods
    let hay = Haystack { dent, strip_dot_prefix: self.strip_dot_prefix };

    // Log partial errors but continue processing
    if let Some(err) = hay.dent.error() {
        ignore_message!("{err}");
    }

    // Priority 1: Explicit entries always pass through
    if hay.is_explicit() {
        return Some(hay);
    }

    // Priority 2: Regular files pass through
    if hay.is_file() {
        return Some(hay);
    }

    // Priority 3: Everything else gets rejected (with debug logging)
    if !hay.is_dir() {
        log::debug!(
            "ignoring {}: failed to pass haystack filter: \
             file type: {:?}, metadata: {:?}",
            hay.dent.path().display(),
            hay.dent.file_type(),
            hay.dent.metadata()
        );
    }
    None
}
```

The `ignore_message!` macro differs from `err_message!`—it's for warnings about things being skipped rather than outright failures. The decision tree's order matters: explicit paths get special treatment before the file-type check.

## Section 4: Explicit vs Discovered Paths

```rust
/// A haystack is a thing we want to search.
///
/// Generally, a haystack is either a file or stdin.
#[derive(Clone, Debug)]
pub(crate) struct Haystack {
    dent: ignore::DirEntry,      // The underlying directory entry
    strip_dot_prefix: bool       // Configuration baked in at construction
}

impl Haystack {
    /// Returns true if and only if this entry corresponds to stdin.
    pub(crate) fn is_stdin(&self) -> bool {
        self.dent.is_stdin()
    }

    /// Returns true if and only if this entry corresponds to a haystack to
    /// search that was explicitly supplied by an end user.
    ///
    /// Generally, this corresponds to either stdin or an explicit file path
    /// argument. e.g., in `rg foo some-file ./some-dir/`, `some-file` is
    /// an explicit haystack, but, e.g., `./some-dir/some-other-file` is not.
    ///
    /// However, note that ripgrep does not see through shell globbing. e.g.,
    /// in `rg foo ./some-dir/*`, `./some-dir/some-other-file` will be treated
    /// as an explicit haystack.
    pub(crate) fn is_explicit(&self) -> bool {
        // stdin is obvious. When an entry has a depth of 0, that means it
        // was explicitly provided to our directory iterator, which means it
        // was in turn explicitly provided by the end user. The !is_dir check
        // means that we want to search files even if their symlinks, again,
        // because they were explicitly provided. (And we never want to try
        // to search a directory.)
        self.is_stdin() || (self.dent.depth() == 0 && !self.is_dir())
    }
}
```

The `depth() == 0` check is the key heuristic: entries at depth zero were passed directly to the walker, not discovered during traversal. The `!is_dir()` check excludes directories since we search their contents, not the directories themselves.

## Section 5: File Type Detection and Symlink Handling

```rust
/// Returns true if and only if this haystack points to a directory after
/// following symbolic links.
fn is_dir(&self) -> bool {
    let ft = match self.dent.file_type() {
        None => return false,      // No file type means can't be a directory
        Some(ft) => ft,
    };
    if ft.is_dir() {
        return true;               // Direct directory
    }
    // If this is a symlink, then we want to follow it to determine
    // whether it's a directory or not.
    self.dent.path_is_symlink() && self.dent.path().is_dir()
}

/// Returns true if and only if this haystack points to a file.
fn is_file(&self) -> bool {
    self.dent.file_type().map_or(false, |ft| ft.is_file())
}
```

Note the asymmetry: `is_file` doesn't follow symlinks (uses only `file_type()`), but `is_dir` does (calls `path().is_dir()` which follows symlinks). This is intentional—discovered symlinks are skipped by the main filter, but we need symlink-aware directory checking for explicit paths.

---

## Section 6: Path Presentation and User Experience

```rust
impl Haystack {
    /// Return the file path corresponding to this haystack.
    ///
    /// If this haystack corresponds to stdin, then a special `<stdin>` path
    /// is returned instead.
    pub(crate) fn path(&self) -> &Path {
        if self.strip_dot_prefix && self.dent.path().starts_with("./") {
            // strip_prefix returns Result, but we know "./" is present
            self.dent.path().strip_prefix("./").unwrap()
        } else {
            self.dent.path()
        }
    }
}
```

The `unwrap()` here is safe because we've already verified the prefix exists with `starts_with("./")`. This transforms `./src/main.rs` into `src/main.rs` for cleaner output when searching the current directory.

---

# Quick Reference

## Type Summary

| Type | Visibility | Purpose |
|---|---|---|
| `HaystackBuilder` | `pub(crate)` | Configures and creates haystacks |
| `Haystack` | `pub(crate)` | Wraps searchable directory entries |

## Haystack Decision Flow

```
DirEntry
    │
    ├─ Error? → Log, return None
    │
    ├─ Explicit? (stdin or depth=0) → Return Some(haystack)
    │
    ├─ Regular file? → Return Some(haystack)
    │
    └─ Otherwise → Log debug, return None
```

## Key Method Signatures

```rust
// Builder
fn new() -> HaystackBuilder
fn build_from_result(&self, Result<DirEntry, Error>) -> Option<Haystack>
fn build(&self, DirEntry) -> Option<Haystack>
fn strip_dot_prefix(&mut self, bool) -> &mut HaystackBuilder

// Haystack
fn path(&self) -> &Path
fn is_stdin(&self) -> bool
fn is_explicit(&self) -> bool
fn is_dir(&self) -> bool      // follows symlinks
fn is_file(&self) -> bool     // does NOT follow symlinks
```

## Explicit Path Heuristic

```
is_stdin() || (depth() == 0 && !is_dir())
```

# Chapter: The Ignore Crate — Intelligent File Discovery

## Overview

The ignore crate is ripgrep's file discovery engine. When you run `rg pattern`, something has to decide which files to search. That "something" is the ignore crate — a sophisticated directory walker that respects gitignore rules, filters by file type, and traverses directories in parallel.

This crate embodies a key insight: **the fastest way to search a file is to not search it at all**. By intelligently filtering files during traversal, ripgrep avoids ever opening files that can't match. This filtering happens at multiple levels: hidden files, gitignore patterns, file type globs, and size limits — all evaluated before any regex touches the file.

The ignore crate is also ripgrep's most reusable component. Unlike grep-searcher or grep-printer which are tightly coupled to search functionality, ignore is a general-purpose library. Other tools like `fd` (a find replacement) use it directly.

---

## What This Crate Provides

The ignore crate offers three main capabilities:

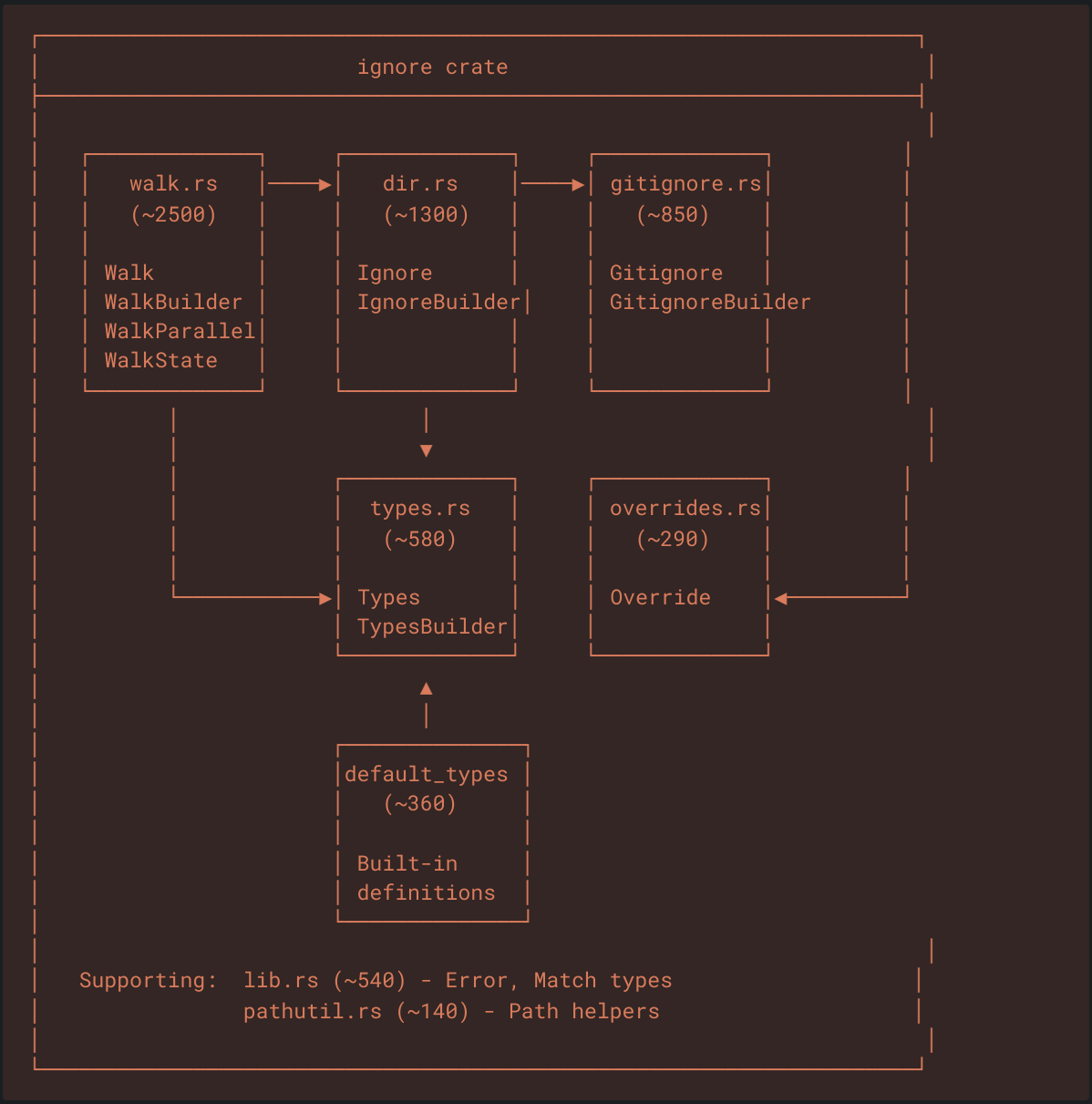
**1. Recursive Directory Walking** - Sequential iteration via `Walk` - Parallel traversal via `WalkParallel` - Builder pattern for extensive configuration (`WalkBuilder`) - The `WalkState` enum for controlling traversal

**2. Gitignore Processing** - Parsing `.gitignore`, `.ignore`, and `.rgignore` files - Proper precedence rules (child overrides parent) - Negation patterns (lines starting with `!`) - Global git excludes

**3. File Type Filtering** - Built-in type definitions (rust, python, c, etc.) - Custom type definitions via `--type-add` - Type selection (`-t`) and negation (`-T`)

---

## Crate Architecture




## Module Summary

| Module | Lines | Purpose |
|---|---|---|
| **walk.rs** | ~2500 | Directory traversal, parallel iteration, the closure-factory pattern |
| **dir.rs** | ~1300 | Per-directory ignore state, rule accumulation |

| | | |
|---|---|---|
| **gitignore.rs** | ~850 | Gitignore file parsing, glob matching |
| **types.rs** | ~580 | File type definitions and matching |
| **default_types.rs** | ~360 | Built-in type definitions (rust:*.rs, etc.) |
| **overrides.rs** | ~290 | Command-line glob overrides (-g flag) |
| **pathutil.rs** | ~140 | Path normalization utilities |
| **lib.rs** | ~540 | Crate root, Error enum, Match enum |

## Table of Contents

This chapter covers the ignore crate in dependency order:


### Part 1: Foundation Types

**1.1 lib.rs — Crate Entry Point** - The `Error` enum and error wrapping patterns - The `Match<T>` enum for ignore/whitelist decisions - Partial error handling philosophy

### Part 2: Glob and Pattern Matching

**2.1 gitignore.rs — Gitignore Parsing** - Gitignore file format and semantics - Pattern compilation and matching - Precedence rules and negation

**2.2 overrides.rs — CLI Glob Overrides** - The `-g/--glob` flag implementation - Override vs ignore precedence

### Part 3: File Type System

**3.1 default_types.rs — Built-in Definitions** - How type definitions are stored - The DEFAULT_TYPES static

**3.2 types.rs — Type Matching** - TypesBuilder configuration - Type selection and negation - Glob-to-type matching

### Part 4: Directory State

**4.1 dir.rs — Per-Directory Ignore State** - The `Ignore` struct - Rule accumulation as you descend - Parent-child override semantics

### Part 5: The Walker

**5.1 walk.rs — Directory Traversal** - `Walk` for sequential iteration - `WalkParallel` for parallel traversal - `WalkBuilder` configuration - The closure-factory pattern - `WalkState` for traversal control - Thread coordination and work stealing

---

## Key Concepts Preview

### The Match Enum

Every filtering decision returns a `Match<T>`:

```
pub enum Match<T> {
    None,         // No rule matched
    Ignore(T),    // Should be ignored (skip this file)
    Whitelist(T) // Explicitly included (override ignore)
}
```

Negation patterns (lines starting with `!`) produce `Whitelist`. The precedence rule: **later rules win**, so a whitelist after an ignore re-includes the file.

### The Closure-Factory Pattern

Parallel traversal uses a pattern you saw in main.rs:

```
walker.build_parallel().run(|| {
    // This closure is called once per thread
    // Return a closure that handles individual entries
    Box::new(|entry| {
        // Process entry
        WalkState::Continue
    })
})
```

The outer closure sets up thread-local state. The inner closure processes entries. This enables per-thread resources without shared mutable state.

### Rule Hierarchy

Ignore rules stack as you descend directories:

```
/project/.gitignore          # Applies to all of /project
/project/src/.gitignore      # Adds rules for /project/src
/project/src/test/.gitignore # Adds rules for /project/src/test
```

Child rules take precedence over parent rules. A whitelist in a child directory can override an ignore in a parent.

---

## How Ripgrep Uses This Crate

From main.rs and hiargs.rs:

```
// Building the walker
let walker = WalkBuilder::new(path)
    .hidden(!show_hidden)           // Skip hidden files?
    .ignore(!no_ignore)             // Respect .ignore files?
    .git_ignore(!no_git_ignore)     // Respect .gitignore?
    .git_global(!no_git_global)     // Respect global gitignore?
    .git_exclude(!no_git_exclude)   // Respect .git/info/exclude?
    .types(type_matcher)            // File type filtering
    .overrides(override_matcher)    // CLI glob overrides
    .threads(thread_count)          // Parallelism level
    .build_parallel();              // Construct parallel walker

// Running in parallel (from main.rs)
walker.run(|| {
    let mut searcher = searcher.clone();
    Box::new(move |entry| {
        // Search this file
        match searcher.search(&entry) { ... }
        WalkState::Continue
    })
});
```

---

## Reading Order Recommendation

For the deepest understanding, read in this order:

1. **lib.rs** — Understand `Error` and `Match` first
2. **gitignore.rs** — Core pattern matching
3. **types.rs** + **default_types.rs** — File type system
4. **overrides.rs** — CLI overrides (builds on gitignore)
5. **dir.rs** — How rules accumulate per-directory
6. **walk.rs** — The actual traversal (ties everything together)

Alternatively, for a top-down view: 1. **walk.rs** — See the big picture first 2. Then dive into components as questions arise

---

## What You'll Learn

By the end of this chapter, you'll understand:

- How gitignore pattern matching actually works
- Why parallel directory traversal is non-trivial
- The closure-factory pattern for thread-local state
- How ignore rules cascade through directory hierarchies
- Why file type filtering happens during traversal, not after
- The design decisions that make ripgrep fast at file discovery

Let's begin with the foundation: **lib.rs and the Error/Match types**.

# Ignore Crate lib.rs: Code Companion

Reference code for the ignore crate lib.rs lecture. Sections correspond to the lecture document.

---

## Section 1: Crate Overview

```rust
/*!
The ignore crate provides a fast recursive directory iterator that respects
various filters such as globs, file types and `.gitignore` files. The precise
matching rules and precedence is explained in the documentation for
`WalkBuilder`.
*/

#![deny(missing_docs)]

use std::path::{Path, PathBuf};

// Re-exports from walk module - the main public API
pub use crate::walk::{
    DirEntry, ParallelVisitor, ParallelVisitorBuilder, Walk, WalkBuilder,
    WalkParallel, WalkState,
};

// Module declarations
mod default_types;  // Built-in file type definitions
mod dir;            // Per-directory ignore state
pub mod gitignore;  // Gitignore parsing (public)
pub mod overrides;  // CLI glob overrides (public)
mod pathutil;       // Path utilities (private)
pub mod types;      // File type matching (public)
mod walk;           // Directory traversal (items re-exported above)
```

**Simple usage:**

```rust
use ignore::Walk;

for result in Walk::new("./") {
    match result {
        Ok(entry) => println!("{}", entry.path().display()),
        Err(err) => println!("ERROR: {}", err),
    }
}
```

**Advanced usage:**

```
use ignore::WalkBuilder;

for result in WalkBuilder::new("./")
    .hidden(false)      // Include hidden files
    .git_ignore(true)   // Respect .gitignore
    .build()
{
    println!("{:?}", result);
}
```

## Section 2: The Error Enum — Variant Overview

```rust
/// Represents an error that can occur when parsing a gitignore file.
#[derive(Debug)]
pub enum Error {
    /// A collection of "soft" errors. These occur when adding an ignore
    /// file partially succeeded.
    Partial(Vec<Error>),

    /// An error associated with a specific line number.
    WithLineNumber {
        line: u64,
        err: Box<Error>,
    },

    /// An error associated with a particular file path.
    WithPath {
        path: PathBuf,
        err: Box<Error>,
    },

    /// An error associated with a particular directory depth.
    WithDepth {
        depth: usize,
        err: Box<Error>,
    },

    /// File system loop detected when following symbolic links.
    Loop {
        ancestor: PathBuf,
        child: PathBuf,
    },

    /// An I/O error.
    Io(std::io::Error),

    /// Glob parsing error.
    Glob {
        glob: Option<String>,
        err: String,
    },

    /// A type selection for a file type that is not defined.
    UnrecognizedFileType(String),

    /// A user specified file type definition could not be parsed.
    InvalidDefinition,
}
```

**Error variant purposes:**

| Variant | When Used |
|---|---|

| | |
|---|---|
| `Partial` | Some operations succeed, others fail |
| `WithLineNumber` | Error in a specific line of a file |
| `WithPath` | Error associated with a file path |
| `WithDepth` | Error at a specific directory depth |
| `Loop` | Symlink cycle detected |
| `Io` | Standard I/O operations |
| `Glob` | Invalid glob pattern syntax |
| `UnrecognizedFileType` | Unknown `-t` argument |
| `InvalidDefinition` | Bad `--type-add` format |

## Section 3: Error Context Wrapping

```
// Example of nested error context:
// Error structure for a bad glob on line 5 of .gitignore:

Error::WithPath {
    path: PathBuf::from(".gitignore"),
    err: Box::new(Error::WithLineNumber {
        line: 5,
        err: Box::new(Error::Glob {
            glob: Some("**[bad".to_string()),
            err: "unclosed character class".to_string(),
        }),
    }),
}

// When displayed: ".gitignore: line 5: error parsing glob '**[bad': unclosed
character class"
```

**Why Box:**

```rust
// Without Box, this would be infinite size:
enum Error {
    WithPath { path: PathBuf, err: Error },  // Error contains Error contains
Error...
}

// With Box, the size is known:
enum Error {
    WithPath { path: PathBuf, err: Box<Error> },  // Box is pointer-sized
}
```

---

## Section 4: Partial Errors

```rust
impl Error {
    /// Returns true if this is a partial error.
    ///
    /// A partial error occurs when only some operations failed while others
    /// may have succeeded.
    pub fn is_partial(&self) -> bool {
        match *self {
            Error::Partial(_) => true,
            Error::WithLineNumber { ref err, .. } => err.is_partial(),
            Error::WithPath { ref err, .. } => err.is_partial(),
            Error::WithDepth { ref err, .. } => err.is_partial(),
            _ => false,
        }
    }
}
```

**Partial error scenario:**

```
# .gitignore file:
*.log           # Line 1: valid
*.tmp           # Line 2: valid
**[broken       # Line 3: invalid - unclosed bracket
*.bak           # Line 4: valid
```

```rust
// Result: Partial error containing the line 3 error
// But lines 1, 2, 4 are still applied!

Error::Partial(vec![
    Error::WithLineNumber {
        line: 3,
        err: Box::new(Error::Glob { ... }),
    }
])
```

## Section 5: I/O Error Handling

```
impl Error {
    /// Returns true if this error is exclusively an I/O error.
    pub fn is_io(&self) -> bool {
        match *self {
            Error::Partial(ref errs) => errs.len() == 1 && errs[0].is_io(),
            Error::WithLineNumber { ref err, .. } => err.is_io(),
            Error::WithPath { ref err, .. } => err.is_io(),
            Error::WithDepth { ref err, .. } => err.is_io(),
            Error::Loop { .. } => false,
            Error::Io(_) => true,
            Error::Glob { .. } => false,
            Error::UnrecognizedFileType(_) => false,
            Error::InvalidDefinition => false,
        }
    }

    /// Inspect the original std::io::Error if there is one.
    pub fn io_error(&self) -> Option<&std::io::Error> {
        match *self {
            Error::Partial(ref errs) => {
                if errs.len() == 1 {
                    errs[0].io_error()
                } else {
                    None
                }
            }
            Error::WithLineNumber { ref err, .. } => err.io_error(),
            Error::WithPath { ref err, .. } => err.io_error(),
            Error::WithDepth { ref err, .. } => err.is_io(),
            Error::Loop { .. } => None,
            Error::Io(ref err) => Some(err),
            Error::Glob { .. } => None,
            Error::UnrecognizedFileType(_) => None,
            Error::InvalidDefinition => None,
        }
    }
}
```

**Clone implementation for io::Error:**

```
impl Clone for Error {
    fn clone(&self) -> Error {
        match *self {
            // ... other variants ...

            // std::io::Error isn't Clone, but we can reconstruct it
            Error::Io(ref err) => match err.raw_os_error() {
                Some(e) => Error::Io(std::io::Error::from_raw_os_error(e)),
                None => {
                    Error::Io(std::io::Error::new(err.kind(), err.to_string()))
                }
            },

            // ... other variants ...
        }
    }
}
```

---

## Section 6: The PartialErrorBuilder

```rust
#[derive(Debug, Default)]
struct PartialErrorBuilder(Vec<Error>);

impl PartialErrorBuilder {
    /// Push an error unconditionally.
    fn push(&mut self, err: Error) {
        self.0.push(err);
    }

    /// Push an error only if it's not an I/O error.
    fn push_ignore_io(&mut self, err: Error) {
        if !err.is_io() {
            self.push(err);
        }
    }

    /// Push an error if Some, ignore if None.
    fn maybe_push(&mut self, err: Option<Error>) {
        if let Some(err) = err {
            self.push(err);
        }
    }

    /// Push non-IO error if Some.
    fn maybe_push_ignore_io(&mut self, err: Option<Error>) {
        if let Some(err) = err {
            self.push_ignore_io(err);
        }
    }

    /// Convert accumulated errors to Option<Error>.
    /// None if empty, single error if one, Partial if multiple.
    fn into_error_option(mut self) -> Option<Error> {
        if self.0.is_empty() {
            None
        } else if self.0.len() == 1 {
            Some(self.0.pop().unwrap())
        } else {
            Some(Error::Partial(self.0))
        }
    }
}
```

**Usage pattern:**

```
fn process_lines(lines: &[String]) -> Option<Error> {
    let mut errs = PartialErrorBuilder::default();

    for (i, line) in lines.iter().enumerate() {
        if let Err(e) = parse_glob(line) {
            errs.push(Error::WithLineNumber {
                line: i as u64 + 1,
                err: Box::new(e),
            });
        }
    }

    errs.into_error_option()
}
```

## Section 7: The Match Enum — The Filtering Decision

```
/// The result of a glob match.
///
/// The type parameter `T` typically refers to a type that provides more
/// information about a particular match.
#[derive(Clone, Debug)]
pub enum Match<T> {
    /// The path didn't match any glob.
    None,
    /// The highest precedent glob matched indicates the path should be
    /// ignored.
    Ignore(T),
    /// The highest precedent glob matched indicates the path should be
    /// whitelisted.
    Whitelist(T),
}
```

**What T typically contains:**

```
// For gitignore matching:
Match<gitignore::Glob>  // Which glob pattern matched

// For type matching:
Match<types::FileTypeDef>  // Which type definition matched

// For overrides:
Match<overrides::Glob>  // Which override glob matched
```

## Section 8: Match Semantics — Ignore vs Whitelist

```rust
impl<T> Match<T> {
    /// Returns true if the match result didn't match any globs.
    pub fn is_none(&self) -> bool {
        match *self {
            Match::None => true,
            Match::Ignore(_) | Match::Whitelist(_) => false,
        }
    }

    /// Returns true if the match result implies the path should be ignored.
    pub fn is_ignore(&self) -> bool {
        match *self {
            Match::Ignore(_) => true,
            Match::None | Match::Whitelist(_) => false,
        }
    }

    /// Returns true if the match result implies the path should be whitelisted.
    pub fn is_whitelist(&self) -> bool {
        match *self {
            Match::Whitelist(_) => true,
            Match::None | Match::Ignore(_) => false,
        }
    }
}
```

**Gitignore example:**

```
# .gitignore
*.log           # Produces Ignore for foo.log
!important.log  # Produces Whitelist for important.log
```

```
// Matching "debug.log":
// First *.log matches → Ignore
// !important.log doesn't match
// Result: Ignore("*.log")

// Matching "important.log":
// First *.log matches → Ignore
// Then !important.log matches → Whitelist (takes precedence!)
// Result: Whitelist("!important.log")
```

## Section 9: Match Combinators

```rust
impl<T> Match<T> {
    /// Inverts the match: Ignore ↔ Whitelist, None stays None.
    pub fn invert(self) -> Match<T> {
        match self {
            Match::None => Match::None,
            Match::Ignore(t) => Match::Whitelist(t),
            Match::Whitelist(t) => Match::Ignore(t),
        }
    }

    /// Return the value inside this match if it exists.
    pub fn inner(&self) -> Option<&T> {
        match *self {
            Match::None => None,
            Match::Ignore(ref t) => Some(t),
            Match::Whitelist(ref t) => Some(t),
        }
    }

    /// Apply a function to the inner value.
    pub fn map<U, F: FnOnce(T) -> U>(self, f: F) -> Match<U> {
        match self {
            Match::None => Match::None,
            Match::Ignore(t) => Match::Ignore(f(t)),
            Match::Whitelist(t) => Match::Whitelist(f(t)),
        }
    }

    /// Return self if not None, otherwise return other.
    pub fn or(self, other: Self) -> Self {
        if self.is_none() { other } else { self }
    }
}
```

**Invert usage (for -T flag):**

```rust
// -t rust: include only rust files
let match_result = types.matched(path);  // Whitelist for .rs files

// -T rust: exclude rust files
let match_result = types.matched(path).invert();  // Now Ignore for .rs files
```

**Or usage (combining matchers):**

```rust
// Check override first, fall back to gitignore
let result = override_match.or(gitignore_match);
```

---

## Section 10: How Match Flows Through the Crate

```rust
// Conceptual flow for deciding whether to yield a file:

fn should_yield(entry: &DirEntry) -> bool {
    let path = entry.path();

    // 1. Check command-line overrides (highest precedence)
    let override_match = overrides.matched(path, entry.is_dir());
    if override_match.is_whitelist() {
        return true;  // Explicit include
    }
    if override_match.is_ignore() {
        return false;  // Explicit exclude
    }

    // 2. Check file types (if any selected)
    if types_selected {
        let type_match = types.matched(path, entry.is_dir());
        if type_match.is_ignore() {
            return false;  // Doesn't match selected types
        }
    }

    // 3. Check gitignore rules
    let ignore_match = gitignore.matched(path, entry.is_dir());
    if ignore_match.is_ignore() {
        return false;
    }

    // 4. Default: yield the file
    true
}
```

**Precedence summary:**

```
Override Whitelist  →  Include (highest)
Override Ignore     →  Exclude
Type Whitelist      →  Include
Type Ignore         →  Exclude
Gitignore Whitelist →  Include
Gitignore Ignore    →  Exclude
No match (None)     →  Include (default)
```

## Quick Reference: Error and Match

```
// Error checking
err.is_partial()     // Contains multiple sub-errors?
err.is_io()          // Is this an I/O error?
err.io_error()       // Get &std::io::Error if present

// Match checking
m.is_none()          // No rule matched?
m.is_ignore()        // Should skip this file?
m.is_whitelist()     // Explicitly included?

// Match transformation
m.invert()           // Flip Ignore ↔ Whitelist
m.inner()            // Get &T if present
m.map(f)             // Transform inner value
m.or(other)          // Fallback if None
```

## Type Aliases You'll See

```
// Common patterns in the crate:

type Result<T> = std::result::Result<T, Error>;

// Match with specific metadata:
type GitignoreMatch = Match<gitignore::Glob>;
type TypeMatch = Match<types::FileTypeDef>;
type OverrideMatch = Match<overrides::Glob>;
```

# Ignore Crate lib.rs: Foundation Types

## What This File Does

The lib.rs file is the ignore crate's entry point. Beyond the standard module declarations and re-exports, it defines two fundamental types that pervade the entire crate: the `Error` enum for error handling and the `Match<T>` enum for representing filtering decisions.

These types establish patterns that every other module follows. Understanding them first makes the rest of the crate much clearer.

---

## Section 1: Crate Overview

The module documentation captures the crate's purpose succinctly: "a fast recursive directory iterator that respects various filters such as globs, file types and `.gitignore` files."

Two usage patterns dominate. The simple `Walk::new("./")` provides an iterator that applies all filtering automatically. The advanced `WalkBuilder` enables fine-grained control over every filtering behavior.

The crate exposes three public modules beyond the walker: `gitignore` for parsing and matching gitignore patterns, `types` for file type definitions, and `overrides` for command-line glob handling. Internal modules (`dir`, `pathutil`, `default_types`) remain private implementation details.

See: Companion Code Section 1

---

## Section 2: The Error Enum — Variant Overview

The Error enum handles every failure mode in the crate. Eight variants cover the cases:

`Partial` wraps a vector of errors. This handles situations where some operations succeed while others fail — like parsing a gitignore file where some lines are valid and others aren't.

`WithLineNumber`, `WithPath`, and `WithDepth` wrap inner errors with additional context. The wrapping is recursive — you can have an error with a path containing an error with a line number.

`Loop` occurs when following symlinks creates a cycle. The ancestor and child paths identify where the loop was detected.

`Io` wraps standard I/O errors. `Glob` wraps glob parsing failures. `UnrecognizedFileType` and `InvalidDefinition` handle file type configuration errors.

See: Companion Code Section 2

---

## Section 3: Error Context Wrapping

The `With*` variants demonstrate a pattern for adding context to errors without losing the original. Rather than converting errors to strings early, you wrap them in increasingly specific context.

A glob parse error in a gitignore file becomes: `WithPath { path: ".gitignore", err: WithLineNumber { line: 5, err: Glob { ... } } }`. When displayed, this produces: ".gitignore: line 5: error parsing glob...".

The wrapping uses `Box<Error>` for the inner error. This prevents infinite size from recursive types. Without boxing, `Error` would contain `Error` which contains `Error` — the compiler can't determine a finite size.

See: Companion Code Section 3

---

## Section 4: Partial Errors

The Partial variant handles a subtle situation: what do you do when parsing a file partially succeeds?

Consider a gitignore file with ten lines. Nine parse correctly; one has invalid syntax. Failing entirely discards nine valid rules. Silently ignoring the bad line hides a real problem. The Partial variant reports all errors while letting valid rules work.

The `is_partial` method recursively checks whether any error in the chain is partial. This lets callers decide how to handle partial success — some might treat it as failure, others as warning-worthy success.

See: Companion Code Section 4

---

## Section 5: I/O Error Handling

I/O errors receive special treatment because they're so common and because callers often need to inspect them.

The `is_io` method checks whether an error is purely an I/O error. Like `is_partial`, it recurses through wrappers. A WithPath containing an Io is still "an I/O error" for this purpose.

The `io_error` method provides borrowed access to the underlying `std::io::Error` if one exists. This enables checking error kinds without ownership transfer. The companion `into_io_error` consumes self for cases where ownership is needed.

The Clone implementation for Error specially handles the Io case. std::io::Error isn't Clone, but you can reconstruct one from the raw OS error code or by copying the kind and message.

See: Companion Code Section 5

---

## Section 6: The PartialErrorBuilder

The PartialErrorBuilder is a private helper for accumulating errors during operations that can partially fail.

Its methods (`push`, `push_ignore_io`, `maybe_push`) provide convenient ways to accumulate errors. The `push_ignore_io` variant silently drops I/O errors — useful when I/O failures should be warnings rather than errors.

The `into_error_option` method collapses the accumulated errors. No errors returns None. One error returns that error directly. Multiple errors wraps them in Partial.

This builder appears throughout the crate. Any operation that processes multiple items and can fail per-item uses it.

See: Companion Code Section 6

---

## Section 7: The Match Enum — The Filtering Decision

The Match enum is how the ignore crate represents filtering decisions. Every pattern match, every type check, every override evaluation returns a Match.

Three variants cover all cases: `None` means no rule matched. `Ignore` means a rule says to skip this file. `Whitelist` means a rule explicitly includes this file.

The type parameter T typically carries metadata about which rule matched. For gitignore, it identifies the specific file and glob. This enables debugging: you can trace exactly why a file was ignored.

See: Companion Code Section 7

---

## Section 8: Match Semantics — Ignore vs Whitelist

The Ignore/Whitelist distinction handles negation patterns. In gitignore, lines starting with `!` negate previous patterns:

```
*.log          # Ignore all .log files
!important.log # But not this one
```

The first pattern produces `Ignore`. The second produces `Whitelist`. When evaluating a path, later matches take precedence. A Whitelist after an Ignore means "include this file."

The `is_ignore` and `is_whitelist` methods check the variant. Note that `None` is neither — it means no rule had an opinion about this file.

See: Companion Code Section 8

## Section 9: Match Combinators

Match provides several combinator methods for working with filtering results.

The `invert` method flips Ignore to Whitelist and vice versa. This handles the `-T` (type negation) flag — selecting all files that don't match a type.

The `inner` method provides access to the wrapped value, returning `Option<&T>`. This enables inspecting match metadata without consuming the Match.

The `map` method transforms the inner value, enabling type conversions while preserving the match variant.

The `or` method provides fallback logic. If self is None, return other. Otherwise return self. This implements "first match wins" semantics when combining multiple matchers.

See: Companion Code Section 9

---

## Section 10: How Match Flows Through the Crate

Understanding Match prepares you for the rest of the crate. Every filtering layer returns Match:

- Gitignore matching: does this path match a gitignore pattern?
- Type matching: does this path match selected file types?
- Override matching: does this path match a command-line glob?

These get combined with precedence rules. Overrides beat gitignore. Explicit includes beat excludes. The final Match determines whether the walker yields the entry.

The flow roughly looks like: 1. Check overrides → Match 2. Check file types → Match 3. Check gitignore rules → Match 4. Combine with precedence → final decision

See: Companion Code Section 10

---

## Key Takeaways

First, errors wrap with context rather than converting to strings. This preserves information for callers who need it.

Second, partial errors handle the common case where some operations succeed and others fail. This is especially important for configuration files.

Third, Match encodes filtering decisions with three states: ignore, whitelist, and no-opinion. The distinction enables negation patterns.

Fourth, Match combinators enable composing multiple filtering layers with clear precedence semantics.

Fifth, the generic parameter T in Match carries provenance information. You can trace exactly which rule caused which decision.

---

## What to Read Next

With Error and Match understood, proceed to:

**gitignore.rs** — See how Match gets produced from gitignore pattern matching.

**types.rs** — See how Match gets produced from file type filtering.

**dir.rs** — See how multiple Match results get combined with precedence.

# Ripgrep Autopsy: Anatomy of an Idiomatic Rust CLI Application

> *"ripgrep was initially a larger pile of tightly coupled code; it did not start out with most of its logic separated into crates."*
> — BurntSushi (Andrew Gallant)

## Overview

Ripgrep (rg) is one of the most studied, well-maintained Rust codebases in existence. Written by Andrew Gallant, who also authored the `regex`, `memchr`, and many other foundational Rust crates, it represents a masterclass in:

- Workspace-based modular architecture
- Separation of concerns via crate boundaries
- Performance-first design without sacrificing correctness
- Idiomatic error handling
- Cross-platform CLI development

This autopsy examines ripgrep's architecture from both high and low levels, extracting patterns and idioms applicable to any serious Rust application.

---

## 1. High-Level Architecture

### 1.1 The Workspace Pattern

Ripgrep uses Cargo's workspace feature to organize code into focused, reusable crates:

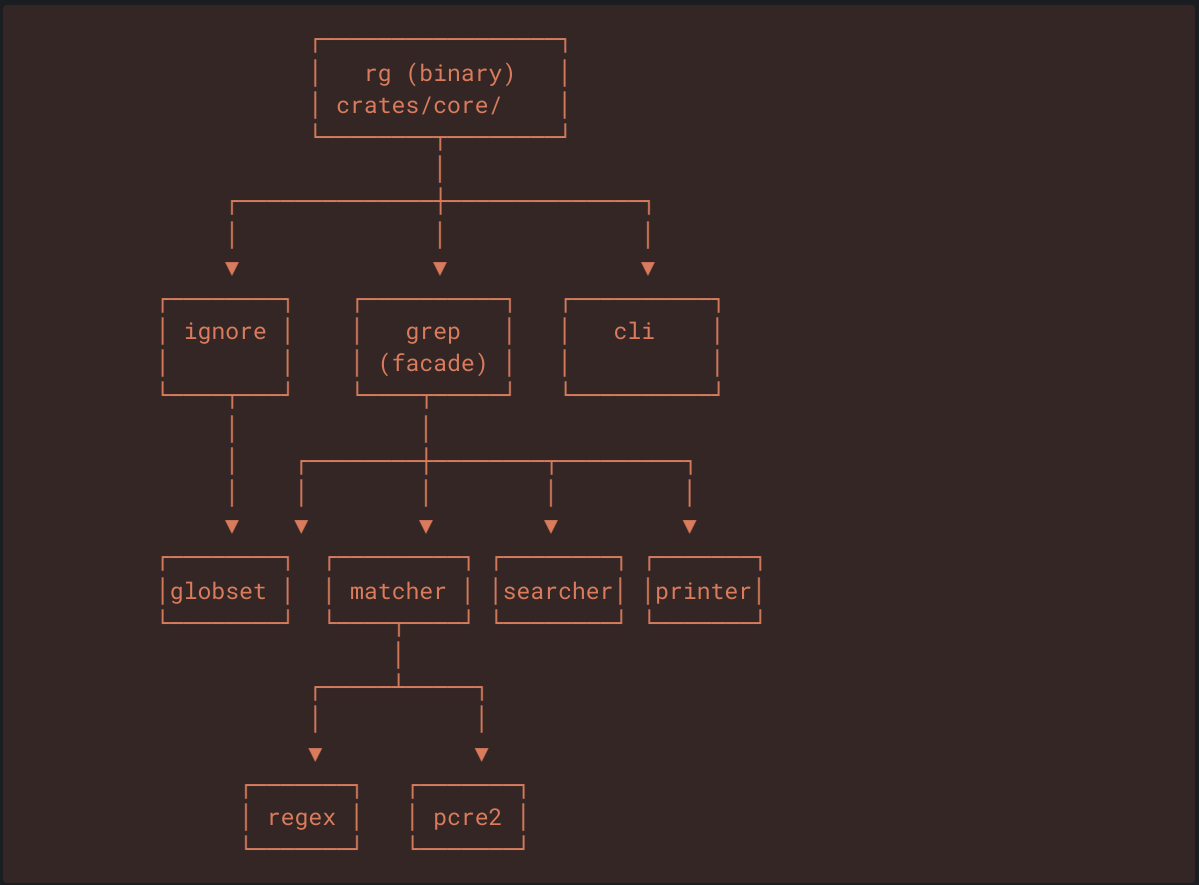
```
# From Cargo.toml
[workspace]
members = [
  "crates/globset",     # Glob pattern matching
  "crates/grep",        # Facade crate
  "crates/cli",         # CLI utilities
  "crates/matcher",     # Abstract matching trait
  "crates/pcre2",       # PCRE2 regex engine
  "crates/printer",     # Output formatting
  "crates/regex",       # Rust regex integration
  "crates/searcher",    # File searching logic
  "crates/ignore",      # .gitignore handling
]
```

**Key Insight**: The main binary is defined separately:

```
[[bin]]
bench = false
path = "crates/core/main.rs"
name = "rg"
```

This means the `crates/core/` directory contains the CLI application itself, while all the library logic lives in separate, independently testable crates.

## 1.2 The Dependency Graph




## 1.3 The Facade Pattern (`grep` crate)

The `grep` crate doesn't contain significant logic — it's a **facade** that re-exports functionality from sub-crates:

```
// crates/grep/src/lib.rs
pub use grep_matcher::*;
pub use grep_printer::*;
pub use grep_regex::*;
pub use grep_searcher::*;
```

**Why This Pattern?**

1. **Simplifies downstream usage** — Users depend on one crate instead of four
2. **Hides internal organization** — Implementation can be refactored without breaking API
3. **Version coordination** — All sub-crates move in lockstep

# 2. Core Abstractions

## 2.1 The `Matcher` Trait

The cornerstone of ripgrep's extensibility is the `Matcher` trait in `grep-matcher`:

```
pub trait Matcher {
    type Captures: Captures;
    type Error: std::error::Error;

    fn find_at(&self, haystack: &[u8], at: usize) -> Result<Option<Match>,
Self::Error>;
    fn new_captures(&self) -> Result<Self::Captures, Self::Error>;
    fn capture_count(&self) -> usize;
    fn line_terminator(&self) -> Option<LineTerminator>;

    // ... more methods
}
```

**Key Design Decisions:**

1. **Works on `&[u8]`**, not `&str` — Handles non-UTF8 files gracefully
2. **Associated types for flexibility** — Different matchers can have different capture/error types
3. **Optional line terminator** — Enables `--null-data` mode where `\0` is the line terminator

## 2.2 The `Sink` Trait

Output formatting is abstracted through the `Sink` trait:

```
pub trait Sink {
    type Error;

    fn matched(&mut self, searcher: &Searcher, mat: &SinkMatch<'_>) ->
Result<bool, Self::Error>;
    fn context(&mut self, searcher: &Searcher, context: &SinkContext<'_>) ->
Result<bool, Self::Error>;
    fn finish(&mut self, searcher: &Searcher, sink_finish: &SinkFinish) ->
Result<(), Self::Error>;
    // ...
}
```

This enables: - Standard grep-like output - JSON Lines output - Custom output formats without modifying search logic

### 2.3 Searcher Configuration

The `Searcher` uses the **builder pattern**:

```
let mut searcher = SearcherBuilder::new()
    .line_number(true)
    .heap_limit(Some(0))
    .build();
```

---

## 3. The CLI Architecture

### 3.1 Two-Level Argument Parsing

Ripgrep uses a sophisticated two-level argument system:




**`LowArgs`**: Mirrors CLI flags directly

```rust
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
pub enum Command {
    Search,
    SearchParallel,
    SearchNever,
    Files,
    FilesParallel,
    TypeList,
    PCRE2Version,
    RipgrepVersion,
    VersionLong,
}
```

`HiArgs`: High-level, validated configuration with computed values

```rust
pub struct HiArgs {
    // Configuration fields computed from LowArgs
    // Handles precedence, defaults, config files, env vars
}

impl HiArgs {
    pub fn matcher(&self) -> Result<impl Matcher>;
    pub fn searcher(&self) -> SearcherBuilder;
    pub fn printer(&self, mode: SearchMode, wtr: impl WriteColor) -> impl Sink;
    pub fn walk_builder(&self) -> Result<WalkBuilder>;
}
```

## 3.2 The `Haystack` Abstraction

```rust
// crates/core/haystack.rs
pub struct Haystack {
    // Represents something to search over (file, stdin, etc.)
}
```

This abstraction normalizes: - Regular files - Standard input - Files from directory walking

## 3.3 The Search Worker

```rust
// crates/core/search.rs
pub struct SearchWorker<M, S, P> {
    matcher: M,
    searcher: S,
    printer: P,
}
```

The `SearchWorker` coordinates: 1. **Matcher** — Pattern matching engine 2. **Searcher** — File reading and searching 3. **Printer** — Output formatting

---

# 4. Performance Patterns

## 4.1 Memory-Mapped vs Buffered I/O

Ripgrep dynamically chooses search strategy:

```
// Pseudo-code from search logic
if single_large_file && file_fits_in_memory {
    use memory_map();  // Faster for big single files
} else {
    use_incremental_buffer();  // Better for many small files
}
```

## 4.2 Parallel Directory Walking

The `ignore` crate provides lock-free parallel directory traversal:

```
args.walk_builder()?.build_parallel().run(|| {
    // This closure is called on multiple threads
    let mut searcher = searcher.clone();

    Box::new(move |entry| {
        // Process each file in parallel
        WalkState::Continue
    })
});
```

**Key Pattern**: `AtomicBool` for coordination:

```
let matched = AtomicBool::new(false);
let searched = AtomicBool::new(false);

// In worker threads:
matched.store(true, Ordering::SeqCst);
```

## 4.3 Literal Optimization

The `grep-regex` crate extracts literals for fast pre-filtering:

```
// crates/regex/src/literal.rs
// Extracts literal substrings from regex patterns
// Uses memchr for SIMD-accelerated byte searching
```

If the pattern is `foo.*bar`, ripgrep can use `memchr` to find `foo` first, then only run the full regex on promising lines.

### 4.4 Smart Defaults

```
// Only count lines if we're going to display them
SearcherBuilder::new().line_number(needs_line_numbers).build()
```

Counting lines is fast but not free — ripgrep only pays the cost when needed.

---

## 5. Error Handling Idioms

### 5.1 The `anyhow` Crate

Ripgrep uses `anyhow` for top-level error handling:

```
// crates/core/main.rs
fn main() {
    if let Err(err) = run() {
        eprintln!("{:#}", err);
        std::process::exit(1);
    }
}

fn run() -> anyhow::Result<()> {
    // Application logic with ? operator
}
```

### 5.2 Custom Error Types in Libraries

Library crates define specific error types:

```rust
// In grep-regex
pub struct Error {
    kind: ErrorKind,
}

pub enum ErrorKind {
    Regex(regex::Error),
    // ... other variants
}
```

### 5.3 Result Type Aliases

```rust
// Common pattern in each crate
pub type Result<T> = std::result::Result<T, Error>;
```

---

## 6. Cross-Platform Patterns

### 6.1 Conditional Compilation

```rust
#[cfg(all(target_env = "musl", target_pointer_width = "64"))]
use tikv_jemallocator;

#[cfg(windows)]
mod windows_specific;
```

### 6.2 Platform-Specific Allocators

```toml
# In Cargo.toml
[target.'cfg(all(target_env = "musl", target_pointer_width =
"64"))'.dependencies.tikv-jemallocator]
version = "0.6.0"
```

Musl builds use jemalloc because musl's allocator is slower for ripgrep's workload.

### 6.3 Build Script (`build.rs`)

```rust
fn main() {
    set_git_revision_hash();
    set_windows_exe_options();
}

fn set_git_revision_hash() {
    // Embeds git hash at compile time via RIPGREP_BUILD_GIT_HASH
}

fn set_windows_exe_options() {
    // Enables long path support on Windows
    // Embeds manifest file
}
```

# 7. Testing Patterns

## 7.1 Integration Tests

```toml
[[test]]
name = "integration"
path = "tests/tests.rs"
```

```rust
// tests/tests.rs
mod binary;
mod feature;
mod json;
mod misc;
mod multiline;
mod regression;
```

## 7.2 In-Crate Unit Tests

Each library crate has internal tests:

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

    #[test]
    fn config_error_heap_limit() {
        let matcher = RegexMatcher::new("");
        let sink = KitchenSink::new();
        let mut searcher = SearcherBuilder::new().heap_limit(Some(0)).build();
        let res = searcher.search_slice(matcher, &[], sink);
        assert!(res.is_err());
    }
}
```

### 7.3 Test Utilities

```
// tests/util.rs
pub struct Dir {
    // Temporary directory for testing
}

impl Dir {
    pub fn create_file(&self, name: &str, contents: &str);
    pub fn rg(&self) -> Command;
}
```

## 8. Configuration System

### 8.1 Environment Variables

```
// RIPGREP_CONFIG_PATH for config file location
// NO_COLOR for disabling colors
// TERM for terminal detection
```

### 8.2 Configuration File Format

```
# ~/.ripgreprc
--max-columns=150
--smart-case
--type-add
web:*.{html,css,js}
```

**Key Constraint**: No escaping, one argument per line.

### 8.3 Configuration Precedence

```
1. Built-in defaults (lowest)
2. Configuration file (via RIPGREP_CONFIG_PATH)
3. Command-line arguments (highest, last flag wins)
```

---

## 9. Code Style and Conventions

### 9.1 rustfmt Configuration

```
# rustfmt.toml
max_width = 79
use_small_heuristics = "max"
edition = "2024"
```

**79 columns** — Stricter than Rust default (100), enables side-by-side diffs.

### 9.2 Documentation Style

Every public item has documentation:

```
/// The command that ripgrep should execute based on the command line
/// configuration.
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
pub enum Command {
    /// Search using exactly one thread.
    Search,
    /// Search using possibly many threads.
    SearchParallel,
    // ...
}
```

### 9.3 Derive Attributes

Standard derives for configuration types:

```
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
```

## 10. Key Takeaways for Your Own Projects

### 10.1 Architectural Patterns

1. **Workspace for modularity** — Separate concerns into crates
2. **Facade crates** — Simplify external dependencies
3. **Trait-based abstraction** — Enable extensibility without coupling
4. **Builder pattern** — Complex configuration with sensible defaults

### 10.2 Performance Patterns

1. **Work on bytes (`&[u8]`)** — Don't require UTF-8 when unnecessary
2. **Conditional optimization** — Choose strategy based on workload
3. **Parallel by default** — Use `crossbeam` for lock-free parallelism
4. **Literal extraction** — Pre-filter with fast substring search

### 10.3 CLI Patterns

1. **Two-level argument parsing** — Raw flags → Validated configuration
2. **Configuration file support** — Respects environment and dotfiles
3. **Graceful degradation** — Works on non-UTF8 files, handles errors cleanly

### 10.4 Code Organization

1. **Binary separate from library** — `crates/core/main.rs` imports library crates
2. **Associated types in traits** — Avoid boxing, enable optimization
3. **Platform abstraction** — Conditional compilation for portability

## Source Files Reference

| | |
|---|---|
| `crates/core/` | The `rg` binary — CLI, argument handling, search orchestration |
| `crates/grep/` | Facade re-exporting matcher, printer, regex, searcher |
| `crates/matcher/` | Abstract `Matcher` trait |
| `crates/regex/` | Rust regex implementation of `Matcher` |
| `crates/pcre2/` | PCRE2 implementation of `Matcher` (optional) |
| `crates/searcher/` | File searching with buffered/mmap strategies |
| `crates/printer/` | Output formatting (standard, JSON) |
| `crates/ignore/` | Parallel directory walking with .gitignore support |
| `crates/globset/` | Fast glob pattern matching |
| `crates/cli/` | CLI utilities (colors, decompression, process handling) |

## Further Reading

1. **BurntSushi's blog**: https://blog.burntsushi.net/ripgrep/
2. **Regex internals**: https://blog.burntsushi.net/regex-internals/
3. **Repository**: https://github.com/BurntSushi/ripgrep

---

*"Your performance intuition is useless. Run perf."*
— Comment in Rust's `layout.rs`, but equally applicable here

# Ripgrep haystack.rs: Code Companion

Reference code for the haystack.rs lecture. Sections correspond to the lecture document.

---

## Section 1: The Builder Pattern

```rust
/// A builder for constructing things to search over.
#[derive(Clone, Debug)]
pub(crate) struct HaystackBuilder {
    strip_dot_prefix: bool,
}

impl HaystackBuilder {
    /// Return a new haystack builder with a default configuration.
    pub(crate) fn new() -> HaystackBuilder {
        HaystackBuilder { strip_dot_prefix: false }
    }

    /// When enabled, if the haystack's file path starts with `./` then it is
    /// stripped.
    ///
    /// This is useful when implicitly searching the current working directory.
    pub(crate) fn strip_dot_prefix(
        &mut self,
        yes: bool,
    ) -> &mut HaystackBuilder {
        self.strip_dot_prefix = yes;
        self
    }
}
```

**Usage in HiArgs:**

```rust
// From hiargs.rs
pub(crate) fn haystack_builder(&self) -> HaystackBuilder {
    let mut builder = HaystackBuilder::new();
    builder.strip_dot_prefix(self.paths.has_implicit_path);
    builder
}
```

**Why a builder for one field?**

- Consistent API pattern across ripgrep
- Room for future configuration
- Separates construction from usage
- Builder is `Clone` — can be shared across threads

---

## Section 2: Building from Results

```rust
impl HaystackBuilder {
    /// Create a new haystack from a possibly missing directory entry.
    ///
    /// If the directory entry isn't present, then the corresponding error is
    /// logged if messages have been configured. Otherwise, if the directory
    /// entry is deemed searchable, then it is returned as a haystack.
    pub(crate) fn build_from_result(
        &self,
        result: Result<ignore::DirEntry, ignore::Error>,
    ) -> Option<Haystack> {
        match result {
            Ok(dent) => self.build(dent),
            Err(err) => {
                err_message!("{err}");
                None
            }
        }
    }
}
```

**How it's used in main.rs:**

```rust
// Single-threaded search
let unsorted = args
    .walk_builder()?
    .build()
    .filter_map(|result| haystack_builder.build_from_result(result));

// Parallel search
Box::new(move |result| {
    let haystack = match haystack_builder.build_from_result(result) {
        Some(haystack) => haystack,
        None => return WalkState::Continue,  // Skip, keep walking
    };
    // ... search the haystack
})
```

**Error handling flow:**

```
Walker yields Result<DirEntry, Error>
           │
           ├── Err(e) → Log error, return None
           │
           └── Ok(dent) → Apply filtering logic
                                │
                                ├── Passes → Some(Haystack)
                                │
                                └── Fails → None
```

## Section 3: The Filtering Decision

```
impl HaystackBuilder {
    /// Create a new haystack using this builder's configuration.
    ///
    /// If a directory entry could not be created or should otherwise not be
    /// searched, then this returns `None` after emitting any relevant log
    /// messages.
    fn build(&self, dent: ignore::DirEntry) -> Option<Haystack> {
        let hay = Haystack {
            dent,
            strip_dot_prefix: self.strip_dot_prefix
        };

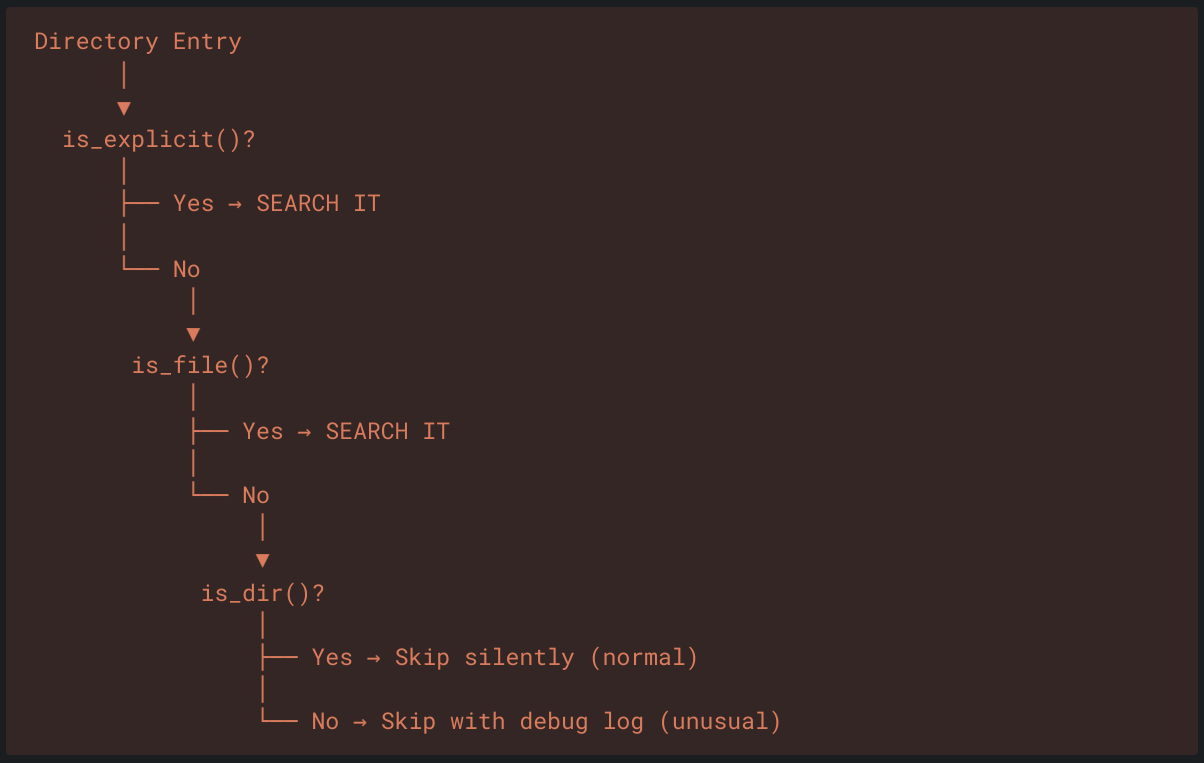
        // Log any partial errors (e.g., metadata read failures)
        if let Some(err) = hay.dent.error() {
            ignore_message!("{err}");
        }

        // Rule 1: Explicit entries always pass
        if hay.is_explicit() {
            return Some(hay);
        }

        // Rule 2: Only search regular files
        if hay.is_file() {
            return Some(hay);
        }

        // Rule 3: Everything else is skipped
        if !hay.is_dir() {
            log::debug!(
                "ignoring {}: failed to pass haystack filter: \
                 file type: {:?}, metadata: {:?}",
                hay.dent.path().display(),
                hay.dent.file_type(),
                hay.dent.metadata()
            );
        }
        None
    }
}
```

**Decision flowchart:**




## Section 4: Explicit vs Implicit

```rust
impl Haystack {
    /// Returns true if and only if this entry corresponds to a haystack to
    /// search that was explicitly supplied by an end user.
    ///
    /// Generally, this corresponds to either stdin or an explicit file path
    /// argument. e.g., in `rg foo some-file ./some-dir/`, `some-file` is
    /// an explicit haystack, but, e.g., `./some-dir/some-other-file` is not.
    ///
    /// However, note that ripgrep does not see through shell globbing. e.g.,
    /// in `rg foo ./some-dir/*`, `./some-dir/some-other-file` will be treated
    /// as an explicit haystack.
    pub(crate) fn is_explicit(&self) -> bool {
        // stdin is always explicit
        // depth() == 0 means directly provided, not discovered
        // !is_dir() because directories are never "searched"
        self.is_stdin() || (self.dent.depth() == 0 && !self.is_dir())
    }

    /// Returns true if and only if this entry corresponds to stdin.
    pub(crate) fn is_stdin(&self) -> bool {
        self.dent.is_stdin()
    }
}
```

**Examples:**

| Command | Path | Explicit? | Why |
|---|---|---|---|
| `rg foo file.txt` | `file.txt` | Yes | depth=0, user provided |
| `rg foo ./src/` | `./src/main.rs` | No | depth>0, discovered |
| `rg foo` | `src/main.rs` | No | depth>0, discovered |
| `rg foo -` | stdin | Yes | stdin is always explicit |
| `echo x \| rg foo` | stdin | Yes | stdin is always explicit |
| `rg foo ./src/*` | `./src/lib.rs` | Yes | shell expanded, depth=0 |

**Why depth=0 means explicit:**

```
// When you run: rg foo file1.txt ./dir/
// The walker sees:
//   file1.txt     → depth 0 (you provided it)
//   ./dir/        → depth 0 (you provided it)
//   ./dir/a.rs    → depth 1 (walker found it)
//   ./dir/sub/b.rs → depth 2 (walker found it)
```

## Section 5: Path Display

```
impl Haystack {
    /// Return the file path corresponding to this haystack.
    ///
    /// If this haystack corresponds to stdin, then a special `<stdin>` path
    /// is returned instead.
    pub(crate) fn path(&self) -> &Path {
        if self.strip_dot_prefix && self.dent.path().starts_with("./") {
            self.dent.path().strip_prefix("./").unwrap()
        } else {
            self.dent.path()
        }
    }
}
```

**Behavior examples:**

| Input | strip_dot_prefix | path() returns |
|---|---|---|
| `./src/main.rs` | `true` | `src/main.rs` |
| `./src/main.rs` | `false` | `./src/main.rs` |
| `src/main.rs` | `true` | `src/main.rs` |
| `src/main.rs` | `false` | `src/main.rs` |
| `-` (stdin) | either | `<stdin>` (from DirEntry) |

**When strip_dot_prefix is enabled:**

```
// In hiargs.rs
builder.strip_dot_prefix(self.paths.has_implicit_path);

// has_implicit_path is true when user ran `rg foo` with no paths
// This makes output cleaner:
//   "src/main.rs:42:fn main()"
// Instead of:
//   "./src/main.rs:42:fn main()"
```

## Section 6: File Type Detection

```rust
impl Haystack {
    /// Returns true if and only if this haystack points to a directory after
    /// following symbolic links.
    fn is_dir(&self) -> bool {
        let ft = match self.dent.file_type() {
            None => return false,  // Can't determine = not a dir
            Some(ft) => ft,
        };
        if ft.is_dir() {
            return true;
        }
        // Symlink that points to a directory?
        self.dent.path_is_symlink() && self.dent.path().is_dir()
    }

    /// Returns true if and only if this haystack points to a file.
    fn is_file(&self) -> bool {
        self.dent.file_type().map_or(false, |ft| ft.is_file())
    }
}
```

**File type scenarios:**

| Filesystem Object | is_file() | is_dir() | Searchable? |
|---|---|---|---|
| Regular file | true | false | Yes |
| Directory | false | true | No |
| Symlink → file | false | false | Only if explicit |
| Symlink → dir | false | true | No |
| Socket/FIFO/etc | false | false | Only if explicit |
| Unknown (no metadata) | false | false | Only if explicit |

**Why symlink handling differs:**

```
// is_dir follows symlinks:
self.dent.path_is_symlink() && self.dent.path().is_dir()

// is_file does NOT follow symlinks:
self.dent.file_type().map_or(false, |ft| ft.is_file())

// Reason: By the time we get here, --follow has already been applied.
// If the user wanted symlinks followed, the walker already did it.
// A symlink that wasn't followed should not be searched.
```

## Section 7: The Wrapper Structure

```
/// A haystack is a thing we want to search.
///
/// Generally, a haystack is either a file or stdin.
#[derive(Clone, Debug)]
pub(crate) struct Haystack {
    dent: ignore::DirEntry,
    strip_dot_prefix: bool,
}
```

**What ignore::DirEntry provides:**

```
// From the ignore crate (not shown in haystack.rs)
impl DirEntry {
    fn path(&self) -> &Path;
    fn file_type(&self) -> Option<FileType>;
    fn metadata(&self) -> Result<Metadata, Error>;
    fn depth(&self) -> usize;
    fn is_stdin(&self) -> bool;
    fn path_is_symlink(&self) -> bool;
    fn error(&self) -> Option<&Error>;
}
```

**Why wrap instead of using DirEntry directly?**

1. **Custom path logic** — strip_dot_prefix behavior
2. **Application-level filtering** — is_explicit, is_file
3. **Encapsulation** — Search code sees Haystack API, not ignore API
4. **Future flexibility** — Can add fields without changing ignore crate

## Quick Reference: Full API

```
// Builder
impl HaystackBuilder {
    pub fn new() -> HaystackBuilder;
    pub fn strip_dot_prefix(&mut self, yes: bool) -> &mut Self;
    pub fn build_from_result(&self, result: Result<DirEntry, Error>) ->
Option<Haystack>;
}

// Haystack
impl Haystack {
    pub fn path(&self) -> &Path;         // Display path
    pub fn is_stdin(&self) -> bool;      // Is this stdin?
    pub fn is_explicit(&self) -> bool;   // User-provided?
    fn is_dir(&self) -> bool;            // Directory? (private)
    fn is_file(&self) -> bool;           // Regular file? (private)
}
```

## Integration Example

```rust
// Complete flow from main.rs search function

fn search(args: &HiArgs, mode: SearchMode) -> anyhow::Result<bool> {
    // 1. Get builder configured from HiArgs
    let haystack_builder = args.haystack_builder();

    // 2. Create iterator that converts DirEntry → Haystack
    let unsorted = args
        .walk_builder()?          // Configure directory walker
        .build()                  // Create iterator over Results
        .filter_map(|result| {    // Convert each Result
            haystack_builder.build_from_result(result)
        });

    // 3. Optionally sort
    let haystacks = args.sort(unsorted);

    // 4. Search each haystack
    for haystack in haystacks {
        let result = searcher.search(&haystack)?;
        // haystack.path() used for output formatting
    }

    Ok(matched)
}
```

**Data flow:**

```
WalkBuilder
    |
    ▼
Iterator<Item = Result<DirEntry, Error>>
    |
    |  filter_map + build_from_result
    ▼
Iterator<Item = Haystack>
    |
    |  sort (optional)
    ▼
Iterator<Item = Haystack>
    |
    |  search loop
    ▼
Results
```

# Ripgrep haystack.rs: The Search Target Abstraction

## What This File Does

The haystack.rs file defines what ripgrep actually searches. A "haystack" is the thing you're looking for a needle in — it could be a file on disk, stdin, or something discovered during directory traversal.

This is a thin abstraction layer, but an important one. It wraps the directory entry from the ignore crate and adds application-level logic: should this thing be searched? How should its path be displayed? The abstraction normalizes different input sources so the search code doesn't need to care whether it's searching a file the user named explicitly or one discovered recursively.

---

## Section 1: The Builder Pattern

Haystack uses the builder pattern, though with minimal configuration. Currently there's only one setting: whether to strip the "./" prefix from paths.

This might seem like overkill for a single boolean, but the pattern leaves room for future configuration without changing the API. More importantly, it separates construction concerns from usage concerns — the builder knows how to create haystacks, and the haystack knows how to be searched.

The builder is cloneable and stored in HiArgs, then passed to every search function. This ensures consistent haystack construction throughout a search operation.

See: Companion Code Section 1

## Section 2: Building from Results

The build_from_result method is the primary entry point. Directory traversal yields Results because listing a directory can fail — permission denied, symlink loops, and other filesystem errors.

When traversal yields an error, the builder logs it and returns None. The search continues with other files. This is the "non-fatal error" philosophy from main.rs in action — one bad file shouldn't stop the entire search.

When traversal yields a valid directory entry, the builder applies filtering logic to decide if it's searchable. Not everything the walker yields should be searched.

See: Companion Code Section 2

---

## Section 3: The Filtering Decision

The build method contains the core filtering logic. It asks a series of questions about each directory entry.

First, was this entry explicitly provided by the user? If someone types "rg pattern somefile", they want somefile searched regardless of what it is. Explicit entries always pass the filter.

Second, is this a regular file? During directory traversal, the walker yields all kinds of things: files, directories, symlinks. Ripgrep only wants to search files. Directories are obviously not searchable. Symlinks are trickier — by the time we see them, they've either been followed (if --follow was set) or they haven't. If they haven't been followed, we skip them.

If the entry passes neither check, it gets logged at debug level and filtered out. The debug logging helps users understand why certain files aren't being searched.

See: Companion Code Section 3

---

## Section 4: Explicit vs Implicit

The distinction between explicit and implicit entries is subtle but important. It affects both filtering and output formatting.

An explicit entry is something the user directly asked to search. This includes stdin and any path argument with depth zero — meaning it wasn't discovered by recursing into a directory, it was provided directly.

An implicit entry is something discovered during directory traversal. When you run "rg pattern ./src", the ./src directory is explicit, but ./src/main.rs is implicit — ripgrep found it by traversing.

Why does this matter? Explicit files bypass certain filters. If you explicitly name a binary file, ripgrep searches it (with appropriate binary handling). If ripgrep discovers a binary file during traversal, it might skip it entirely. The user's explicit intent overrides automatic filtering.

See: Companion Code Section 4

---

## Section 5: Path Display

The path method handles how file paths appear in output. This seems trivial but involves a subtle UX decision.

When you run "rg pattern" with no arguments, ripgrep searches the current directory. Internally, this becomes "./". Without special handling, every match would show paths like "./src/main.rs".

The strip_dot_prefix option removes this leading "./". So matches appear as "src/main.rs" instead. This matches user expectations — they didn't type "./", so they don't want to see it.

But when the user explicitly provides "./" as an argument, the prefix stays. Their explicit choice is respected.

See: Companion Code Section 5

---

## Section 6: File Type Detection

The is_dir and is_file methods determine what kind of filesystem object we're dealing with. This sounds simple but has edge cases around symlinks.

For directories, the check follows symlinks. If a symlink points to a directory, is_dir returns true. This matters because we never want to "search" a directory — that doesn't make sense.

For files, the check is stricter. Only actual files return true, not symlinks to files. This is because symlink handling has already happened during traversal. If symlinks should be followed, the walker followed them. What we see here is what we should search.

The file_type method from the directory entry can return None if metadata couldn't be read. Both methods handle this gracefully by returning false — if we can't determine what something is, we don't search it.

See: Companion Code Section 6

---

## Section 7: The Wrapper Structure

The Haystack struct itself is minimal: a directory entry plus the strip_dot_prefix flag. It's a wrapper that adds behavior without adding much state.

This wrapper pattern is common in Rust. The underlying ignore::DirEntry has its own API, but we want to present a different API to the rest of ripgrep. The wrapper lets us do that while keeping the original entry accessible for its metadata.

The struct derives Clone and Debug. Clone is needed because haystacks might be sent across threads in parallel search. Debug is useful for logging and debugging.

See: Companion Code Section 7

---

## Key Takeaways

First, abstraction enables uniform handling. The search code doesn't care if it's searching stdin or a file — both are haystacks with the same interface.

Second, explicit user intent matters. Files the user names directly are treated differently than files discovered automatically.

Third, small abstractions have value. This file is under 100 lines, but it cleanly separates "what to search" from "how to search it."

Fourth, error handling is graceful. Filesystem errors during traversal become log messages, not crashes.

Fifth, the builder pattern scales. Even with one configuration option, the pattern makes the API clean and extensible.

---

## What to Read Next

Understanding haystack raises questions about its context:

How does the ignore crate's DirEntry work? Read the ignore crate documentation.

How does HaystackBuilder get configured? Review hiargs.rs, specifically the haystack_builder method.

What does SearchWorker do with a Haystack? Read search.rs.

How do haystacks flow through parallel search? Review the search_parallel function in main.rs.

# Ripgrep hiargs.rs: Code Companion

Reference code for the hiargs.rs lecture. Sections correspond to the lecture document.

---

## Section 1: The HiArgs Struct

```rust
#[derive(Debug)]
pub(crate) struct HiArgs {
    // Display options
    color: ColorChoice,
    colors: grep::printer::ColorSpecs,
    column: bool,
    heading: bool,
    line_number: bool,
    byte_offset: bool,

    // Search behavior
    case: CaseMode,
    boundary: Option<BoundaryMode>,
    fixed_strings: bool,
    invert_match: bool,
    multiline: bool,
    multiline_dotall: bool,

    // What to search
    patterns: Patterns,
    paths: Paths,
    mode: Mode,

    // Filtering
    types: ignore::types::Types,
    globs: ignore::overrides::Override,
    binary: BinaryDetection,
    max_filesize: Option<u64>,
    max_depth: Option<usize>,

    // Performance
    threads: usize,
    mmap_choice: grep::searcher::MmapChoice,
    dfa_size_limit: Option<usize>,
    regex_size_limit: Option<usize>,

    // ... approximately 70 fields total
}
```

**Field categories:**

| Category | Examples |
|---|---|
| Display | `color`, `heading`, `line_number`, `column` |
| Search | `case`, `boundary`, `multiline`, `invert_match` |
| Filtering | `types`, `globs`, `binary`, `hidden` |

| | |
|---|---|
| Performance | `threads`, `mmap_choice`, `*_size_limit` |
| Context | `context`, `context_separator`, `file_separator` |
| Output format | `mode`, `quiet`, `stats`, `vimgrep` |

## Section 2: The Transformation — from_low_args

```rust
impl HiArgs {
    pub(crate) fn from_low_args(mut low: LowArgs) -> anyhow::Result<HiArgs> {
        // Assertion: special modes already handled
        assert_eq!(None, low.special, "special mode demands short-circuiting");

        // Validate sorting support
        if let Some(ref sort) = low.sort {
            sort.supported()?;
        }

        // Mode adjustments for flag interactions
        match low.mode {
            Mode::Search(ref mut mode) => match *mode {
                // -v --count-matches → -v --count
                SearchMode::CountMatches if low.invert_match => {
                    *mode = SearchMode::Count;
                }
                // -o --count → --count-matches
                SearchMode::Count if low.only_matching => {
                    *mode = SearchMode::CountMatches;
                }
                _ => {}
            },
            _ => {}
        }

        // Initialize shared state
        let mut state = State::new()?;

        // Sequential conversions (order matters!)
        let patterns = Patterns::from_low_args(&mut state, &mut low)?;
        let paths = Paths::from_low_args(&mut state, &patterns, &mut low)?;
        let binary = BinaryDetection::from_low_args(&state, &low);
        let colors = take_color_specs(&mut state, &mut low);
        let hyperlink_config = take_hyperlink_config(&mut state, &mut low)?;
        let stats = stats(&low);
        let types = types(&low)?;
        let globs = globs(&state, &low)?;
        let pre_globs = preprocessor_globs(&state, &low)?;

        // Derived values...

        Ok(HiArgs { /* ... */ })
    }
}
```

**Flag interaction examples:**

```
// Invert match + count-matches doesn't make sense semantically
// Counting inverted matches → just count lines without matches
SearchMode::CountMatches if low.invert_match => SearchMode::Count

// Only-matching + count is effectively count-matches
// When only printing matched parts, counting them makes sense
SearchMode::Count if low.only_matching => SearchMode::CountMatches
```

## Section 3: State Management

```
#[derive(Debug)]
struct State {
    /// Is stdout connected to a terminal?
    is_terminal_stdout: bool,
    /// Has stdin already been read (for patterns)?
    stdin_consumed: bool,
    /// Captured at startup, doesn't change
    cwd: PathBuf,
}

impl State {
    fn new() -> anyhow::Result<State> {
        use std::io::IsTerminal;

        let cwd = current_dir()?;
        log::debug!("read CWD from environment: {}", cwd.display());

        Ok(State {
            is_terminal_stdout: std::io::stdout().is_terminal(),
            stdin_consumed: false,
            cwd,
        })
    }
}
```

**How state affects behavior:**

```
// Terminal detection → default colors
let color = match low.color {
    ColorChoice::Auto if !state.is_terminal_stdout => ColorChoice::Never,
    _ => low.color,
};

// Terminal detection → default line numbers
let line_number = low.line_number.unwrap_or_else(|| {
    state.is_terminal_stdout && !paths.is_only_stdin()
});

// Stdin tracking → prevent double consumption
if path == Path::new("-") && state.stdin_consumed {
    anyhow::bail!("stdin has already been consumed");
}
```

---

## Section 4: Pattern Handling

```rust
#[derive(Debug)]
struct Patterns {
    patterns: Vec<String>,
}

impl Patterns {
    fn from_low_args(
        state: &mut State,
        low: &mut LowArgs,
    ) -> anyhow::Result<Patterns> {
        // Non-search mode = no patterns needed
        if !matches!(low.mode, Mode::Search(_)) {
            return Ok(Patterns { patterns: vec![] });
        }

        // No -e/-f flags = first positional is the pattern
        if low.patterns.is_empty() {
            anyhow::ensure!(
                !low.positional.is_empty(),
                "ripgrep requires at least one pattern"
            );
            let ospat = low.positional.remove(0);
            let Ok(pat) = ospat.into_string() else {
                anyhow::bail!("pattern is not valid UTF-8")
            };
            return Ok(Patterns { patterns: vec![pat] });
        }

        // Collect from -e/--regexp and -f/--file with deduplication
        let mut seen = HashSet::new();
        let mut patterns = Vec::with_capacity(low.patterns.len());

        let mut add = |pat: String| {
            if !seen.contains(&pat) {
                seen.insert(pat.clone());
                patterns.push(pat);
            }
        };

        for source in low.patterns.drain(..) {
            match source {
                PatternSource::Regexp(pat) => add(pat),
                PatternSource::File(path) => {
                    if path == Path::new("-") {
                        anyhow::ensure!(
                            !state.stdin_consumed,
                            "stdin has already been consumed"
                        );
                        for pat in grep::cli::patterns_from_stdin()? {
                            add(pat);
                        }
                        state.stdin_consumed = true;  // Mark consumed!
                    } else {
                        for pat in grep::cli::patterns_from_path(&path)? {
```

```
                add(pat);                }
    }           }           }       Ok(Patterns { patterns })    }}
```

**Pattern sources:**

| Source | Example | Notes |
|---|---|---|
| Positional | `rg foo` | First arg when no -e/-f |
| `-e` flag | `rg -e foo -e bar` | Can repeat |
| `-f` flag | `rg -f patterns.txt` | One pattern per line |
| `-f -` | `rg -f -` | Read from stdin |

## Section 5: Path Handling

```rust
#[derive(Debug)]
struct Paths {
    paths: Vec<PathBuf>,
    has_implicit_path: bool,  // Did we guess the path?
    is_one_file: bool,        // Exactly one non-directory?
}

impl Paths {
    fn from_low_args(
        state: &mut State,
        _: &Patterns,  // Ensures patterns processed first
        low: &mut LowArgs,
    ) -> anyhow::Result<Paths> {
        // Explicit paths provided
        let mut paths = Vec::with_capacity(low.positional.len());
        for osarg in low.positional.drain(..) {
            let path = PathBuf::from(osarg);
            if state.stdin_consumed && path == Path::new("-") {
                anyhow::bail!("attempted to search stdin after reading patterns
from it");
            }
            paths.push(path);
        }

        if !paths.is_empty() {
            let is_one_file = paths.len() == 1
                && (paths[0] == Path::new("-") || !paths[0].is_dir());
            return Ok(Paths {
                paths,
                has_implicit_path: false,
                is_one_file
            });
        }

        // No paths: use heuristics
        let is_readable_stdin = grep::cli::is_readable_stdin();
        let use_cwd = !is_readable_stdin
            || state.stdin_consumed
            || matches!(low.mode, Mode::Search(_));

        let (path, is_one_file) = if use_cwd {
            (PathBuf::from("./"), false)
        } else {
            (PathBuf::from("-"), true)  // Search stdin
        };

        Ok(Paths {
            paths: vec![path],
            has_implicit_path: true,
            is_one_file
        })
    }
}
```

**Heuristic decision tree:**

```
No explicit paths given?
├── stdin readable AND not consumed AND search mode?
│   └── Yes → Search stdin
│   └── No  → Search current directory
└── Explicit paths → Use them directly
```

---

## Section 6: Binary Detection

```rust
#[derive(Debug)]
struct BinaryDetection {
    explicit: grep::searcher::BinaryDetection,  // For named files
    implicit: grep::searcher::BinaryDetection,  // For discovered files
}

impl BinaryDetection {
    fn from_low_args(_: &State, low: &LowArgs) -> BinaryDetection {
        let none = matches!(low.binary, BinaryMode::AsText) || low.null_data;
        let convert = matches!(low.binary, BinaryMode::SearchAndSuppress);

        // Explicit files: never quit early
        let explicit = if none {
            grep::searcher::BinaryDetection::none()
        } else {
            grep::searcher::BinaryDetection::convert(b'\x00')
        };

        // Implicit files: can quit early
        let implicit = if none {
            grep::searcher::BinaryDetection::none()
        } else if convert {
            grep::searcher::BinaryDetection::convert(b'\x00')
        } else {
            grep::searcher::BinaryDetection::quit(b'\x00')  // Default!
        };

        BinaryDetection { explicit, implicit }
    }
}
```

**Binary detection modes:**

| Mode | Explicit files | Implicit files |
|---|---|---|

| Default | Convert NUL | Quit on NUL |
| --- | --- | --- |
| `-a/--text` | None | None |
| `--binary` | Convert NUL | Convert NUL |
| `--null-data` | None | None |

## Section 7: Building the Matcher

```
pub(crate) fn matcher(&self) -> anyhow::Result<PatternMatcher> {
    match self.engine {
        EngineChoice::Default => match self.matcher_rust() {
            Ok(m) => Ok(m),
            Err(err) => {
                anyhow::bail!(suggest_other_engine(err.to_string()));
            }
        },
        EngineChoice::PCRE2 => Ok(self.matcher_pcre2()?),
        EngineChoice::Auto => {
            // Try Rust first
            let rust_err = match self.matcher_rust() {
                Ok(m) => return Ok(m),
                Err(err) => err,
            };
            // Fall back to PCRE2
            let pcre_err = match self.matcher_pcre2() {
                Ok(m) => return Ok(m),
                Err(err) => err,
            };
            // Both failed — show both errors
            anyhow::bail!(
                "regex could not be compiled with either engine.\n\n\
                 Rust regex error:\n{rust_err}\n\n\
                 PCRE2 error:\n{pcre_err}"
            );
        }
    }
}

fn matcher_rust(&self) -> anyhow::Result<PatternMatcher> {
    let mut builder = grep::regex::RegexMatcherBuilder::new();
    builder
        .multi_line(true)
        .unicode(!self.no_unicode)
        .octal(false)
        .fixed_strings(self.fixed_strings);

    // Case sensitivity
    match self.case {
        CaseMode::Sensitive => builder.case_insensitive(false),
        CaseMode::Insensitive => builder.case_insensitive(true),
        CaseMode::Smart => builder.case_smart(true),
    };

    // Word/line boundaries
    if let Some(ref boundary) = self.boundary {
        match *boundary {
            BoundaryMode::Line => builder.whole_line(true),
            BoundaryMode::Word => builder.word(true),
        };
    }

    // Line terminator handling
```

```
    if self.multiline {
builder.dot_matches_new_line(self.multiline_dotall);    } else {
builder.line_terminator(Some(b'\n'));        if self.null_data {
builder.line_terminator(Some(b'\x00'));        }    }    // Binary detection
interaction    if !self.binary.is_none() {
builder.ban_byte(Some(b'\x00'));    }    let m =
builder.build_many(&self.patterns.patterns)?;
Ok(PatternMatcher::RustRegex(m))}
```

**Error suggestion helpers:**

```
fn suggest_pcre2(msg: &str) -> Option<String> {
    if msg.contains("backreferences") || msg.contains("look-around") {
        Some(format!(
            "{msg}\n\nConsider enabling PCRE2 with the --pcre2 flag."
        ))
    } else {
        None
    }
}

fn suggest_multiline(msg: String) -> String {
    if msg.contains("the literal") && msg.contains("not allowed") {
        format!("{msg}\n\nConsider enabling multiline mode with -U.")
    } else {
        msg
    }
}
```

## Section 8: Building the Searcher

```
pub(crate) fn searcher(&self) -> anyhow::Result<grep::searcher::Searcher> {
    // Line terminator configuration
    let line_term = if self.crlf {
        grep::matcher::LineTerminator::crlf()
    } else if self.null_data {
        grep::matcher::LineTerminator::byte(b'\x00')
    } else {
        grep::matcher::LineTerminator::byte(b'\n')
    };

    let mut builder = grep::searcher::SearcherBuilder::new();
    builder
        .line_terminator(line_term)
        .line_number(self.line_number)
        .invert_match(self.invert_match)
        .multi_line(self.multiline)
        .memory_map(self.mmap_choice.clone())
        .max_matches(self.max_count)
        .stop_on_nonmatch(self.stop_on_nonmatch);

    // Context configuration
    match self.context {
        ContextMode::Passthru => builder.passthru(true),
        ContextMode::Limited(ref limited) => {
            let (before, after) = limited.get();
            builder.before_context(before);
            builder.after_context(after);
        }
    }

    // Encoding configuration
    match self.encoding {
        EncodingMode::Auto => {}  // Default BOM sniffing
        EncodingMode::Some(ref enc) => {
            builder.encoding(Some(enc.clone()));
        }
        EncodingMode::Disabled => {
            builder.bom_sniffing(false);
        }
    }

    Ok(builder.build())
}
```

**Memory map decision:**

```
let mmap_choice = {
    let maybe = unsafe { grep::searcher::MmapChoice::auto() };
    let never = grep::searcher::MmapChoice::never();

    match low.mmap {
        MmapMode::Auto => {
            // Heuristic: mmap for ≤10 regular files
            if paths.paths.len() <= 10
                && paths.paths.iter().all(|p| p.is_file())
            {
                maybe
            } else {
                never
            }
        }
        MmapMode::AlwaysTryMmap => maybe,
        MmapMode::Never => never,
    }
};
```

## Section 9: Building the Printer

```
pub(crate) fn printer<W: termcolor::WriteColor>(
    &self,
    search_mode: SearchMode,
    wtr: W,
) -> Printer<W> {
    // Determine summary kind for non-standard modes
    let summary_kind = if self.quiet {
        match search_mode {
            SearchMode::FilesWithMatches | SearchMode::Count
            | SearchMode::CountMatches | SearchMode::JSON
            | SearchMode::Standard => SummaryKind::QuietWithMatch,
            SearchMode::FilesWithoutMatch => SummaryKind::QuietWithoutMatch,
        }
    } else {
        match search_mode {
            SearchMode::FilesWithMatches => SummaryKind::PathWithMatch,
            SearchMode::FilesWithoutMatch => SummaryKind::PathWithoutMatch,
            SearchMode::Count => SummaryKind::Count,
            SearchMode::CountMatches => SummaryKind::CountMatches,
            SearchMode::JSON => return Printer::JSON(self.printer_json(wtr)),
            SearchMode::Standard => {
                return Printer::Standard(self.printer_standard(wtr))
            }
        }
    };
    Printer::Summary(self.printer_summary(wtr, summary_kind))
}

fn printer_standard<W: termcolor::WriteColor>(
    &self,
    wtr: W,
) -> grep::printer::Standard<W> {
    let mut builder = grep::printer::StandardBuilder::new();
    builder
        .byte_offset(self.byte_offset)
        .color_specs(self.colors.clone())
        .column(self.column)
        .heading(self.heading)
        .hyperlink(self.hyperlink_config.clone())
        .max_columns(self.max_columns)
        .max_columns_preview(self.max_columns_preview)
        .only_matching(self.only_matching)
        .path(self.with_filename)
        .replacement(self.replace.clone().map(|r| r.into()))
        .stats(self.stats.is_some())
        .trim_ascii(self.trim);

    // Single-threaded: printer owns file separator
    if self.threads == 1 {
        builder.separator_search(self.file_separator.clone());
    }

    builder.build(wtr)
}
```

**Printer types:**

| Mode | Printer | Output |
|---|---|---|
| Standard | `Standard<W>` | Matching lines with context |
| `-l` | `Summary<W>` | Filenames only |
| `-c` | `Summary<W>` | Counts per file |
| `--json` | `JSON<W>` | Structured JSON lines |

## Section 10: Building the Walker

```rust
pub(crate) fn walk_builder(&self) -> anyhow::Result<ignore::WalkBuilder> {
    let mut builder = ignore::WalkBuilder::new(&self.paths.paths[0]);

    // Add additional paths
    for path in self.paths.paths.iter().skip(1) {
        builder.add(path);
    }

    // Add custom ignore files
    if !self.no_ignore_files {
        for path in self.ignore_file.iter() {
            if let Some(err) = builder.add_ignore(path) {
                ignore_message!("{err}");
            }
        }
    }

    builder
        .max_depth(self.max_depth)
        .follow_links(self.follow)
        .max_filesize(self.max_filesize)
        .threads(self.threads)
        .same_file_system(self.one_file_system)
        .overrides(self.globs.clone())
        .types(self.types.clone())
        // Ignore rule configuration
        .hidden(!self.hidden)
        .parents(!self.no_ignore_parent)
        .ignore(!self.no_ignore_dot)
        .git_global(!self.no_ignore_vcs && !self.no_ignore_global)
        .git_ignore(!self.no_ignore_vcs)
        .git_exclude(!self.no_ignore_vcs && !self.no_ignore_exclude)
        .require_git(!self.no_require_git)
        .ignore_case_insensitive(self.ignore_file_case_insensitive);

    // Add .rgignore support
    if !self.no_ignore_dot {
        builder.add_custom_ignore_filename(".rgignore");
    }

    // Path sorting during traversal (ascending only)
    if let Some(ref sort) = self.sort {
        if !sort.reverse && matches!(sort.kind, SortModeKind::Path) {
            builder.sort_by_file_name(|a, b| a.cmp(b));
        }
    }

    Ok(builder)
}
```

**Ignore file precedence (highest to lowest):**

1. Command-line `--ignore-file`
2. `.rgignore`
3. `.ignore`
4. `.gitignore`
5. Global git ignores

---

## Section 11: Derived Configuration

```
// Thread count derivation
let threads = if low.sort.is_some() || paths.is_one_file {
    1  // Sorting or single file → no parallelism benefit
} else if let Some(threads) = low.threads {
    threads  // User specified
} else {
    std::thread::available_parallelism()
        .map_or(1, |n| n.get())
        .min(12)  // Cap at 12
};

// Line number derivation
let line_number = low.line_number.unwrap_or_else(|| {
    if low.quiet { return false; }
    let Mode::Search(ref mode) = low.mode else { return false };
    match *mode {
        SearchMode::FilesWithMatches
        | SearchMode::FilesWithoutMatch
        | SearchMode::Count
        | SearchMode::CountMatches => false,
        SearchMode::JSON => true,
        SearchMode::Standard => {
            (state.is_terminal_stdout && !paths.is_only_stdin())
                || column
                || low.vimgrep
        }
    }
});

// Heading derivation
let heading = match low.heading {
    None => !low.vimgrep && state.is_terminal_stdout,
    Some(false) => false,
    Some(true) => !low.vimgrep,  // vimgrep overrides
};

// Filename display derivation
let with_filename = low.with_filename
    .unwrap_or_else(|| low.vimgrep || !paths.is_one_file);
```

## Section 12: Helper Functions

**Type definitions:**

```
fn types(low: &LowArgs) -> anyhow::Result<ignore::types::Types> {
    let mut builder = ignore::types::TypesBuilder::new();
    builder.add_defaults();  // rust, python, c, etc.

    for tychange in low.type_changes.iter() {
        match *tychange {
            TypeChange::Clear { ref name } => builder.clear(name),
            TypeChange::Add { ref def } => builder.add_def(def)?,
            TypeChange::Select { ref name } => builder.select(name),
            TypeChange::Negate { ref name } => builder.negate(name),
        }
    }
    Ok(builder.build()?)
}
```

**Glob overrides:**

```
fn globs(
    state: &State,
    low: &LowArgs,
) -> anyhow::Result<ignore::overrides::Override> {
    let mut builder = ignore::overrides::OverrideBuilder::new(&state.cwd);

    // Global case insensitivity
    if low.glob_case_insensitive {
        builder.case_insensitive(true).unwrap();
    }

    // Regular globs
    for glob in low.globs.iter() {
        builder.add(glob)?;
    }

    // Case-insensitive globs (--iglob)
    builder.case_insensitive(true).unwrap();
    for glob in low.iglobs.iter() {
        builder.add(&glob)?;
    }

    Ok(builder.build()?)
}
```

**Hostname resolution:**

```rust
fn hostname(bin: Option<&Path>) -> Option<String> {
    // Try user-specified binary first
    let Some(bin) = bin else { return platform_hostname() };

    let bin = match grep::cli::resolve_binary(bin) {
        Ok(bin) => bin,
        Err(err) => {
            log::debug!("hostname binary failed: {err}");
            return platform_hostname();
        }
    };

    let mut cmd = std::process::Command::new(&bin);
    cmd.stdin(std::process::Stdio::null());

    let rdr = match grep::cli::CommandReader::new(&mut cmd) {
        Ok(rdr) => rdr,
        Err(_) => return platform_hostname(),
    };

    let out = std::io::read_to_string(rdr).ok()?;
    let hostname = out.trim();

    if hostname.is_empty() {
        platform_hostname()
    } else {
        Some(hostname.to_string())
    }
}
```

## Quick Reference: HiArgs Methods

| Method | Returns | Purpose |
|---|---|---|
| `matcher()` | `PatternMatcher` | Compiled regex engine |
| `searcher()` | `Searcher` | File reading configuration |
| `printer()` | `Printer<W>` | Output formatter |
| `search_worker()` | `SearchWorker<W>` | Coordinates all three |
| `walk_builder()` | `WalkBuilder` | Directory traversal |

| | | |
|---|---|---|
| `haystack_builder()` | `HaystackBuilder` | Path → Haystack converter |
| `buffer_writer()` | `BufferWriter` | Thread-safe output |
| `stdout()` | `StandardStream` | Colored stdout writer |
| `sort()` | `Iterator` | Optional result sorting |
| `threads()` | `usize` | Parallelism level |
| `mode()` | `Mode` | What command to run |
| `stats()` | `Option<Stats>` | Statistics tracking |
| `types()` | `&Types` | File type matcher |

# Ripgrep hiargs.rs: The Configuration Oracle

## What This File Does

The hiargs.rs file transforms raw command-line flags into a validated, computed configuration. It's the bridge between "what the user typed" and "what ripgrep should do." Every call to `args.something()` in main.rs ultimately flows through this struct.

The name "HiArgs" stands for "high-level arguments" — in contrast to "LowArgs" which directly mirrors CLI flags. HiArgs resolves conflicts, applies defaults, computes derived values, and builds the complex objects that ripgrep needs.

---

## Section 1: The HiArgs Struct

The struct contains roughly 70 fields. This might seem excessive, but each field represents a distinct configuration decision. The struct is the single source of truth for the entire search operation.

Fields fall into several categories. Display options control how output looks: colors, headings, line numbers, column numbers. Search options control matching behavior: case sensitivity, word boundaries, multiline mode. Filter options control what gets searched: file types, globs, ignore rules. Performance options control resource usage: thread count, memory maps, size limits.

The struct is marked Debug but not Clone. This is intentional — HiArgs is created once and borrowed throughout the search. Cloning would be expensive and unnecessary.

See: Companion Code Section 1

---

## Section 2: The Transformation — from_low_args

The from_low_args function is where the magic happens. It takes a LowArgs (raw flags) and produces a HiArgs (computed configuration). This transformation can fail — invalid globs, inaccessible directories, conflicting options.

The function begins with an assertion that no "special mode" is present. Special modes like help and version short-circuit before reaching this point. If we're here, we're doing real work.

Early in the function, mode adjustments happen. Two cases demonstrate flag interaction: when invert_match is set with count-matches mode, it becomes regular count mode. When only_matching is set with count mode, it becomes count-matches mode. These aren't arbitrary — they reflect what makes semantic sense.

The function creates a State struct to track information needed across multiple conversions. Then it processes patterns, paths, binary detection, colors, hyperlinks, stats, types, and globs in sequence. Order matters here — later conversions may depend on earlier ones.

See: Companion Code Section 2

---

## Section 3: State Management

The State struct tracks three things: whether stdout is a terminal, whether stdin has been consumed, and the current working directory.

Terminal detection affects default behavior. When output goes to a terminal, ripgrep enables colors and line numbers by default. When piped to another program, it disables them. This matches user expectations without requiring explicit flags.

Stdin consumption tracking prevents a subtle bug. If someone runs "rg -f - -", they're trying to read patterns from stdin AND search stdin. That's impossible — stdin can only be read once. The state tracks whether patterns already consumed stdin so path processing can error appropriately.

The current working directory is captured once at startup. This matters for glob resolution and path normalization. If ripgrep is run from a directory that gets deleted during execution, having the original CWD prevents confusing errors.

See: Companion Code Section 3

---

## Section 4: Pattern Handling

The Patterns struct extracts search patterns from multiple sources. Patterns can come from positional arguments, the -e flag (multiple times), or files via -f. The common case is a single positional argument: "rg foo" means search for "foo."

When -e or -f is used, the first positional is no longer a pattern — it becomes a path. This matches grep convention but requires careful logic to handle correctly.

Pattern deduplication happens during collection. This might seem unnecessary, but duplicate patterns can cause significant slowdown. The regex engine will eventually deduplicate internally, but not until after parsing and compilation. With many duplicates, that overhead becomes noticeable. A HashSet catches duplicates early.

Reading patterns from stdin via "-f -" interacts with the state tracking. Once stdin is consumed for patterns, it can't also be searched. The state records this so later path processing knows stdin is unavailable.

See: Companion Code Section 4

---

## Section 5: Path Handling

The Paths struct determines what to search. This involves subtle heuristics when no paths are given explicitly.

When paths are provided, they're used directly. The code tracks whether exactly one file (not directory) is being searched — this affects whether filenames appear in output.

When no paths are given, heuristics kick in. If stdin appears readable AND hasn't been consumed by pattern reading AND we're in search mode, ripgrep searches stdin. Otherwise, it searches the current directory. This heuristic exists because many programs open stdin pipes even when not providing input, causing ripgrep to appear to hang waiting for input that will never come.

The has_implicit_path flag records whether ripgrep guessed the path. This affects output formatting — explicit paths get "./" prefixes, implicit ones don't. It also affects error messages — ripgrep only warns about "nothing searched" when the path was implicit.

See: Companion Code Section 5

---

## Section 6: Binary Detection

Ripgrep uses two different binary detection strategies depending on whether a file was explicitly requested or discovered through directory traversal.

For explicitly requested files, ripgrep never quits early. If you say "rg pattern binary.exe", you want that file searched even if it's binary. The detection mode converts NUL bytes to something displayable or disables detection entirely.

For implicitly discovered files during directory traversal, ripgrep can quit early when it detects binary content. This is the default behavior that makes ripgrep fast — it skips binary files automatically without the user having to think about it.

The distinction matters for correctness. An explicit file path is a command: "search this." Quitting early would be wrong. An implicit file is a suggestion: "this might be relevant." Quitting early is helpful filtering.

See: Companion Code Section 6

---

## Section 7: Building the Matcher

The matcher method constructs the regex engine based on user choice. Three engines are available: default (Rust regex), PCRE2 (optional feature), and auto (try Rust, fall back to PCRE2).

The default engine uses Rust's regex crate. It's fast and has good error messages but doesn't support lookaround or backreferences. The builder configures case sensitivity, word boundaries, line terminators, multiline mode, Unicode handling, and size limits.

PCRE2 provides advanced regex features at the cost of being an external C dependency. It supports lookaround and backreferences, which some patterns require. The JIT compiler is enabled on 64-bit systems for better performance.

Auto mode is clever. It first tries the Rust engine. If that fails (perhaps due to unsupported syntax), it tries PCRE2. If both fail, the error message shows both failures so the user understands what went wrong.

Error messages include suggestions. If the Rust engine fails because of backreferences, the error suggests trying PCRE2. This is user-friendly — instead of just failing, ripgrep guides toward a solution.

See: Companion Code Section 7

## Section 8: Building the Searcher

The searcher method configures how files are read and searched. This includes line terminator handling, context lines, memory mapping strategy, and encoding detection.

Line terminators aren't always newlines. The null_data flag switches to NUL byte terminators for searching NUL-delimited data. CRLF mode handles Windows line endings. Getting this right is essential for correct line counting and context display.

Memory mapping is a performance optimization with tradeoffs. It's faster for searching a few large files but slower for many small files due to setup overhead. The auto mode uses heuristics: if searching ten or fewer files that are all regular files, enable memory maps. Otherwise, use buffered I/O.

The comment about memory map safety is worth reading. Memory maps are inherently unsafe in Rust's model — if the underlying file is modified or truncated during reading, you get undefined behavior (likely SIGBUS). The code explicitly acknowledges this tradeoff.

See: Companion Code Section 8

---

## Section 9: Building the Printer

The printer method constructs output formatters. Three printer types exist: standard (grep-like lines), summary (counts and filenames), and JSON (structured data).

The search mode determines which printer to use. Standard mode uses the standard printer. Count modes use the summary printer. JSON mode uses the JSON printer. The quiet flag further modifies behavior — quiet summary printers still track matches but suppress output.

The standard printer has extensive configuration. Byte offsets, column numbers, headings, hyperlinks, color specs, context separators, path formatting — everything that affects how a match line looks. Single-threaded versus multi-threaded mode affects whether the printer handles file separators itself or defers to the buffer writer.

See: Companion Code Section 9

---

## Section 10: Building the Walker

The walk_builder method configures directory traversal. This is where ignore rules, file type filters, depth limits, and symlink following get set up.

The ignore crate handles the complexity of gitignore-style rules. Multiple ignore files stack: global git ignores, repository ignores, directory-local ignores, rgignore files. Each can be individually enabled or disabled via flags.

File type filtering uses pre-built definitions. "Rust" means .rs files. "Python" means .py files. Users can define custom types or modify existing ones. The types are applied during traversal, not after — so ignored files never even get enumerated.

Sorting interacts with parallelism. When sorting is requested, parallelism is disabled because you can't sort results that arrive out of order. Path sorting in ascending order gets special treatment — the walker can sort during traversal, avoiding a collect-and-sort step.

See: Companion Code Section 10

---

## Section 11: Derived Configuration

Many HiArgs fields are derived from combinations of LowArgs fields. These derivations encode ripgrep's opinion about sensible defaults.

Thread count derivation considers multiple factors. Sorting forces single-threaded mode. Searching a single file forces single-threaded mode (parallelism overhead exceeds benefit). Otherwise, use the user's requested count, or auto-detect from available parallelism, capped at twelve.

Line number display depends on context. Terminal output gets line numbers by default. Piped output doesn't. Certain modes (counts, file lists) never show line numbers. Requesting columns or vimgrep mode implies line numbers.

Color choice respects both user preference and environment. Auto mode enables colors for terminals, disables for pipes. The NO_COLOR environment variable can override. The TERM variable affects capability detection.

See: Companion Code Section 11

## Section 12: Helper Functions

The file ends with standalone helper functions for building complex objects.

The types function builds the file type matcher from user-specified changes. Users can clear existing types, add new definitions, select types to include, or negate types to exclude.

The globs function builds override matchers from -g and --iglob flags. Globs can include or exclude files. Case sensitivity can be toggled globally or per-glob.

The hostname function retrieves the system hostname for hyperlink formatting. It first tries a user-specified binary, then falls back to platform-specific methods. This flexibility lets users override hostname detection in unusual environments.

Error message enhancement functions inspect regex compilation failures and suggest solutions. If the error mentions backreferences, suggest PCRE2. If it mentions literal newlines, suggest multiline mode. If it mentions NUL bytes, suggest text mode.

See: Companion Code Section 12

---

## Key Takeaways

First, configuration is computation. HiArgs doesn't just store flags — it resolves interactions, applies heuristics, and pre-builds complex objects.

Second, the transformation is ordered. Patterns before paths. State tracked throughout. Each conversion can influence later ones.

Third, user experience drives design. Error messages include suggestions. Defaults adapt to context. Heuristics handle common cases.

Fourth, builder patterns dominate. Matcher, searcher, printer, walker — each is constructed through a builder that HiArgs configures.

Fifth, explicit versus implicit matters. Files requested by name get different treatment than files discovered by traversal.

---

## What to Read Next

Understanding HiArgs raises questions about the objects it creates:

How does SearchWorker use the matcher, searcher, and printer together? Read search.rs.

How does the Haystack abstraction work? Read haystack.rs.

What does LowArgs look like before transformation? Read flags/lowargs.rs.

How does the walker actually traverse directories? Read the ignore crate.

# Ripgrep logger.rs: Code Companion

Reference code for the logger.rs lecture. Sections correspond to the lecture document.

---

## Section 1: The Log Crate Facade

```
/*!
Defines a super simple logger that works with the `log` crate.

We don't do anything fancy. We just need basic log levels and the ability to
print to stderr. We therefore avoid bringing in extra dependencies just for
this functionality.
*/

use log::Log;
```

**The facade pattern:**

```
// Library code (doesn't know about Logger):
log::debug!("Processing file: {}", path.display());
log::trace!("Binary detection: {:?}", mode);

// Application code (provides the implementation):
Logger::init()?;  // Now debug!/trace! flow to our Logger
```

**Log levels (from lowest to highest priority):**

```
log::trace!("...");  // Most verbose, rarely needed
log::debug!("...");  // Debugging information
log::info!("...");   // General information
log::warn!("...");   // Warnings
log::error!("...");  // Errors
```

## Section 2: The Logger Struct

```rust
/// The simplest possible logger that logs to stderr.
///
/// This logger does no filtering. Instead, it relies on the `log` crates
/// filtering via its global max_level setting.
#[derive(Debug)]
pub(crate) struct Logger(());

/// A singleton used as the target for an implementation of the `Log` trait.
const LOGGER: &'static Logger = &Logger(());
```

**Zero-sized type (ZST):**

```rust
// Logger contains () which has size 0
std::mem::size_of::<Logger>()  // Returns 0

// No runtime cost — exists only for type system
// The reference &Logger also has a fixed size (pointer)
// but points to "nothing" in a sense
```

**Why the newtype pattern:**

```rust
// Can't do this:
impl Log for () { ... }  // Error: can't impl foreign trait on foreign type

// Can do this:
struct Logger(());
impl Log for Logger { ... }  // OK: Logger is our type
```

---

## Section 3: Initialization

```rust
impl Logger {
    /// Create a new logger that logs to stderr and initialize it as the
    /// global logger. If there was a problem setting the logger, then an
    /// error is returned.
    pub(crate) fn init() -> Result<(), log::SetLoggerError> {
        log::set_logger(LOGGER)
    }
}
```

**Usage in ripgrep startup:**

```
// In argument processing (simplified):
if args.debug {
    log::set_max_level(log::LevelFilter::Debug);
    Logger::init()?;
} else if args.trace {
    log::set_max_level(log::LevelFilter::Trace);
    Logger::init()?;
} else {
    log::set_max_level(log::LevelFilter::Off);
    // Logger::init() not needed - no logs will be emitted
}
```

**Level filtering:**

```
log::set_max_level(log::LevelFilter::Debug);
// Now: trace! → skipped, debug!/info!/warn!/error! → logged

log::set_max_level(log::LevelFilter::Off);
// Now: all log macros → skipped (no overhead)
```

---

## Section 4: The Log Trait — enabled

```
impl Log for Logger {
    fn enabled(&self, _: &log::Metadata<'_>) -> bool {
        // We set the log level via log::set_max_level, so we don't need to
        // implement filtering here.
        true
    }
    // ...
}
```

**Why always return true:**

```
Log macro expansion (simplified):

log::debug!("message")
    ↓
if log::max_level() >= Debug {        // First filter (fast)
    if logger.enabled(metadata) {      // Second filter (we return true)
        logger.log(record);
    }
}

Since max_level() handles filtering, enabled() just says "yes, proceed"
```

## Section 5: The Log Trait — log

```
fn log(&self, record: &log::Record<'_>) {
    match (record.file(), record.line()) {
        (Some(file), Some(line)) => {
            eprintln_locked!(
                "{}|{}|{}:{}: {}",
                record.level(),
                record.target(),
                file,
                line,
                record.args()
            );
        }
        (Some(file), None) => {
            eprintln_locked!(
                "{}|{}|{}: {}",
                record.level(),
                record.target(),
                file,
                record.args()
            );
        }
        _ => {
            eprintln_locked!(
                "{}|{}: {}",
                record.level(),
                record.target(),
                record.args()
            );
        }
    }
}
```

**Output format examples:**

```
DEBUG|ripgrep::search|search.rs:42: searching file.txt
TRACE|ignore::walk|walk.rs:128: skipping hidden directory
DEBUG|grep_searcher: mmap enabled for large file
```

**Record fields:**

```
record.level()   // DEBUG, TRACE, etc.
record.target()  // Usually module path: "ripgrep::search"
record.file()    // Source file: Some("search.rs") or None
record.line()    // Line number: Some(42) or None
record.args()    // The formatted message
```

**Thread-safety via eprintln_locked:**

```
// From messages.rs — ensures debug output doesn't interleave
eprintln_locked!("{}|{}: {}", level, target, message);
// Locks stdout (to coordinate with search output)
// Then locks stderr
// Writes complete message atomically
```

---

## Section 6: The Log Trait — flush

```
fn flush(&self) {
    // We use eprintln_locked! which is flushed on every call.
}
```

**Why empty:**

```
// eprintln_locked! does:
writeln!(stderr, ...)?;  // writeln flushes line-buffered streams
drop(stdout);            // Releases lock, implicit flush

// No buffering in our logger → nothing to flush
```

**Contrast with buffered logger:**

```
// A buffered logger might do:
fn log(&self, record: &Record) {
    self.buffer.push(format!("{}", record.args()));
    if self.buffer.len() > 100 {
        self.flush();
    }
}

fn flush(&self) {
    for msg in self.buffer.drain(..) {
        eprintln!("{}", msg);
    }
}
```

## Section 7: Integration with ripgrep

**Activation via flags:**

```
rg --debug pattern file.txt
# Output includes: DEBUG|...|search.rs:N: ...

rg --trace pattern file.txt
# Output includes: TRACE|...|... (even more verbose)

RIPGREP_LOG=debug rg pattern file.txt
# Same as --debug, via environment variable
```

**Example log output during search:**

```
DEBUG|ripgrep::flags::hiargs: using 8 threads
DEBUG|ripgrep::search|search.rs:255: file.txt: binary detection: Quit
TRACE|ignore::walk|walk.rs:892: ignoring .git (hidden directory)
TRACE|grep_searcher::searcher: searching with memory map
DEBUG|ripgrep::search|search.rs:347: file.txt: found 3 matches
```

**Log statements in search.rs:**

```
// From the actual search.rs:
log::trace!("{}: binary detection: {:?}", path.display(), bin);
```

**No overhead when disabled:**

```
// When max_level is Off:
log::debug!("expensive: {}", compute_something());
// Expands to roughly:
if log::max_level() >= Debug {  // false, so...
    // This entire block is skipped
    // compute_something() is never called
}
```

## Quick Reference: Log Crate API

```
// Setting up
log::set_max_level(LevelFilter::Debug);  // Enable debug and above
log::set_logger(&LOGGER)?;               // Register our implementation

// Emitting logs (from anywhere in the codebase)
log::trace!("very verbose: {}", detail);
log::debug!("debugging: {}", info);
log::info!("general info: {}", msg);
log::warn!("warning: {}", issue);
log::error!("error: {}", err);

// Level filters
log::LevelFilter::Off    // Nothing
log::LevelFilter::Error  // Only errors
log::LevelFilter::Warn   // Warn and above
log::LevelFilter::Info   // Info and above
log::LevelFilter::Debug  // Debug and above
log::LevelFilter::Trace  // Everything
```

---

## Complete Module (72 lines)

```
use log::Log;

#[derive(Debug)]
pub(crate) struct Logger(());

const LOGGER: &'static Logger = &Logger(());

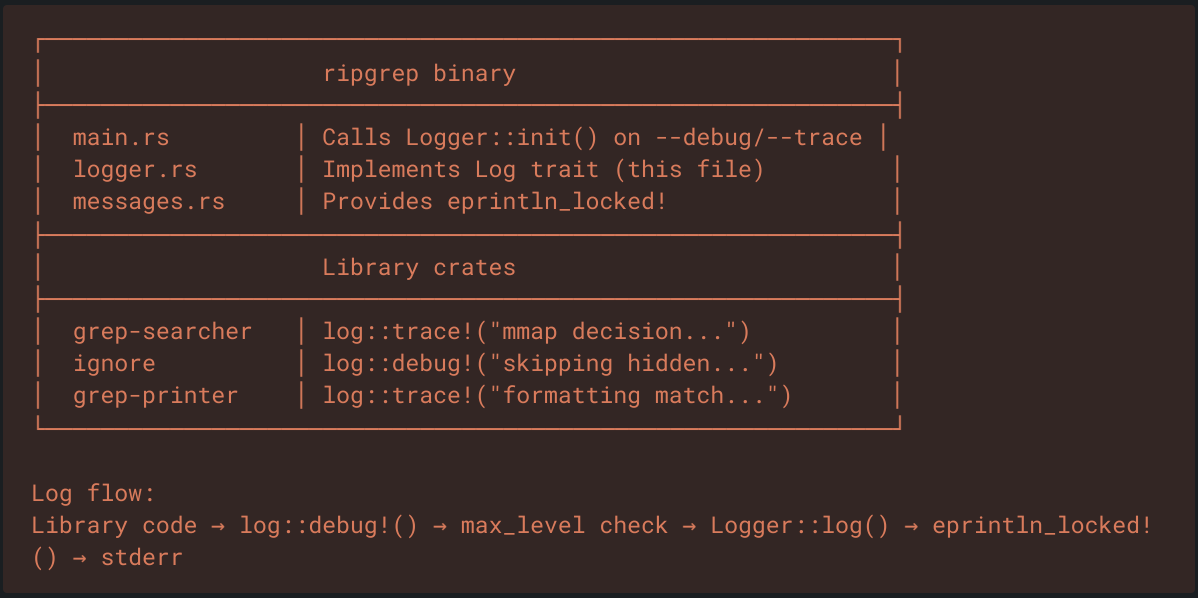
impl Logger {
    pub(crate) fn init() -> Result<(), log::SetLoggerError> {
        log::set_logger(LOGGER)
    }
}

impl Log for Logger {
    fn enabled(&self, _: &log::Metadata<'_>) -> bool {
        true
    }

    fn log(&self, record: &log::Record<'_>) {
        match (record.file(), record.line()) {
            (Some(file), Some(line)) => {
                eprintln_locked!(
                    "{}|{}|{}:{}: {}",
                    record.level(), record.target(), file, line, record.args()
                );
            }
            (Some(file), None) => {
                eprintln_locked!(
                    "{}|{}|{}: {}",
                    record.level(), record.target(), file, record.args()
                );
            }
            _ => {
                eprintln_locked!(
                    "{}|{}: {}",
                    record.level(), record.target(), record.args()
                );
            }
        }
    }

    fn flush(&self) {}
}
```

## Relationship to Other Modules




Log flow:
Library code → log::debug!() → max_level check → Logger::log() → eprintln_locked!() → stderr

# Ripgrep logger.rs: Minimal Debug Logging

## What This File Does

The logger.rs module implements a minimal logger for the standard `log` crate facade. At 72 lines, it's the smallest core module. Its job is simple: when debug or trace logging is enabled, format log messages and print them to stderr using the thread-safe infrastructure from messages.rs.

The module comment captures the philosophy: "We don't do anything fancy. We just need basic log levels and the ability to print to stderr. We therefore avoid bringing in extra dependencies just for this functionality."

---

## Section 1: The Log Crate Facade

Rust's `log` crate provides a logging facade — a set of macros (log!, debug!, trace!, etc.) that emit log records without knowing where those records go. The actual destination is determined by a logger implementation registered at runtime.

This separation lets library code emit logs without depending on a specific logging framework. Ripgrep's library crates use log macros throughout. The binary crate provides the logger implementation that decides what happens to those messages.

Ripgrep could use a full-featured logging crate like `env_logger` or `tracing`, but that would add dependencies. For ripgrep's needs — occasional debug output to stderr — a custom 30-line implementation suffices.

See: Companion Code Section 1

---

## Section 2: The Logger Struct

The Logger struct is a zero-sized type — it contains no data. The unit tuple `()` is just a placeholder. Zero-sized types have no runtime cost; they exist only for the type system.

Why wrap a unit tuple in a struct? The `Log` trait requires a concrete type to implement. You can't implement traits on primitives like `()`. The newtype pattern gives us a type we control.

The LOGGER constant is a static reference to a Logger. Since Logger is zero-sized, this doesn't allocate anything. The `&'static` lifetime means the reference is valid for the entire program, which the log crate requires for its global logger.

See: Companion Code Section 2

---

## Section 3: Initialization

The init function registers the logger with the log crate. After this call, any log macro invocation anywhere in the program flows through this logger.

The function returns a Result because set_logger can fail if a logger was already registered. In ripgrep, init is called once during startup; failure means something is badly misconfigured.

Filtering happens elsewhere. The log crate has a global maximum level setting. Messages above that level never reach the logger. Ripgrep sets this based on the --debug or --trace flags.

See: Companion Code Section 3

---

## Section 4: The Log Trait — enabled

The Log trait has three methods. The enabled method determines whether a given log level should be recorded. Returning false causes the log crate to skip the message entirely.

Ripgrep's implementation always returns true. Filtering happens via log::set_max_level instead. This is a valid approach — the max_level filter runs before enabled is even called.

Why not filter in enabled? It would work, but checking max_level is faster because it happens in the macro expansion before constructing the log record. The enabled method exists for more complex filtering scenarios that ripgrep doesn't need.

See: Companion Code Section 4

## Section 5: The Log Trait — log

The log method does the actual work. It receives a Record containing the log level, target module, optional file/line information, and the formatted message.

The implementation formats output differently based on available metadata. With file and line: "DEBUG|ripgrep::search|search.rs:42: message". With just file: "DEBUG|ripgrep::search|search.rs: message". With neither: "DEBUG|ripgrep::search: message".

Output uses eprintln_locked from messages.rs. This ensures debug messages don't interleave with each other or with error messages during parallel search. The same thread-safety mechanism protects all stderr output.

See: Companion Code Section 5

---

## Section 6: The Log Trait — flush

The flush method ensures buffered output is written. Ripgrep's implementation is empty because eprintln_locked flushes after every message.

Buffered loggers might accumulate messages for performance, flushing periodically or on demand. Ripgrep's use case — infrequent debug messages — doesn't benefit from buffering. Immediate output is more useful for debugging.

See: Companion Code Section 6

---

## Section 7: Integration with ripgrep

Debug logging activates via --debug or RIPGREP_LOG=debug. Trace logging provides even more detail via --trace. These flags set the max_level filter and call Logger::init.

The search.rs module uses log::trace! to report binary detection decisions. The ignore crate uses debug logging to explain why files are skipped. Various modules log timing and configuration information.

Without explicit activation, the max_level defaults to Off, meaning no log records are created. There's no overhead from unused logging — the macros short-circuit before allocating anything.

See: Companion Code Section 7

---

## Key Takeaways

First, the log crate's facade pattern separates log emission from log handling. Libraries emit; applications decide where logs go.

Second, zero-sized types enable trait implementations without data. The Logger struct exists only to satisfy the type system.

Third, eprintln_locked provides thread-safety for debug output, reusing the same infrastructure as error messages.

Fourth, minimal implementations suffice for simple needs. Ripgrep avoids dependency bloat by implementing Log in 30 lines.

Fifth, filtering at max_level is more efficient than filtering in enabled. The check happens before record construction.

---

## What to Read Next

This completes the core internal modules. The remaining "roots" are external crates:

How does the ignore crate traverse directories? Read ignore's walk module.

How does grep-searcher read and match files? Read the searcher implementation.

How does grep-printer format output? Read the printer implementations.

# Ripgrep lowargs.rs: Code Companion

Reference code for the lowargs.rs lecture. Sections correspond to the lecture document.

---

## Section 1: The Design Philosophy

```
/*!
Provides the definition of low level arguments from CLI flags.
*/

/// A collection of "low level" arguments.
///
/// The "low level" here is meant to constrain this type to be as close to the
/// actual CLI flags and arguments as possible. Namely, other than some
/// convenience types to help validate flag values and deal with overrides
/// between flags, these low level arguments do not contain any higher level
/// abstractions.
///
/// Another self-imposed constraint is that populating low level arguments
/// should not require anything other than validating what the user has
/// provided. For example, low level arguments should not contain a
/// `HyperlinkConfig`, since in order to get a full configuration, one needs to
/// discover the hostname of the current system (which might require running a
/// binary or a syscall).
```

**LowArgs vs HiArgs:**

| Aspect | LowArgs | HiArgs |
|---|---|---|
| Contains | Raw flag values | Computed configuration |
| Validation | Syntax only | Semantic + environmental |
| Can fail due to | Invalid flag values | Missing files, platform issues |
| --help works when | Always | Always (uses LowArgs only) |

## Section 2: The LowArgs Struct

```rust
#[derive(Debug, Default)]
pub(crate) struct LowArgs {
    // Essential arguments.
    pub(crate) special: Option<SpecialMode>,
    pub(crate) mode: Mode,
    pub(crate) positional: Vec<OsString>,
    pub(crate) patterns: Vec<PatternSource>,

    // Everything else, sorted lexicographically.
    pub(crate) binary: BinaryMode,
    pub(crate) boundary: Option<BoundaryMode>,
    pub(crate) buffer: BufferMode,
    pub(crate) byte_offset: bool,
    pub(crate) case: CaseMode,
    pub(crate) color: ColorChoice,
    pub(crate) colors: Vec<UserColorSpec>,
    pub(crate) column: Option<bool>,
    pub(crate) context: ContextMode,
    // ... ~60 more fields
    pub(crate) threads: Option<usize>,
    pub(crate) trim: bool,
    pub(crate) type_changes: Vec<TypeChange>,
    pub(crate) unrestricted: usize,
    pub(crate) vimgrep: bool,
    pub(crate) with_filename: Option<bool>,
}
```

**Field categories:**

| Category | Examples | Count |
|---|---|---|
| Essential | mode, patterns, positional | 4 |
| Display | color, heading, line_number | ~15 |
| Search | case, boundary, multiline | ~10 |
| Filter | globs, type_changes, hidden | ~15 |
| Performance | threads, mmap, max_filesize | ~10 |

| | | |
|---|---|---|
| Ignore rules | no_ignore_*, follow | ~10 |

## Section 3: SpecialMode — The Short-Circuit Cases

```rust
/// A "special" mode that supercedes everything else.
///
/// When one of these modes is present, it overrides everything else and causes
/// ripgrep to short-circuit.
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
pub(crate) enum SpecialMode {
    /// Show a condensed version of "help" output.
    /// Corresponds to the `-h` flag.
    HelpShort,
    /// Shows a very verbose version of the "help" output.
    /// Corresponds to the `--help` flag.
    HelpLong,
    /// Show condensed version information. e.g., `ripgrep x.y.z`.
    VersionShort,
    /// Show verbose version information including build features.
    VersionLong,
    /// Show PCRE2's version information.
    VersionPCRE2,
}
```

**Usage in main.rs:**

```rust
fn run(result: ParseResult<HiArgs>) -> anyhow::Result<ExitCode> {
    let args = match result {
        ParseResult::Err(err) => return Err(err),
        ParseResult::Special(mode) => return special(mode),  // Short-circuit!
        ParseResult::Ok(args) => args,
    };
    // ... normal processing
}
```

## Section 4: Mode — What Ripgrep Should Do

```rust
/// The overall mode that ripgrep should operate in.
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
pub(crate) enum Mode {
    /// ripgrep will execute a search of some kind.
    Search(SearchMode),
    /// Show the files that *would* be searched, but don't search them.
    Files,
    /// List all file type definitions configured.
    Types,
    /// Generate various things like the man page and completion files.
    Generate(GenerateMode),
}

impl Default for Mode {
    fn default() -> Mode {
        Mode::Search(SearchMode::Standard)
    }
}

impl Mode {
    /// Update this mode to the new mode while implementing override semantics.
    pub(crate) fn update(&mut self, new: Mode) {
        match *self {
            // If we're in a search mode, then anything can override it.
            Mode::Search(_) => *self = new,
            _ => {
                // Once in a non-search mode, only other non-search modes
                // can override. So `--files -l` stays Mode::Files.
                if !matches!(new, Mode::Search(_)) {
                    *self = new;
                }
            }
        }
    }
}
```

**Override examples:**

```
rg -l pattern          # Mode::Search(FilesWithMatches)
rg --files -l pattern  # Mode::Files (search mode can't override)
rg --files --types     # Mode::Types (non-search can override)
```

---

## Section 5: SearchMode — Search Output Variations

```rust
/// The kind of search that ripgrep is going to perform.
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
pub(crate) enum SearchMode {
    /// The default standard mode. Print matches when found.
    Standard,
    /// Show files containing at least one match. (-l)
    FilesWithMatches,
    /// Show files that don't contain any matches. (--files-without-match)
    FilesWithoutMatch,
    /// Show match count per file. (-c)
    Count,
    /// Show total match count per file. (--count-matches)
    CountMatches,
    /// Print matches in JSON lines format. (--json)
    JSON,
}
```

**Count vs CountMatches:**

```
File content: "foo foo bar foo"
Pattern: "foo"

-c (Count):        1  (one matching line)
--count-matches:   3  (three matches)
```

## Section 6: BinaryMode — Handling Non-Text Files

```rust
/// Indicates how ripgrep should treat binary data.
#[derive(Debug, Default, Eq, PartialEq)]
pub(crate) enum BinaryMode {
    /// Automatically determine based on how file was specified.
    /// Explicit files: SearchAndSuppress
    /// Implicit files: skip entirely
    #[default]
    Auto,
    /// Search but suppress matches, showing only a warning.
    /// NUL bytes replaced with line terminators.
    SearchAndSuppress,
    /// Treat all files as plain text. No skipping, no NUL replacement.
    AsText,
}
```

**Detection flow:**

```
File specified explicitly (rg pattern file.bin):
  → SearchAndSuppress: search, but warn about binary

File discovered during traversal:
  → Quit on first NUL byte: skip silently

-a/--text flag:
  → AsText: search everything as-is
```

---

## Section 7: CaseMode — Pattern Matching Sensitivity

```rust
/// Indicates the case mode for pattern interpretation.
#[derive(Debug, Default, Eq, PartialEq)]
pub(crate) enum CaseMode {
    /// 'a' matches only 'a'.
    #[default]
    Sensitive,
    /// 'a' matches both 'a' and 'A'. (-i)
    Insensitive,
    /// Case-insensitive only when pattern is all lowercase. (-S)
    Smart,
}
```

**Smart case examples:**

```
rg foo      # Matches: foo, Foo, FOO (pattern is lowercase)
rg Foo      # Matches: Foo only (pattern has uppercase)
rg FOO      # Matches: FOO only
```

---

## Section 8: ColorChoice — Output Coloring

```
/// Indicates whether ripgrep should include color in output.
#[derive(Debug, Default, Eq, PartialEq)]
pub(crate) enum ColorChoice {
    /// Color will never be used.
    Never,
    /// Color only when stdout is a tty.
    #[default]
    Auto,
    /// Color will always be used.
    Always,
    /// Always use ANSI escapes (Windows legacy console workaround).
    Ansi,
}

impl ColorChoice {
    /// Convert to the termcolor crate's equivalent type.
    pub(crate) fn to_termcolor(&self) -> termcolor::ColorChoice {
        match *self {
            ColorChoice::Never => termcolor::ColorChoice::Never,
            ColorChoice::Auto => termcolor::ColorChoice::Auto,
            ColorChoice::Always => termcolor::ColorChoice::Always,
            ColorChoice::Ansi => termcolor::ColorChoice::AlwaysAnsi,
        }
    }
}
```

## Section 9: ContextMode — Lines Around Matches

```rust
/// Indicates the line context options ripgrep should use.
#[derive(Debug, Eq, PartialEq)]
pub(crate) enum ContextMode {
    /// All lines will be printed (--passthru).
    Passthru,
    /// Show specific number of lines before/after matches.
    Limited(ContextModeLimited),
}

/// Tracks before/after/both context separately for precedence.
#[derive(Debug, Default, Eq, PartialEq)]
pub(crate) struct ContextModeLimited {
    before: Option<usize>,  // -B
    after: Option<usize>,   // -A
    both: Option<usize>,    // -C
}

impl ContextModeLimited {
    /// Returns (before, after) with proper precedence.
    /// -B and -A always override -C regardless of order.
    pub(crate) fn get(&self) -> (usize, usize) {
        let (mut before, mut after) =
            self.both.map(|lines| (lines, lines)).unwrap_or((0, 0));
        if let Some(lines) = self.before {
            before = lines;
        }
        if let Some(lines) = self.after {
            after = lines;
        }
        (before, after)
    }
}
```

**Precedence examples:**

```
rg -C5 pattern           # (5, 5)
rg -C5 -B2 pattern       # (2, 5) — -B overrides -C's before
rg -B2 -C5 pattern       # (2, 5) — same! -B always wins
rg -C5 -A0 pattern       # (5, 0) — -A overrides -C's after
```

## Section 10: EngineChoice — Regex Implementation

```rust
/// The regex engine to use.
#[derive(Debug, Default, Eq, PartialEq)]
pub(crate) enum EngineChoice {
    /// Uses Rust's `regex` crate (default).
    #[default]
    Default,
    /// Try default, fall back to PCRE2 if pattern fails.
    Auto,
    /// Uses PCRE2 if available.
    PCRE2,
}
```

**When to use each:**

```
rg 'simple.*pattern'        # Default: fast, good errors
rg -P '(?<=foo)bar'         # PCRE2: lookbehind required
rg --auto-hybrid '(?<=x)y'  # Auto: try default, fall back
```

---

## Section 11: MmapMode — Memory Mapping Strategy

```rust
/// Indicates when to use memory maps.
#[derive(Debug, Default, Eq, PartialEq)]
pub(crate) enum MmapMode {
    /// Use heuristics to decide.
    #[default]
    Auto,
    /// Always try memory maps when possible.
    AlwaysTryMmap,
    /// Never use memory maps.
    Never,
}
```

**Heuristic factors (Auto mode):** - File count: mmap overhead hurts with many files - Input type: stdin/FIFOs can't be mmapped - Platform: mmap performance varies

---

## Section 12: PatternSource — Where Patterns Come From

```rust
/// Represents a source of patterns that ripgrep should search for.
#[derive(Debug, Eq, PartialEq)]
pub(crate) enum PatternSource {
    /// Comes from the `-e/--regexp` flag.
    Regexp(String),
    /// Comes from the `-f/--file` flag.
    File(PathBuf),
}
```

**Usage examples:**

```
rg foo                    # Positional → treated specially
rg -e foo -e bar          # Two Regexp sources
rg -f patterns.txt        # One File source
rg -e foo -f more.txt -e x # Mixed: [Regexp, File, Regexp]
```

---

## Section 13: SortMode — Result Ordering

```
/// The sort criteria, if present.
#[derive(Debug, Eq, PartialEq)]
pub(crate) struct SortMode {
    /// Whether to reverse (descending order).
    pub(crate) reverse: bool,
    /// The actual sorting criteria.
    pub(crate) kind: SortModeKind,
}

#[derive(Debug, Eq, PartialEq)]
pub(crate) enum SortModeKind {
    Path,
    LastModified,
    LastAccessed,
    Created,
}

impl SortMode {
    /// Check if sorting mode is supported on this platform.
    pub(crate) fn supported(&self) -> anyhow::Result<()> {
        match self.kind {
            SortModeKind::Path => Ok(()),
            SortModeKind::LastModified => {
                // Probe by checking current exe's metadata
                let md = std::env::current_exe()
                    .and_then(|p| p.metadata())
                    .and_then(|md| md.modified());
                let Err(err) = md else { return Ok(()) };
                anyhow::bail!("sorting by last modified isn't supported: {err}");
            }
            // Similar for LastAccessed, Created...
        }
    }
}
```

## Section 14: TypeChange — File Type Modifications

```rust
/// A single instance of a type change or selection.
#[derive(Debug, Eq, PartialEq)]
pub(crate) enum TypeChange {
    /// Clear the given type from ripgrep.
    Clear { name: String },
    /// Add a new type definition (name and glob).
    Add { def: String },
    /// Select the type for filtering (include).
    Select { name: String },
    /// Select the type for filtering but negate it (exclude).
    Negate { name: String },
}
```

**Command line → TypeChange:**

```
--type-clear=all       # Clear { name: "all" }
--type-add='foo:*.foo' # Add { def: "foo:*.foo" }
-t rust                # Select { name: "rust" }
-T python              # Negate { name: "python" }
```

**Order matters:**

```
# Clear all, add custom, then select it
rg --type-clear=all --type-add='mycode:*.mc' -t mycode pattern
```

## Section 15: Separator Types

```rust
/// Context separator between non-contiguous blocks (default: "--").
#[derive(Clone, Debug, Eq, PartialEq)]
pub(crate) struct ContextSeparator(Option<BString>);

impl ContextSeparator {
    /// Create from user input with escape handling.
    pub(crate) fn new(os: &OsStr) -> anyhow::Result<ContextSeparator> {
        let Some(string) = os.to_str() else {
            anyhow::bail!("separator must be valid UTF-8 (use escape
sequences)");
        };
        Ok(ContextSeparator(Some(Vec::unescape_bytes(string).into())))
    }

    /// Disable separators entirely.
    pub(crate) fn disabled() -> ContextSeparator {
        ContextSeparator(None)
    }
}

/// Field separator for context lines (default: "-").
pub(crate) struct FieldContextSeparator(BString);

/// Field separator for match lines (default: ":").
pub(crate) struct FieldMatchSeparator(BString);
```

**Escape sequence examples:**

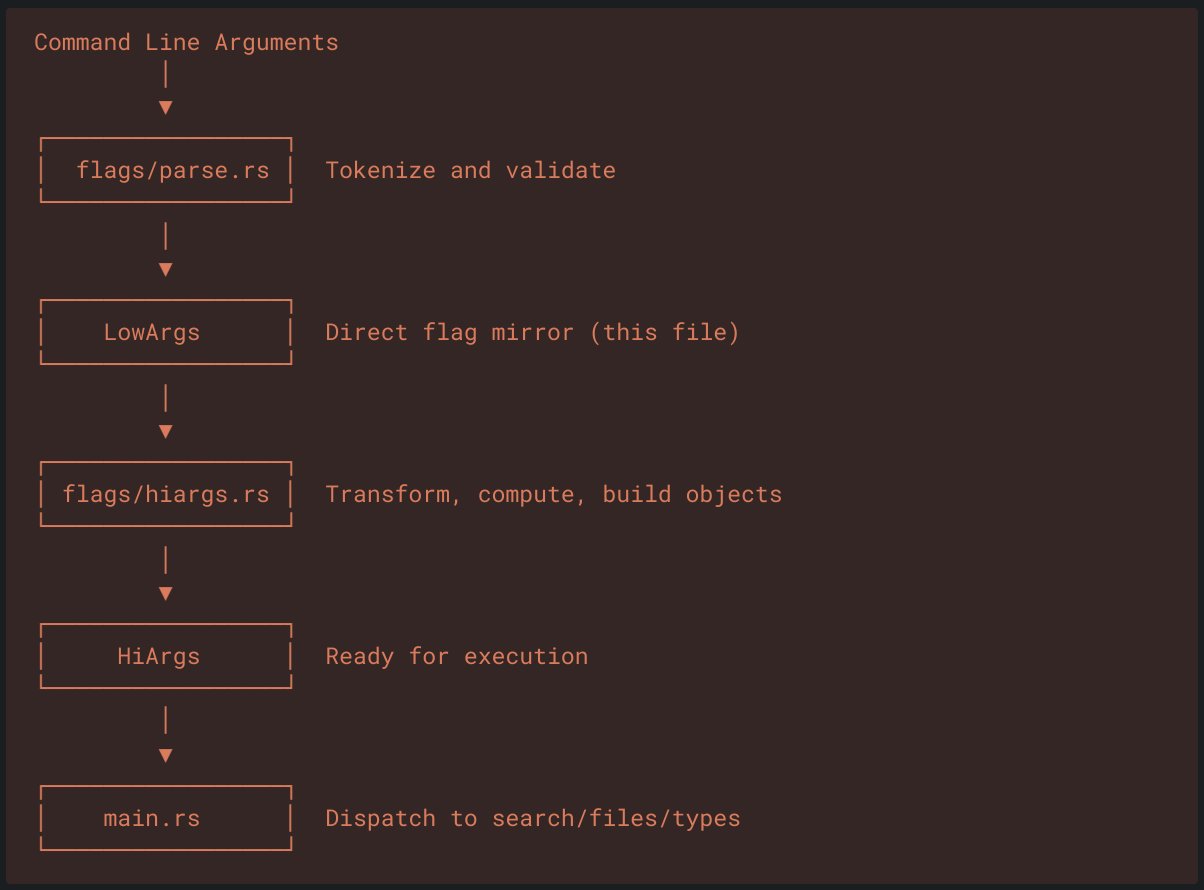
```
--context-separator=$'\t'      # Tab character
--context-separator='\x00'     # NUL byte
--context-separator=''         # Empty (no separator)
--no-context-separator         # Disabled entirely
```

## Quick Reference: Flag → Field Mapping

```
// Selected examples showing flag → LowArgs field

-i, --ignore-case      → case: CaseMode::Insensitive
-S, --smart-case       → case: CaseMode::Smart
-l, --files-with-matches → mode: Mode::Search(SearchMode::FilesWithMatches)
-c, --count            → mode: Mode::Search(SearchMode::Count)
--files                → mode: Mode::Files
-t, --type             → type_changes: Vec<TypeChange::Select>
-T, --type-not         → type_changes: Vec<TypeChange::Negate>
-g, --glob             → globs: Vec<String>
--iglob                → iglobs: Vec<String>
-j, --threads          → threads: Option<usize>
-A, --after-context    → context: ContextMode (set_after)
-B, --before-context   → context: ContextMode (set_before)
-C, --context          → context: ContextMode (set_both)
-e, --regexp           → patterns: Vec<PatternSource::Regexp>
-f, --file             → patterns: Vec<PatternSource::File>
-h                     → special: Some(SpecialMode::HelpShort)
--help                 → special: Some(SpecialMode::HelpLong)
```

## Data Flow: CLI to Execution



# Ripgrep lowargs.rs: The Raw Flag Mirror

## What This File Does

The lowargs.rs file defines LowArgs and its supporting types — a direct mirror of command-line flags. Where HiArgs represents computed, validated configuration, LowArgs represents exactly what the user typed. No heuristics, no derived values, no environment probing.

The file is roughly 720 lines, but most of that is type definitions. It's essentially a schema describing every flag ripgrep accepts and the valid values each can take.

---

## Section 1: The Design Philosophy

The module doc comment articulates a key constraint: "populating low level arguments should not require anything other than validating what the user has provided." LowArgs won't contain a hostname for hyperlinks because discovering the hostname requires syscalls or running binaries. That work belongs in HiArgs.

This separation serves robustness. Parsing LowArgs can fail if flags have invalid values, but those failures are predictable and local. HiArgs transformation can fail for environmental reasons — missing files, inaccessible directories, unsupported platform features. By separating these concerns, ripgrep ensures --help always works even when the environment is broken.

The practical benefit: you can always get help even if your current directory was deleted, your config file is corrupted, or you're on a weird platform where hostname detection fails.

See: Companion Code Section 1

---

## Section 2: The LowArgs Struct

The struct contains roughly 70 fields, one for each conceptual flag or flag group. Fields are public within the crate but not externally visible. The parser populates them; HiArgs reads them.

Fields group into categories. Essential arguments include the mode, positional arguments, and pattern sources. Display options cover colors, headings, line numbers. Search options handle case

sensitivity, word boundaries, multiline matching. Filter options control ignore rules, file types, and globs. Performance options set thread counts, memory limits, and mmap behavior.

The Default derive provides sensible starting values. Most booleans start false. Most Options start None. Enums typically default to their "auto" or "standard" variant.

See: Companion Code Section 2

## Section 3: SpecialMode — The Short-Circuit Cases

SpecialMode represents flags that bypass normal processing entirely. When present, ripgrep shows the requested information and exits immediately.

Five variants exist: HelpShort (-h), HelpLong (--help), VersionShort (-V), VersionLong (--version), and VersionPCRE2 (--pcre2-version). The short versus long distinction matters because they produce different output formats.

The comment explains why these get special treatment: "we avoid converting low-level argument types into higher level arguments types that can fail for various reasons related to the environment." Help should always work, even when nothing else does.

See: Companion Code Section 3

## Section 4: Mode — What Ripgrep Should Do

Mode represents the primary operation ripgrep will perform. Four variants cover the cases: Search (find matches), Files (list files that would be searched), Types (list file type definitions), and Generate (produce man pages or completions).

Search wraps a SearchMode that further distinguishes standard output, files-with-matches, counts, and JSON. This nesting reflects that most search variations share infrastructure while Generate modes are completely different code paths.

The update method implements override semantics. Once in a non-search mode, search modes can't override it. So "rg --files -l pattern" stays in Files mode despite -l normally enabling files-with-matches. This matches user expectations — explicit mode selection takes precedence.

See: Companion Code Section 4

## Section 5: SearchMode — Search Output Variations

SearchMode distinguishes how search results appear. Standard prints matching lines. FilesWithMatches prints only filenames containing matches. FilesWithoutMatch prints filenames without matches. Count prints match counts per file. CountMatches counts individual matches rather than matching lines. JSON outputs structured data.

There's no explicit flag for Standard — it's the default. Other modes have both enabling flags (-l, -c, --json) and negation flags (--no-json) to return to Standard.

The distinction between Count and CountMatches matters for patterns that match multiple times per line. Count reports one match per matching line. CountMatches reports the actual match count.

See: Companion Code Section 5

---

## Section 6: BinaryMode — Handling Non-Text Files

BinaryMode controls how ripgrep treats files containing binary data (specifically NUL bytes, which don't appear in text files).

Auto applies different strategies based on how the file was specified. Explicitly named files get searched with binary data converted or suppressed. Implicitly discovered files during traversal get skipped entirely when binary content is detected.

SearchAndSuppress searches the file but replaces NUL bytes with line terminators and suppresses match output, showing only a warning. This prevents memory issues from extremely long "lines" in binary files.

AsText (-a/--text) treats everything as text, disabling all binary detection. Useful when you know a file contains mostly text despite some binary content.

See: Companion Code Section 6

---

## Section 7: CaseMode — Pattern Matching Sensitivity

CaseMode controls whether pattern matching distinguishes uppercase from lowercase.

Sensitive is the default — 'a' matches only 'a'. Insensitive (-i) makes 'a' match both 'a' and 'A'. Smart (-S) enables case-insensitive matching only when the pattern contains no uppercase characters. The pattern 'foo' matches 'FOO', but 'Foo' matches only 'Foo'.

Smart case is particularly useful for interactive use. Searching for a common word doesn't require caring about case. Searching for a specific CamelCase identifier naturally becomes case-sensitive.

See: Companion Code Section 7

---

## Section 8: ColorChoice — Output Coloring

ColorChoice controls when ripgrep produces colored output with ANSI escape codes.

Never disables colors entirely. Always enables them unconditionally. Auto (the default) enables colors only when stdout is a terminal. Ansi forces ANSI escapes even on Windows where ripgrep might otherwise use legacy console APIs.

The to_termcolor method converts to the termcolor crate's equivalent enum. This conversion happens frequently since termcolor handles the actual escape code generation.

See: Companion Code Section 8

---

## Section 9: ContextMode — Lines Around Matches

ContextMode controls how many non-matching lines appear around each match for context.

Passthru (-p/--passthru) shows all lines, with matches highlighted. This is useful for viewing entire files with matches marked.

Limited specifies exact counts via ContextModeLimited, which tracks before, after, and both separately. The -B flag sets before context, -A sets after context, -C sets both. The separation matters because -B and -A always override -C regardless of flag order.

The get method in ContextModeLimited implements the precedence rules. If both is set to 5 but before is set to 2, you get 2 lines before and 5 lines after.

See: Companion Code Section 9

---

## Section 10: EngineChoice — Regex Implementation

EngineChoice selects which regex engine processes patterns.

Default uses Rust's regex crate (technically regex-automata). It's fast and has good error messages but lacks some advanced features.

PCRE2 uses the PCRE2 library, supporting lookaround assertions and backreferences. It's available only when ripgrep is compiled with the pcre2 feature.

Auto tries Default first, falling back to PCRE2 if compilation fails. This lets patterns requiring advanced features work without explicit engine selection.

See: Companion Code Section 10

---

## Section 11: MmapMode — Memory Mapping Strategy

MmapMode controls whether ripgrep uses memory-mapped I/O for reading files.

Auto (default) uses heuristics. Memory maps work well for a few large files but have overhead for many small files. The heuristic considers file count and whether all inputs are regular files.

AlwaysTryMmap forces memory mapping when possible. Some inputs can't be mapped (stdin, FIFOs), but regular files will use mmap.

Never disables memory mapping entirely. Even multiline mode, which needs the entire file in memory, will read into heap-allocated buffers instead.

See: Companion Code Section 11

---

## Section 12: PatternSource — Where Patterns Come From

PatternSource tracks pattern origins. Patterns can come from -e/--regexp flags (Regexp variant) or from files via -f/--file (File variant).

Why track the source? Order matters. Patterns from multiple sources are searched in the order provided. Also, patterns from files require reading those files, which might fail. The error message should indicate which file caused the problem.

The File variant stores the path. The Regexp variant stores the pattern string directly. Both get processed during the LowArgs to HiArgs transformation.

See: Companion Code Section 12

---

## Section 13: SortMode — Result Ordering

SortMode controls file processing order. The struct combines a sorting criterion (SortModeKind) with a reversal flag.

Four criteria exist: Path (alphabetical), LastModified, LastAccessed, and Created. Not all platforms support all criteria — the supported method checks by probing the current executable's metadata.

When sorting is enabled, parallelism is disabled. You can't sort results that arrive out of order. Path sorting ascending gets special optimization — the walker can sort during traversal.

See: Companion Code Section 13

---

## Section 14: TypeChange — File Type Modifications

TypeChange represents modifications to file type definitions. Ripgrep has built-in types (rust, python, c, etc.) that users can modify.

Clear removes a type definition entirely. Add creates a new definition with name and glob pattern. Select enables a type for filtering — only files matching that type get searched. Negate inverts a type — files matching are excluded.

These accumulate in a Vec because order matters. You might clear all types, add custom definitions, then select which to use. The Vec preserves that sequence for HiArgs to process.

See: Companion Code Section 14

## Section 15: Separator Types

Several types represent customizable separators with escape sequence support.

ContextSeparator (default "--") appears between non-contiguous context blocks.
FieldContextSeparator (default "-") separates metadata fields on context lines.
FieldMatchSeparator (default ":") separates metadata fields on match lines.

Each provides a new constructor that handles escape sequence unescaping. Users can specify "\t" for tabs, "\x00" for NUL bytes, etc. The BString type (from the bstr crate) holds arbitrary bytes, not just valid UTF-8.

See: Companion Code Section 15

---

## Key Takeaways

First, LowArgs is a direct mirror of CLI flags. No computation, no heuristics, no environmental queries.

Second, the separation between LowArgs and HiArgs isolates parsing failures from environmental failures. This ensures help always works.

Third, enum types with explicit variants make flag interactions clear. CaseMode::Smart is a distinct state, not a combination of other flags.

Fourth, order-sensitive operations (patterns, type changes) use Vec to preserve user intent.

Fifth, custom types handle validation at parse time. Invalid separator escapes fail immediately, not during search.

---

## What to Read Next

Understanding lowargs.rs completes the argument story:

How does the parser populate LowArgs? Read flags/parse.rs.

How does HiArgs transform these raw values? Read flags/hiargs.rs (already covered).

How are individual flags defined? Read flags/defs.rs.

# Ripgrep main.rs: Code Companion

Reference code for the main.rs lecture. Sections correspond to the lecture document.

---

## Section 1: The Allocator Override

```
// Conditional compilation: only on 64-bit musl targets
#[cfg(all(target_env = "musl", target_pointer_width = "64"))]
#[global_allocator]
static ALLOC: tikv_jemallocator::Jemalloc = tikv_jemallocator::Jemalloc;
```

**Attributes explained:** - `#[cfg(...)]` — Conditional compilation predicate - `target_env = "musl"` — Building against musl libc (common for static binaries) - `target_pointer_width = "64"` — 64-bit architecture (jemalloc doesn't support i686) - `#[global_allocator]` — Replaces the default allocator for the entire binary

---

## Section 2: Module Organization

```
#[macro_use]
mod messages;      // Must be first — exports macros used by other modules

mod flags;         // Two-level argument parsing (LowArgs → HiArgs)
mod haystack;      // Abstraction: "something to search"
mod logger;        // Debug logging via `log` crate
mod search;        // SearchWorker: coordinates matcher/searcher/printer
```

**Macros exported by messages:**

```
// Used throughout main.rs:
err_message!("{}: {}", path, err);      // Thread-safe error output
eprintln_locked!("{:#}", err);          // Thread-safe stderr with formatting
```

---

## Section 3: The Entry Point

```rust
fn main() -> ExitCode {
    match run(flags::parse()) {
        Ok(code) => code,
        Err(err) => {
            // Walk the error chain looking for BrokenPipe
            for cause in err.chain() {
                if let Some(ioerr) = cause.downcast_ref::<std::io::Error>() {
                    if ioerr.kind() == std::io::ErrorKind::BrokenPipe {
                        return ExitCode::from(0);  // Broken pipe = success
                    }
                }
            }
            eprintln_locked!("{:#}", err);  // {:#} shows full error chain
            ExitCode::from(2)               // Exit code 2 = error
        }
    }
}
```

**Error chain methods (from anyhow):**

```rust
err.chain()                             // Iterator over error + all causes
cause.downcast_ref::<T>()               // Try to cast to concrete type
```

**Exit codes (grep convention):**

```rust
ExitCode::from(0)  // Matches found (or broken pipe)
ExitCode::from(1)  // No matches found
ExitCode::from(2)  // Error occurred
```

## Section 4: The Dispatcher

```rust
fn run(result: crate::flags::ParseResult<HiArgs>) -> anyhow::Result<ExitCode> {
    use crate::flags::{Mode, ParseResult};

    // Three-phase parse result
    let args = match result {
        ParseResult::Err(err) => return Err(err),
        ParseResult::Special(mode) => return special(mode),  // --help, --version
        ParseResult::Ok(args) => args,
    };

    // Dispatch based on mode, with match guards for conditions
    let matched = match args.mode() {
        Mode::Search(_) if !args.matches_possible() => false,  // Early exit
        Mode::Search(mode) if args.threads() == 1 => search(&args, mode)?,
        Mode::Search(mode) => search_parallel(&args, mode)?,
        Mode::Files if args.threads() == 1 => files(&args)?,
        Mode::Files => files_parallel(&args)?,
        Mode::Types => return types(&args),
        Mode::Generate(mode) => return generate(mode),
    };

    // Determine exit code
    Ok(if matched && (args.quiet() || !messages::errored()) {
        ExitCode::from(0)
    } else if messages::errored() {
        ExitCode::from(2)
    } else {
        ExitCode::from(1)
    })
}
```

**ParseResult enum (conceptually):**

```rust
enum ParseResult<T> {
    Ok(T),              // Normal: here's your config
    Err(anyhow::Error), // Parse failed
    Special(SpecialMode), // Short-circuit: --help, --version
}
```

**HiArgs as configuration oracle:**

```rust
args.mode()              // What command to run
args.threads()           // Parallelism level
args.matches_possible()  // Can matches occur? (optimization)
args.quiet()             // Suppress output?
```

# Section 5: Single-Threaded Search

```rust
fn search(args: &HiArgs, mode: SearchMode) -> anyhow::Result<bool> {
    let started_at = std::time::Instant::now();
    let haystack_builder = args.haystack_builder();

    // Build the file iterator pipeline
    let unsorted = args
        .walk_builder()?           // Directory traversal with ignore rules
        .build()                   // Create the iterator
        .filter_map(|result| {     // Convert DirEntry → Haystack
            haystack_builder.build_from_result(result)
        });
    let haystacks = args.sort(unsorted);  // Optional sorting

    let mut matched = false;
    let mut searched = false;
    let mut stats = args.stats();  // Option<Stats>

    // Create the search coordinator
    let mut searcher = args.search_worker(
        args.matcher()?,           // Pattern matching engine
        args.searcher()?,          // File reading strategy
        args.printer(mode, args.stdout()),  // Output formatter
    )?;

    // Main search loop
    for haystack in haystacks {
        searched = true;
        let search_result = match searcher.search(&haystack) {
            Ok(search_result) => search_result,
            Err(err) if err.kind() == std::io::ErrorKind::BrokenPipe => break,
            Err(err) => {
                err_message!("{}: {}", haystack.path().display(), err);
                continue;  // Non-fatal: keep searching
            }
        };
        matched = matched || search_result.has_match();

        // Accumulate stats if enabled
        if let Some(ref mut stats) = stats {
            *stats += search_result.stats().unwrap();
        }

        // Early exit for -l (files-with-matches)
        if matched && args.quit_after_match() {
            break;
        }
    }

    Ok(matched)
}
```

**Objects created from HiArgs:**

| Method | Returns | Purpose |
|---|---|---|
| `haystack_builder()` | `HaystackBuilder` | Converts paths → searchable units |
| `walk_builder()` | `WalkBuilder` | Directory traversal + filtering |
| `matcher()` | `impl Matcher` | Regex engine |
| `searcher()` | `Searcher` | File I/O strategy |
| `printer()` | `impl Sink` | Output formatting |
| `search_worker()` | `SearchWorker` | Coordinates above three |

## Section 6: Parallel Search

```rust
fn search_parallel(args: &HiArgs, mode: SearchMode) -> anyhow::Result<bool> {
    use std::sync::atomic::{AtomicBool, Ordering};

    let started_at = std::time::Instant::now();
    let haystack_builder = args.haystack_builder();
    let bufwtr = args.buffer_writer();  // Thread-safe output buffer

    // Shared state
    let stats = args.stats().map(std::sync::Mutex::new);  // Protected by mutex
    let matched = AtomicBool::new(false);
    let searched = AtomicBool::new(false);

    // Create template searcher (will be cloned per thread)
    let mut searcher = args.search_worker(
        args.matcher()?,
        args.searcher()?,
        args.printer(mode, bufwtr.buffer()),
    )?;

    // Parallel directory walk
    args.walk_builder()?.build_parallel().run(|| {
        // Capture shared references
        let bufwtr = &bufwtr;
        let stats = &stats;
        let matched = &matched;
        let searched = &searched;
        let haystack_builder = &haystack_builder;

        // Clone searcher for this thread
        let mut searcher = searcher.clone();

        // Return the per-file handler
        Box::new(move |result| {
            // Convert to haystack
            let haystack = match haystack_builder.build_from_result(result) {
                Some(haystack) => haystack,
                None => return WalkState::Continue,
            };
            searched.store(true, Ordering::SeqCst);

            // Clear thread-local buffer
            searcher.printer().get_mut().clear();

            // Search
            let search_result = match searcher.search(&haystack) {
                Ok(search_result) => search_result,
                Err(err) => {
                    err_message!("{}: {}", haystack.path().display(), err);
                    return WalkState::Continue;
                }
            };

            // Update shared state
            if search_result.has_match() {
```

```
                matched.store(true, Ordering::SeqCst);            }            if
let Some(ref locked_stats) = *stats {                let mut stats =
locked_stats.lock().unwrap();                *stats +=
search_result.stats().unwrap();            }            // Flush buffer to stdout
atomically            if let Err(err) =
bufwtr.print(searcher.printer().get_mut()) {                if err.kind() ==
std::io::ErrorKind::BrokenPipe {                    return WalkState::Quit;
}                err_message!("{}: {}", haystack.path().display(), err);
}            // Continue or quit?            if matched.load(Ordering::SeqCst) &&
args.quit_after_match() {                WalkState::Quit            } else {
WalkState::Continue            }        })    });
Ok(matched.load(Ordering::SeqCst))}
```

**Parallel walk pattern:**

```
// Closure factory: called once per thread
walk_builder.build_parallel().run(|| {
    // Setup thread-local state here
    let mut local_state = something.clone();

    // Return the per-item handler
    Box::new(move |item| {
        // Process item using local_state
        WalkState::Continue  // or WalkState::Quit
    })
});
```

**Atomic operations:**

```
let flag = AtomicBool::new(false);

flag.store(true, Ordering::SeqCst);   // Write
flag.load(Ordering::SeqCst)           // Read
```

**Buffer writer pattern:**

```
let bufwtr = args.buffer_writer();
let buffer = bufwtr.buffer();         // Get thread-local buffer

// ... write to buffer ...

bufwtr.print(&mut buffer)?;           // Atomically flush to stdout
```

## Section 7: Error Handling Patterns

**Broken pipe = graceful exit:**

```rust
// Pattern appears throughout:
Err(err) if err.kind() == std::io::ErrorKind::BrokenPipe => break,

// In parallel mode:
if err.kind() == std::io::ErrorKind::BrokenPipe {
    return WalkState::Quit;
}
```

**Non-fatal errors continue:**

```rust
Err(err) => {
    err_message!("{}: {}", haystack.path().display(), err);
    continue;  // Keep searching other files
}
```

**Fatal errors bubble up:**

```rust
let searcher = args.search_worker(...)?;  // ? propagates error
```

---

## Section 8: Supporting Functions

**File listing (--files mode):**

```rust
fn files(args: &HiArgs) -> anyhow::Result<bool> {
    let haystack_builder = args.haystack_builder();
    let haystacks = args.sort(
        args.walk_builder()?.build()
            .filter_map(|r| haystack_builder.build_from_result(r))
    );

    let mut matched = false;
    let mut path_printer = args.path_printer_builder().build(args.stdout());

    for haystack in haystacks {
        matched = true;
        if args.quit_after_match() { break; }
        path_printer.write(haystack.path())?;
    }
    Ok(matched)
}
```

**Type listing (--type-list mode):**

```rust
fn types(args: &HiArgs) -> anyhow::Result<ExitCode> {
    let mut stdout = args.stdout();
    for def in args.types().definitions() {
        stdout.write_all(def.name().as_bytes())?;
        stdout.write_all(b": ")?;
        // ... write globs ...
    }
    Ok(ExitCode::from(0))
}
```

**Generation modes (--generate-*):**

```rust
fn generate(mode: GenerateMode) -> anyhow::Result<ExitCode> {
    let output = match mode {
        GenerateMode::Man => flags::generate_man_page(),
        GenerateMode::CompleteBash => flags::generate_complete_bash(),
        GenerateMode::CompleteZsh => flags::generate_complete_zsh(),
        GenerateMode::CompleteFish => flags::generate_complete_fish(),
        GenerateMode::CompletePowerShell =>
flags::generate_complete_powershell(),
    };
    writeln!(std::io::stdout(), "{}", output.trim_end())?;
    Ok(ExitCode::from(0))
}
```

**Special modes (--help, --version):**

```rust
fn special(mode: SpecialMode) -> anyhow::Result<ExitCode> {
    let output = match mode {
        SpecialMode::HelpShort => flags::generate_help_short(),
        SpecialMode::HelpLong => flags::generate_help_long(),
        SpecialMode::VersionShort => flags::generate_version_short(),
        SpecialMode::VersionLong => flags::generate_version_long(),
        SpecialMode::VersionPCRE2 => { /* special handling */ }
    };
    writeln!(std::io::stdout(), "{}", output.trim_end())?;
    Ok(ExitCode::from(0))
}
```

## Quick Reference: Key Types

```
// From flags module
struct HiArgs { /* validated configuration */ }
enum Mode { Search(SearchMode), Files, Types, Generate(GenerateMode) }
enum ParseResult<T> { Ok(T), Err(Error), Special(SpecialMode) }

// From ignore crate
enum WalkState { Continue, Quit, Skip }

// From search module
struct SearchWorker<M, S, P> { matcher: M, searcher: S, printer: P }

// Standard library
struct ExitCode;  // Returned from main()
struct AtomicBool;  // Lock-free boolean flag
```

# Ripgrep main.rs: The Orchestrator

## What This File Does

The main.rs file is ripgrep's entry point and orchestrator. It's surprisingly small for what ripgrep accomplishes — around 350 lines. That's because it delegates almost everything to library crates. Its job is to parse arguments, decide what mode to run in, and dispatch to the appropriate handler.

---

## Section 1: The Allocator Override

The file opens with a conditional allocator swap. On 64-bit musl builds only, ripgrep replaces the default system allocator with jemalloc.

Why does this matter? Musl is a lightweight C library often used for static linking. Its allocator prioritizes small code size and correctness over raw speed. For ripgrep's workload, this makes a noticeable difference. The interesting thing is that ripgrep isn't even allocation-heavy — the gains come from allocator quality: better cache behavior, less fragmentation.

This is conditional compilation at the deepest level. The resulting binary literally has different allocation code depending on the build target.

See: Companion Code Section 1

---

## Section 2: Module Organization

The file declares five internal modules. The messages module comes first with the macro_use attribute because it exports macros that other modules need.

The module breakdown follows a clear separation of concerns. Messages handles thread-safe error output. Flags contains the two-level argument parsing system. Haystack abstracts over "something to search" whether that's a file or stdin. Logger provides debug output via the standard log crate. Search contains the SearchWorker that coordinates the actual searching.

This is a common pattern in Rust CLI applications: the binary crate is thin, importing modules that could potentially be split into library crates later.

See: Companion Code Section 2

---

## Section 3: The Entry Point

The main function returns ExitCode, which is the modern Rust approach since version 1.61. It immediately calls run with the parsed arguments and handles the result.

The error handling here demonstrates an important Unix convention. When someone pipes ripgrep's output to another program like head, and that program exits early, the pipe breaks. In C programs, this triggers a SIGPIPE signal that terminates the process. Rust doesn't register for SIGPIPE, so instead you get an IO error with the BrokenPipe kind.

Ripgrep treats broken pipe as success, not failure. This matches user expectations — if you run "rg pattern | head -5", you want exit code zero even though ripgrep didn't finish writing all its output.

The error chain walking is also notable. When using anyhow for error handling, errors can be wrapped in layers. The chain method iterates through all the causes, letting you find a specific error type buried inside wrapper errors.

Exit codes follow grep convention: zero means matches were found, one means no matches, two means an error occurred.

See: Companion Code Section 3

---

## Section 4: The Dispatcher

The run function is the central dispatcher. It receives a ParseResult, which is a three-variant enum: Ok with the parsed arguments, Err with a parsing failure, or Special for short-circuit modes like help and version.

The Special variant is clever. When someone passes --help, you want to show help even if other initialization would fail. Maybe the current directory is inaccessible, or there's some configuration problem. By short-circuiting early, ripgrep ensures help is always available.

Once we have valid arguments, the HiArgs struct becomes the single source of truth. It's called HiArgs for "high-level arguments" — the result of resolving, validating, and computing derived values from the raw command-line flags.

The dispatch logic uses match guards to create a decision tree. First it checks if matches are even possible — with flags like --max-count=0, we know upfront no matches will occur. Then it checks thread count to choose between sequential and parallel implementations. Each mode — search, files, types, generate — has its own handler.

See: Companion Code Section 4

---

## Section 5: Single-Threaded Search

The sequential search function shows ripgrep's pipeline architecture. First, a walk builder creates an iterator over files to search. This handles directory recursion, gitignore rules, file type filtering, and all the other filtering logic.

Each directory entry gets converted to a Haystack through a builder. The haystack abstraction normalizes different input sources. A file on disk and stdin both become haystacks.

Optional sorting happens here. When you pass --sort, ripgrep collects all haystacks into a vector and sorts them. This necessarily happens in single-threaded mode because you can't sort results across parallel workers.

Then we construct a SearchWorker. This is the coordinator that holds three things: a matcher for pattern matching, a searcher for file reading, and a printer for output formatting. All three are created through HiArgs methods, which handle all the configuration translation.

The loop is straightforward. For each haystack, call search. Handle errors gracefully — broken pipe means stop, other errors get logged but searching continues. Accumulate statistics if requested.

See: Companion Code Section 5

---

## Section 6: Parallel Search

Parallel search is where things get interesting. The ignore crate provides a parallel directory walker. You give it a closure factory — a function that returns a closure — and it calls that factory once per thread.

Each thread needs its own SearchWorker, so we clone the worker inside the closure factory. Cloning is cheap because the expensive parts (the compiled regex) are shared via Arc internally.

Coordination between threads uses atomics. Two AtomicBool values track whether we found any matches and whether we searched anything. These use SeqCst ordering, which is the strongest memory ordering — probably overkill for boolean flags, but simple and safe.

Statistics accumulation is the one place requiring a mutex. Each thread's results get added to a shared Stats struct protected by a Mutex. This is acceptable because stats updates are infrequent compared to the actual searching.

Output uses a buffer writer pattern. Each thread writes its results to a private buffer, then atomically flushes that buffer to stdout. This prevents interleaved output from multiple threads. You never see half a line from one thread mixed with half a line from another.

The WalkState enum controls traversal. Returning Continue keeps searching. Returning Quit stops all threads. This matters for flags like -l (files-with-matches) where finding one match in a file is enough.

See: Companion Code Section 6

---

## Section 7: Error Handling Philosophy

Throughout the file, you'll notice consistent error handling patterns.

Broken pipe errors always cause graceful termination with success exit code. This check appears in every search function explicitly. It's not abstracted into a helper, making the behavior obvious at each call site.

Non-fatal errors log a message and continue. If ripgrep can't read one file due to permissions, it prints a warning and keeps searching other files. This matches user expectations for a recursive search tool.

Fatal errors bubble up via the question mark operator and ultimately cause exit code two.

See: Companion Code Section 7

---

## Section 8: Supporting Functions

The file ends with several supporting functions. The files and files_parallel functions handle --files mode, which lists files that would be searched without actually searching them. They follow the same sequential versus parallel pattern as the search functions.

The types function handles --type-list, printing all known file type definitions.

The generate function produces man pages and shell completions. These are generated from the flag definitions, ensuring documentation stays synchronized with the actual command-line interface.

The special function handles help and version output. It's separate from generate because these modes short-circuit argument parsing entirely.

See: Companion Code Section 8

---

## Key Takeaways

First, the binary is thin. All the real work happens in library crates. Main.rs is purely orchestration.

Second, HiArgs is the configuration oracle. Every decision flows from this struct. It encapsulates all the complexity of argument interaction and precedence.

Third, parallelism is opt-in but automatic. Users don't choose parallel mode; ripgrep detects whether parallelism is appropriate based on thread count and sorting requirements.

Fourth, error handling is explicit and consistent. Broken pipe is success. File errors continue. Parse errors exit.

Fifth, the closure-factory pattern enables parallel iteration while maintaining thread-local state.

---

## What to Read Next

Understanding main.rs raises questions that the following files answer:

How does HiArgs actually resolve all those flags? Read flags/hiargs.rs.

What does SearchWorker.search actually do? Read search.rs.

How does the haystack abstraction work? Read haystack.rs.

How do the locked printing macros work? Read messages.rs.

# Ripgrep messages.rs: Code Companion

Reference code for the messages.rs lecture. Sections correspond to the lecture document.

---

## Section 1: The Problem Being Solved

```
// Without synchronization, concurrent writes interleave:
// Thread 1: eprintln!("rg: file1.txt: permission denied");
// Thread 2: eprintln!("rg: file2.txt: not found");
// Output: "rg: file1.trg:x t:f ilpee2r.mtixsts:i opne rdmenisesdi oenn ideed"

// Even locking per-call doesn't help:
// Thread 1: lock stderr, write "rg: ", unlock
// Thread 2: lock stderr, write "rg: ", unlock  <- interleaves here
// Thread 1: lock stderr, write "file1.txt...", unlock
```

**The solution:** Hold a single lock across the entire message output.

---

## Section 2: The Global State

```rust
use std::sync::atomic::{AtomicBool, Ordering};

/// When false, "messages" will not be printed.
static MESSAGES: AtomicBool = AtomicBool::new(false);

/// When false, "messages" related to ignore rules will not be printed.
static IGNORE_MESSAGES: AtomicBool = AtomicBool::new(false);

/// Flipped to true when an error message is printed.
static ERRORED: AtomicBool = AtomicBool::new(false);
```

**Flag purposes:**

| Flag | Default | Controlled By | Purpose |
|---|---|---|---|

| | | | |
|---|---|---|---|
| `MESSAGES` | false | `--no-messages` | Master switch for all messages |
| `IGNORE_MESSAGES` | false | `--no-ignore-messages` | Ignore file parse errors |
| `ERRORED` | false | `err_message!` macro | Track if any error occurred |

**Lifecycle:**

```
Program start     → All flags false
Argument parsing  → MESSAGES, IGNORE_MESSAGES configured
Parallel search   → ERRORED may flip to true
Program exit      → ERRORED checked for exit status
```

## Section 3: Memory Ordering

```rust
// All operations use Relaxed ordering
MESSAGES.load(Ordering::Relaxed)
MESSAGES.store(yes, Ordering::Relaxed)
```

**Why Relaxed is safe here:**

```
Timeline:
──────────────────────────────────────────────────────────
  | Single-threaded init | Parallel search phase | Exit |
──────────────────────────────────────────────────────────
  ↑                      ↑                       ↑
  Flags set here         Flags read here         ERRORED read

  No concurrent writes during reads → Relaxed is sufficient
```

**Memory ordering comparison:**

| Ordering | Guarantees | Use Case |
|---|---|---|
| `Relaxed` | Atomicity only | Flags set before threads spawn |
| `Acquire/Release` | Synchronizes memory | Producer/consumer patterns |

| | | |
|---|---|---|
| `SeqCst` | Total ordering | When order matters across threads |

## Section 4: The eprintln_locked Macro

```
/// Like eprintln, but locks stdout to prevent interleaving lines.
///
/// This locks stdout, not stderr, even though this prints to stderr. This
/// avoids the appearance of interleaving output when stdout and stderr both
/// correspond to a tty.
#[macro_export]
macro_rules! eprintln_locked {
    ($($tt:tt)*) => {{
        {
            use std::io::Write;

            // This is a bit of an abstraction violation because we explicitly
            // lock stdout before printing to stderr. This avoids interleaving
            // lines within ripgrep because `search_parallel` uses `termcolor`,
            // which accesses the same stdout lock when writing lines.
            let stdout = std::io::stdout().lock();
            let mut stderr = std::io::stderr().lock();

            // Error handling for write operations...
            if let Err(err) = write!(stderr, "rg: ") {
                // ... handle error
            }
            if let Err(err) = writeln!(stderr, $($tt)*) {
                // ... handle error
            }

            drop(stdout);  // Explicit drop for clarity
        }
    }}
}
```

**Why lock stdout for stderr writes:**

```
Search thread (via termcolor):     Error thread:
─────────────────────────────      ─────────────

lock(stdout)
write match to stdout
                                   lock(stdout) ← blocks here
unlock(stdout)
                                   lock(stderr)
                                   write error to stderr
                                   unlock(stderr)
                                   unlock(stdout)

Result: Match output completes before error appears
```

**The macro pattern `$($tt:tt)*`:**

```rust
// $tt:tt matches any single token tree
// $($tt:tt)* matches zero or more token trees
// This captures arbitrary format arguments:

eprintln_locked!("simple message");
eprintln_locked!("{}: {}", path, err);
eprintln_locked!("{:#}", err);  // Pretty-print with error chain
```

---

## Section 5: Broken Pipe Handling

```rust
// Within eprintln_locked:
if let Err(err) = write!(stderr, "rg: ") {
    if err.kind() == std::io::ErrorKind::BrokenPipe {
        std::process::exit(0);   // Broken pipe = success
    } else {
        std::process::exit(2);   // Other error = failure
    }
}
if let Err(err) = writeln!(stderr, $($tt)*) {
    if err.kind() == std::io::ErrorKind::BrokenPipe {
        std::process::exit(0);
    } else {
        std::process::exit(2);
    }
}
```

**Broken pipe scenario:**

```
$ rg pattern 2>&1 | head -5
# head exits after 5 lines
# ripgrep's next write fails with BrokenPipe
# Exit code 0 (success) because head got what it needed
```

**Exit codes:**

```
exit(0)  // Success (matches found, or broken pipe)
exit(1)  // No matches found
exit(2)  // Error occurred
```

---

## Section 6: The message Macro

```
/// Emit a non-fatal error message, unless messages were disabled.
#[macro_export]
macro_rules! message {
    ($($tt:tt)*) => {
        if crate::messages::messages() {
            eprintln_locked!($($tt)*);
        }
    }
}
```

**Conditional output:**

```
// With --no-messages flag:
set_messages(false);
message!("{}: {}", path, err);  // No output, but expression evaluated

// Default behavior:
set_messages(true);
message!("{}: {}", path, err);  // Prints: "rg: path: err"
```

**Note on macro hygiene:**

```
// #[macro_export] makes the macro available at crate root
// Usage elsewhere: crate::message! or just message! with #[macro_use]
```

---

## Section 7: The err_message Macro

```rust
/// Like message, but sets ripgrep's "errored" flag, which controls the exit
/// status.
#[macro_export]
macro_rules! err_message {
    ($($tt:tt)*) => {
        crate::messages::set_errored();  // Always set, even if messages disabled
        message!($($tt)*);               // Conditionally print
    }
}
```

**Exit status logic in main.rs:**

```rust
// Simplified from actual code:
Ok(if matched && !messages::errored() {
    ExitCode::from(0)   // Matches found, no errors
} else if messages::errored() {
    ExitCode::from(2)   // Error occurred
} else {
    ExitCode::from(1)   // No matches found
})
```

**Why separate set_errored from printing:**

```rust
// Error tracking happens even with --no-messages
err_message!("{}: {}", path, err);
// ERRORED = true (always)
// Message printed only if MESSAGES = true
```

---

## Section 8: The ignore_message Macro

```rust
/// Emit a non-fatal ignore-related error message (like a parse error), unless
/// ignore-messages were disabled.
#[macro_export]
macro_rules! ignore_message {
    ($($tt:tt)*) => {
        if crate::messages::messages() && crate::messages::ignore_messages() {
            eprintln_locked!($($tt)*);
        }
    }
}
```

**Two-level gating:**

```
MESSAGES = true,  IGNORE_MESSAGES = true  → prints
MESSAGES = true,  IGNORE_MESSAGES = false → silent
MESSAGES = false, IGNORE_MESSAGES = true  → silent
MESSAGES = false, IGNORE_MESSAGES = false → silent
```

**Typical ignore message:**

```
ignore_message!("{}: {}", gitignore_path, parse_error);
// Output: "rg: .gitignore: invalid glob pattern on line 5"
```

---

## Section 9: The Accessor Functions

```rust
/// Returns true if and only if messages should be shown.
pub(crate) fn messages() -> bool {
    MESSAGES.load(Ordering::Relaxed)
}

/// Set whether messages should be shown or not.
///
/// By default, they are not shown.
pub(crate) fn set_messages(yes: bool) {
    MESSAGES.store(yes, Ordering::Relaxed)
}

/// Returns true if and only if "ignore" related messages should be shown.
pub(crate) fn ignore_messages() -> bool {
    IGNORE_MESSAGES.load(Ordering::Relaxed)
}

/// Set whether "ignore" related messages should be shown or not.
///
/// By default, they are not shown.
///
/// Note that this is overridden if `messages` is disabled.
pub(crate) fn set_ignore_messages(yes: bool) {
    IGNORE_MESSAGES.store(yes, Ordering::Relaxed)
}

/// Returns true if and only if ripgrep came across a non-fatal error.
pub(crate) fn errored() -> bool {
    ERRORED.load(Ordering::Relaxed)
}

/// Indicate that ripgrep has come across a non-fatal error.
///
/// Callers should not use this directly. Instead, it is called automatically
/// via the `err_message` macro.
pub(crate) fn set_errored() {
    ERRORED.store(true, Ordering::Relaxed);
}
```

**Visibility:**

```rust
pub(crate)  // Visible within the ripgrep binary crate only
            // Not exported to library users
```

## Section 10: Usage Patterns in main.rs

**Initialization during argument parsing:**

```
// In hiargs.rs or similar:
messages::set_messages(!low.no_messages);
messages::set_ignore_messages(!low.no_ignore_messages);
```

**Error handling in search loop:**

```
// From main.rs search function:
let search_result = match searcher.search(&haystack) {
    Ok(search_result) => search_result,
    Err(err) if err.kind() == std::io::ErrorKind::BrokenPipe => break,
    Err(err) => {
        err_message!("{}: {}", haystack.path().display(), err);
        continue;  // Non-fatal: keep searching other files
    }
};
```

**Exit status determination:**

```
// From main.rs run function:
Ok(if matched && (args.quiet() || !messages::errored()) {
    ExitCode::from(0)
} else if messages::errored() {
    ExitCode::from(2)
} else {
    ExitCode::from(1)
})
```

## Quick Reference: Macro Invocations

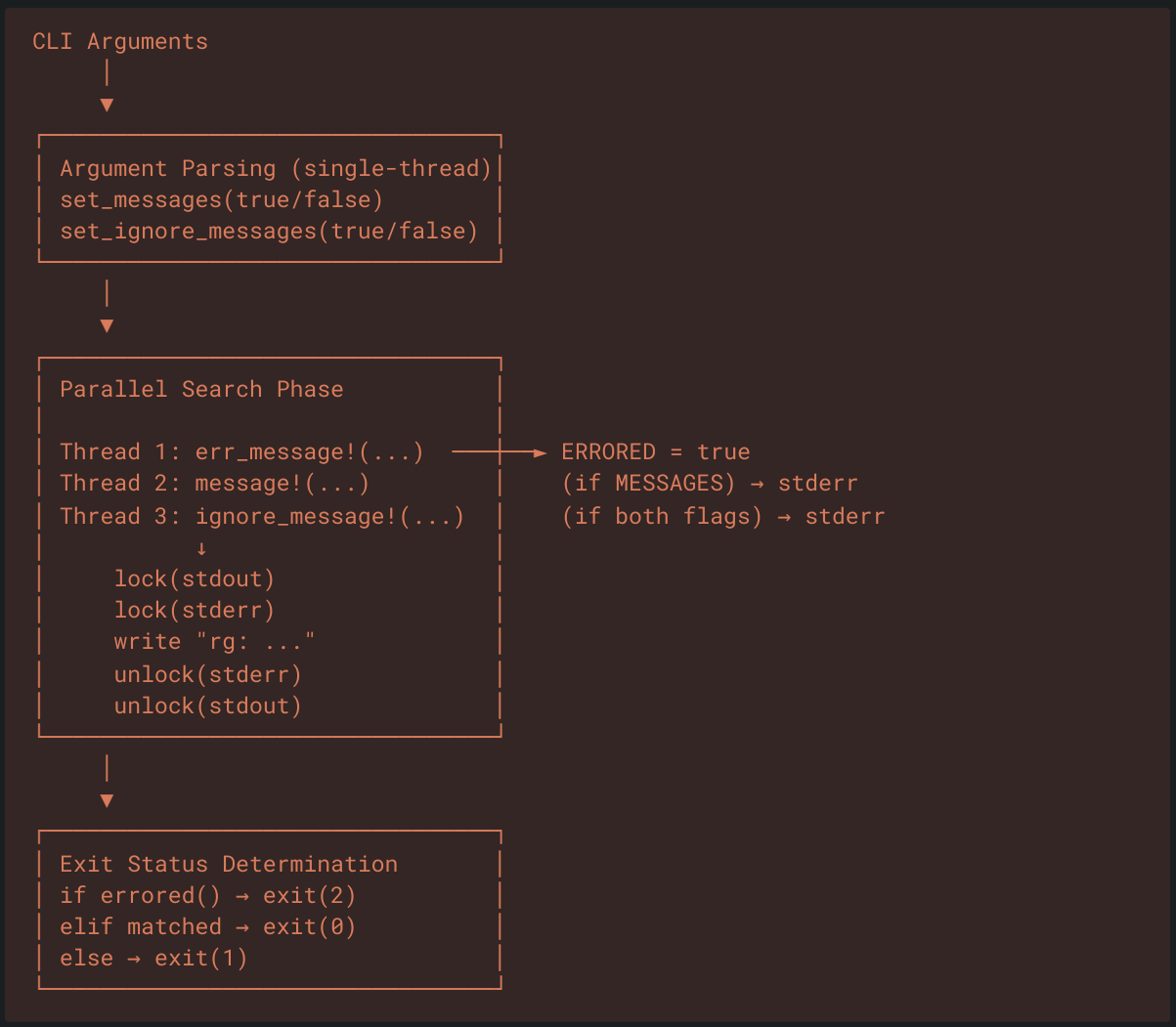
```
// Thread-safe stderr with "rg: " prefix, always outputs
eprintln_locked!("message");
eprintln_locked!("{}: {}", path, err);
eprintln_locked!("{:#}", anyhow_err);  // Full error chain

// Conditional on MESSAGES flag
message!("warning: {}", msg);

// Sets ERRORED flag, conditional output on MESSAGES
err_message!("{}: {}", path, err);

// Conditional on both MESSAGES and IGNORE_MESSAGES
ignore_message!("{}: {}", ignore_path, parse_err);
```

## Data Flow Summary




## Comparison with Standard Macros

```
// Standard library (not thread-safe for multi-line):
eprintln!("rg: {}", msg);

// This module (thread-safe):
eprintln_locked!("{}", msg);  // Adds "rg: " prefix automatically

// Key differences:
// 1. Locks stdout AND stderr
// 2. Prefixes all output with "rg: "
// 3. Handles broken pipe gracefully
// 4. Exits on write errors
```

# Ripgrep messages.rs: Thread-Safe Output Infrastructure

## What This File Does

The messages.rs module provides thread-safe error and warning output for ripgrep. It's only 139 lines, but it solves a subtle problem: when multiple threads write to stderr simultaneously, their output can interleave at the character level, producing garbled messages.

The module also tracks global state: whether messages should be displayed at all, and whether any errors occurred during execution. This error flag ultimately affects ripgrep's exit status.

---

## Section 1: The Problem Being Solved

When ripgrep runs in parallel mode, multiple threads search files simultaneously. Each thread might encounter errors — permission denied, file not found, encoding issues. If threads write error messages concurrently, the output becomes unreadable.

Consider two threads printing errors at the same time. Without synchronization, you might see: "rg: file1.trg:x t:f ilpee2r.mtixsts:i opne rdmenisesdi oenn ideed" — a jumbled mess of two messages interleaved character by character.

Standard library print macros don't prevent this. They lock stderr for each print call, but two writeln! calls from different threads can still interleave between calls. The solution requires holding a lock across the entire message.

See: Companion Code Section 1

---

## Section 2: The Global State

Three AtomicBool values track global state. MESSAGES controls whether error messages display at all — the --no-messages flag disables them. IGNORE_MESSAGES controls whether ignore-rule parse errors display — useful for debugging gitignore issues. ERRORED tracks whether any error occurred during execution.

All three start as false and get configured once during argument parsing. After that, MESSAGES and IGNORE_MESSAGES are read-only. ERRORED starts false and gets flipped to true if any error occurs, never back to false.

The choice of AtomicBool over other synchronization primitives reflects the access pattern. These flags are set once, then read many times from many threads. Atomic operations are the lightest-weight synchronization that provides the necessary guarantees.

See: Companion Code Section 2

---

## Section 3: Memory Ordering

All atomic operations use Ordering::Relaxed, the weakest memory ordering. This might seem surprising for shared state, but it's correct here.

Relaxed ordering guarantees atomicity — reads and writes won't tear — but provides no synchronization between threads. A write on one thread might not be immediately visible to reads on other threads.

This works for ripgrep because the flags are set during single-threaded initialization, before any parallel searching begins. By the time multiple threads read the flags, the writes have long since completed and propagated. There's no race between setting and reading.

For ERRORED, relaxed ordering means a late-arriving error might not be visible when exit status is determined. In practice, this doesn't matter — the error message itself was printed, and missing the exit status change is a minor inconsistency in an already-failing run.

See: Companion Code Section 3

---

## Section 4: The eprintln_locked Macro

The eprintln_locked macro is the foundation of thread-safe output. Its implementation reveals a subtle trick: it locks stdout before writing to stderr.

This seems backwards until you consider how ripgrep's parallel search works. The search threads use termcolor to write colored output to stdout. Termcolor acquires stdout's lock when writing. Meanwhile, error messages go to stderr.

If both stdout and stderr point to the same terminal, interleaving can occur even between different streams. By acquiring stdout's lock first, error messages wait for any in-progress search output to complete. The output appears sequential even though it's generated concurrently.

The "abstraction violation" comment acknowledges this coupling. The macro knows implementation details of the search code — specifically that it uses termcolor and stdout's lock. If that changed, this macro would need updating.

See: Companion Code Section 4

---

## Section 5: Broken Pipe Handling

Within eprintln_locked, broken pipe errors receive special treatment. If writing fails with BrokenPipe, the process exits with code zero. Any other write error exits with code two.

Broken pipe occurs when the receiving end of a pipe closes. If someone runs "rg pattern 2>&1 | head -5", the head command exits after five lines. The pipe to head's stdin breaks. Ripgrep's next write to stdout or stderr fails with BrokenPipe.

Treating this as success (exit code zero) matches Unix conventions. The user got what they asked for — head received its five lines. Ripgrep's inability to write more isn't an error in the meaningful sense.

The exit calls within the macro are somewhat unusual. Macros typically expand to expressions, not control flow. But here, a failed write is unrecoverable — there's nothing sensible to do except exit.

See: Companion Code Section 5

---

## Section 6: The message Macro

The message macro conditionally prints messages based on the MESSAGES flag. It's a thin wrapper around eprintln_locked that checks the flag first.

When --no-messages is passed, MESSAGES is false, and this macro becomes a no-op. The formatting expressions still get evaluated (Rust macros don't short-circuit argument evaluation), but no I/O occurs.

This conditional display matters for scripting. When ripgrep runs as part of a pipeline, error messages might interfere with parsing downstream. Suppressing them keeps output clean while

still recording that errors occurred (via the ERRORED flag).

See: Companion Code Section 6

---

## Section 7: The err_message Macro

The err_message macro extends message with error tracking. Before printing, it sets the ERRORED flag. This flag persists and affects the final exit status.

The separation between message and err_message matters for categorization. Some messages are informational — warnings about unusual conditions that don't indicate failure. Error messages indicate something went wrong that the user should know about.

Exit status conventions distinguish these cases. A search that found matches but encountered errors returns exit code two, not zero. The err_message macro ensures these errors get counted even if message display is suppressed.

See: Companion Code Section 7

---

## Section 8: The ignore_message Macro

The ignore_message macro handles a specific category: parse errors in ignore files. These occur when .gitignore, .rgignore, or similar files contain invalid syntax.

The macro checks both MESSAGES and IGNORE_MESSAGES flags. Both must be true for output to occur. This two-level gating lets users suppress ignore warnings specifically while keeping other error messages visible.

Why separate ignore messages? Ignore file syntax errors are common and often benign. A .gitignore written for a different tool might use unsupported syntax. Flooding the terminal with these warnings during every search would be annoying.

See: Companion Code Section 8

---

## Section 9: The Accessor Functions

Four pairs of functions provide controlled access to the global flags. The accessor functions (messages, ignore_messages, errored) read the current state. The mutator functions (set_messages, set_ignore_messages, set_errored) modify it.

The pub(crate) visibility restricts access to within the ripgrep binary. External code can't manipulate these flags — they're internal implementation details.

The set_errored function's doc comment notes that callers shouldn't use it directly. The err_message macro handles the common case. Direct calls might be needed for errors that don't warrant a message, but those are rare.

See: Companion Code Section 9

---

## Section 10: Usage Patterns in main.rs

Looking at how main.rs uses these macros reveals the design's purpose.

In the search loop, file access errors use err_message. The search continues, but the error is recorded. At program exit, if ERRORED is true and the search otherwise succeeded, exit code becomes two instead of zero.

The broken pipe check appears both in the macro and in explicit checks throughout main.rs. The macro handles errors during error reporting — a meta-level concern. The explicit checks handle broken pipe during normal search output.

The initialization code in argument parsing calls set_messages and set_ignore_messages based on flag values. This happens once, before any parallelism, establishing the configuration that all threads will read.

See: Companion Code Section 10

---

## Key Takeaways

First, thread-safe output requires holding locks across entire messages, not just individual writes.

Second, locking stdout before writing to stderr prevents interleaving when both streams go to the same terminal.

Third, global atomic flags work well for configuration set once and read many times.

Fourth, relaxed memory ordering suffices when writes complete before reads begin.

Fifth, exit status tracking separate from message display enables clean scripting while preserving error information.

---

## What to Read Next

Understanding messages.rs completes the core module picture. The remaining questions involve the library crates:

How does termcolor manage colored output and interact with these locks? Read the termcolor crate.

How does the parallel walker coordinate with message output? Read the ignore crate's parallel module.

How do the actual search errors get generated? Read grep-searcher's error handling.

# Ripgrep search.rs: Code Companion

Reference code for the search.rs lecture. Sections correspond to the lecture document.

---

## Section 1: The Config Struct

```rust
/// The configuration for the search worker.
///
/// Among a few other things, the configuration primarily controls the way we
/// show search results to users at a very high level.
#[derive(Clone, Debug)]
struct Config {
    preprocessor: Option<std::path::PathBuf>,
    preprocessor_globs: ignore::overrides::Override,
    search_zip: bool,
    binary_implicit: grep::searcher::BinaryDetection,
    binary_explicit: grep::searcher::BinaryDetection,
}

impl Default for Config {
    fn default() -> Config {
        Config {
            preprocessor: None,
            preprocessor_globs: ignore::overrides::Override::empty(),
            search_zip: false,
            binary_implicit: grep::searcher::BinaryDetection::none(),
            binary_explicit: grep::searcher::BinaryDetection::none(),
        }
    }
}
```

**Config field purposes:**

| Field | Type | Purpose |
|---|---|---|
| `preprocessor` | `Option<PathBuf>` | External command to transform files |
| `preprocessor_globs` | `Override` | Which files to preprocess |

| | | |
|---|---|---|
| `search_zip` | `bool` | Auto-decompress compressed files |
| `binary_implicit` | `BinaryDetection` | Detection for discovered files |
| `binary_explicit` | `BinaryDetection` | Detection for user-specified files |

## Section 2: The Builder Pattern

```
/// A builder for configuring and constructing a search worker.
#[derive(Clone, Debug)]
pub(crate) struct SearchWorkerBuilder {
    config: Config,
    command_builder: grep::cli::CommandReaderBuilder,
}

impl SearchWorkerBuilder {
    /// Create a new builder for configuring and constructing a search worker.
    pub(crate) fn new() -> SearchWorkerBuilder {
        let mut command_builder = grep::cli::CommandReaderBuilder::new();
        command_builder.async_stderr(true);

        SearchWorkerBuilder { config: Config::default(), command_builder }
    }

    /// Create a new search worker using the given searcher, matcher and
    /// printer.
    pub(crate) fn build<W: WriteColor>(
        &self,
        matcher: PatternMatcher,
        searcher: grep::searcher::Searcher,
        printer: Printer<W>,
    ) -> SearchWorker<W> {
        let config = self.config.clone();
        let command_builder = self.command_builder.clone();

        // Lazy initialization: only build decompressor when enabled
        let decomp_builder = config.search_zip.then(|| {
            let mut decomp_builder =
                grep::cli::DecompressionReaderBuilder::new();
            decomp_builder.async_stderr(true);
            decomp_builder
        });

        SearchWorker {
            config,
            command_builder,
            decomp_builder,
            matcher,
            searcher,
            printer,
        }
    }
}
```

**The .then() pattern:**

```
// Option<T> if condition is true, None otherwise
let maybe_builder = condition.then(|| expensive_construction());
```

# Section 3: The Builder Methods

```
impl SearchWorkerBuilder {
    /// Set the path to a preprocessor command.
    pub(crate) fn preprocessor(
        &mut self,
        cmd: Option<std::path::PathBuf>,
    ) -> anyhow::Result<&mut SearchWorkerBuilder> {
        if let Some(ref prog) = cmd {
            // Validate and resolve binary path early
            let bin = grep::cli::resolve_binary(prog)?;
            self.config.preprocessor = Some(bin);
        } else {
            self.config.preprocessor = None;
        }
        Ok(self)
    }

    /// Set the globs for determining which files should be run through the
    /// preprocessor.
    pub(crate) fn preprocessor_globs(
        &mut self,
        globs: ignore::overrides::Override,
    ) -> &mut SearchWorkerBuilder {
        self.config.preprocessor_globs = globs;
        self
    }

    /// Enable the decompression and searching of common compressed files.
    pub(crate) fn search_zip(
        &mut self,
        yes: bool,
    ) -> &mut SearchWorkerBuilder {
        self.config.search_zip = yes;
        self
    }

    /// Binary detection for implicitly discovered files (directory traversal).
    /// Typically uses BinaryDetection::quit() to skip binary files.
    pub(crate) fn binary_detection_implicit(
        &mut self,
        detection: grep::searcher::BinaryDetection,
    ) -> &mut SearchWorkerBuilder {
        self.config.binary_implicit = detection;
        self
    }

    /// Binary detection for explicitly requested files.
    /// Should NOT use quit() — never skip user-requested files.
    pub(crate) fn binary_detection_explicit(
        &mut self,
        detection: grep::searcher::BinaryDetection,
    ) -> &mut SearchWorkerBuilder {
        self.config.binary_explicit = detection;
        self
    }
```

```
}
```

**BinaryDetection modes:**

```
BinaryDetection::none()     // No detection, search everything
BinaryDetection::quit()     // Stop searching if binary detected
BinaryDetection::convert()  // Replace NUL bytes with replacement char
```

---

## Section 4: SearchResult

```rust
/// The result of executing a search.
#[derive(Clone, Debug, Default)]
pub(crate) struct SearchResult {
    has_match: bool,
    stats: Option<grep::printer::Stats>,
}

impl SearchResult {
    /// Whether the search found a match or not.
    pub(crate) fn has_match(&self) -> bool {
        self.has_match
    }

    /// Return aggregate search statistics for a single search, if available.
    ///
    /// It can be expensive to compute statistics, so these are only present
    /// if explicitly enabled in the printer provided by the caller.
    pub(crate) fn stats(&self) -> Option<&grep::printer::Stats> {
        self.stats.as_ref()
    }
}
```

**Exit code mapping:**

```
// In main.rs
if search_result.has_match() {
    ExitCode::from(0)  // Matches found
} else {
    ExitCode::from(1)  // No matches
}
```

---

## Section 5: PatternMatcher Enum

```rust
/// The pattern matcher used by a search worker.
#[derive(Clone, Debug)]
pub(crate) enum PatternMatcher {
    RustRegex(grep::regex::RegexMatcher),
    #[cfg(feature = "pcre2")]
    PCRE2(grep::pcre2::RegexMatcher),
}
```

**Conditional compilation:**

```rust
// When pcre2 feature is disabled, this variant doesn't exist
#[cfg(feature = "pcre2")]
PCRE2(grep::pcre2::RegexMatcher),

// Usage in match expressions:
match self.matcher {
    RustRegex(ref m) => /* ... */,
    #[cfg(feature = "pcre2")]
    PCRE2(ref m) => /* ... */,
}
```

**Why enum over trait object:**

```rust
// Enum approach (used here):
match matcher { RustRegex(m) => m.search(), PCRE2(m) => m.search() }
// Pro: inlinable, no vtable
// Con: closed set of variants

// Trait object approach:
dyn Matcher
// Pro: open to extension
// Con: virtual dispatch overhead
```

---

## Section 6: Printer Enum

```rust
/// The printer used by a search worker.
///
/// The `W` type parameter refers to the type of the underlying writer.
#[derive(Clone, Debug)]
pub(crate) enum Printer<W> {
    /// Use the standard printer, which supports the classic grep-like format.
    Standard(grep::printer::Standard<W>),
    /// Use the summary printer, which supports aggregate displays of search
    /// results.
    Summary(grep::printer::Summary<W>),
    /// A JSON printer, which emits results in the JSON Lines format.
    JSON(grep::printer::JSON<W>),
}

impl<W: WriteColor> Printer<W> {
    /// Return a mutable reference to the underlying printer's writer.
    pub(crate) fn get_mut(&mut self) -> &mut W {
        match *self {
            Printer::Standard(ref mut p) => p.get_mut(),
            Printer::Summary(ref mut p) => p.get_mut(),
            Printer::JSON(ref mut p) => p.get_mut(),
        }
    }
}
```

**Printer types by mode:**

| Flag | Printer Type | Purpose |
|---|---|---|
| (default) | Standard | Line-by-line matches |
| `-c` | Summary | Match counts |
| `-l` | Summary | Filenames only |
| `--json` | JSON | Structured output |

## Section 7: The SearchWorker Struct

```rust
/// A worker for executing searches.
///
/// It is intended for a single worker to execute many searches, and is
/// generally intended to be used from a single thread.
#[derive(Clone, Debug)]
pub(crate) struct SearchWorker<W> {
    config: Config,
    command_builder: grep::cli::CommandReaderBuilder,
    decomp_builder: Option<grep::cli::DecompressionReaderBuilder>,
    matcher: PatternMatcher,
    searcher: grep::searcher::Searcher,
    printer: Printer<W>,
}
```

**Component responsibilities:**

| Component | Crate | Purpose |
|---|---|---|
| `matcher` | grep-regex / grep-pcre2 | Pattern matching |
| `searcher` | grep-searcher | File I/O, line iteration |
| `printer` | grep-printer | Output formatting |
| `command_builder` | grep-cli | Running preprocessors |
| `decomp_builder` | grep-cli | Running decompressors |

## Section 8: The Search Method

```rust
impl<W: WriteColor> SearchWorker<W> {
    /// Execute a search over the given haystack.
    pub(crate) fn search(
        &mut self,
        haystack: &crate::haystack::Haystack,
    ) -> io::Result<SearchResult> {
        // Select binary detection based on explicit vs implicit
        let bin = if haystack.is_explicit() {
            self.config.binary_explicit.clone()
        } else {
            self.config.binary_implicit.clone()
        };
        let path = haystack.path();
        log::trace!("{}: binary detection: {:?}", path.display(), bin);

        self.searcher.set_binary_detection(bin);

        // Decision tree: route to appropriate handler
        if haystack.is_stdin() {
            self.search_reader(path, &mut io::stdin().lock())
        } else if self.should_preprocess(path) {
            self.search_preprocessor(path)
        } else if self.should_decompress(path) {
            self.search_decompress(path)
        } else {
            self.search_path(path)
        }
    }

    /// Return a mutable reference to the underlying printer.
    pub(crate) fn printer(&mut self) -> &mut Printer<W> {
        &mut self.printer
    }
}
```

**Routing priority:** 1. stdin → search_reader (can't preprocess or mmap) 2. preprocessor match → search_preprocessor (user override) 3. decompression match → search_decompress (transparent) 4. otherwise → search_path (fast path with mmap)

## Section 9: Helper Predicates

```
impl<W: WriteColor> SearchWorker<W> {
    /// Returns true if and only if the given file path should be
    /// decompressed before searching.
    fn should_decompress(&self, path: &Path) -> bool {
        self.decomp_builder.as_ref().is_some_and(|decomp_builder| {
            decomp_builder.get_matcher().has_command(path)
        })
    }

    /// Returns true if and only if the given file path should be run through
    /// the preprocessor.
    fn should_preprocess(&self, path: &Path) -> bool {
        // No preprocessor configured
        if !self.config.preprocessor.is_some() {
            return false;
        }
        // No globs = preprocess everything
        if self.config.preprocessor_globs.is_empty() {
            return true;
        }
        // Check if path matches globs (negation-aware)
        !self.config.preprocessor_globs.matched(path, false).is_ignore()
    }
}
```

**The is_some_and pattern (Rust 1.70+):**

```
// Old way
option.map(|x| predicate(x)).unwrap_or(false)

// New way
option.is_some_and(|x| predicate(x))
```

## Section 10: Preprocessor Search

```rust
/// Search the given file path by first asking the preprocessor for the
/// data to search instead of opening the path directly.
fn search_preprocessor(
    &mut self,
    path: &Path,
) -> io::Result<SearchResult> {
    use std::{fs::File, process::Stdio};

    let bin = self.config.preprocessor.as_ref().unwrap();
    let mut cmd = std::process::Command::new(bin);
    cmd.arg(path).stdin(Stdio::from(File::open(path)?));

    let mut rdr = self.command_builder.build(&mut cmd).map_err(|err| {
        io::Error::new(
            io::ErrorKind::Other,
            format!("preprocessor command could not start: '{cmd:?}': {err}"),
        )
    })?;

    let result = self.search_reader(path, &mut rdr).map_err(|err| {
        io::Error::new(
            io::ErrorKind::Other,
            format!("preprocessor command failed: '{cmd:?}': {err}"),
        )
    });

    // Critical: wait for child process
    let close_result = rdr.close();
    let search_result = result?;
    close_result?;
    Ok(search_result)
}
```

**Error handling pattern:**

```rust
// Search might fail
let result = self.search_reader(path, &mut rdr);
// Close might fail (check after search completes)
let close_result = rdr.close();
// Return first error encountered
let search_result = result?;
close_result?;
Ok(search_result)
```

## Section 11: Decompression Search

```
/// Attempt to decompress the data at the given file path and search the
/// result.
fn search_decompress(&mut self, path: &Path) -> io::Result<SearchResult> {
    let Some(ref decomp_builder) = self.decomp_builder else {
        return self.search_path(path);
    };

    let mut rdr = decomp_builder.build(path)?;
    let result = self.search_reader(path, &mut rdr);
    let close_result = rdr.close();
    let search_result = result?;
    close_result?;
    Ok(search_result)
}
```

**Supported formats (via grep-cli):** - gzip (.gz) - bzip2 (.bz2)
- xz (.xz) - lz4 (.lz4) - lzma (.lzma) - zstd (.zst)

## Section 12: Direct File Search

```
/// Search the contents of the given file path.
fn search_path(&mut self, path: &Path) -> io::Result<SearchResult> {
    use self::PatternMatcher::*;

    let (searcher, printer) = (&mut self.searcher, &mut self.printer);
    match self.matcher {
        RustRegex(ref m) => search_path(m, searcher, printer, path),
        #[cfg(feature = "pcre2")]
        PCRE2(ref m) => search_path(m, searcher, printer, path),
    }
}
```

**Why search_path is the fast path:** - Memory mapping possible (avoids copying to userspace) - No child process overhead - No decompression overhead - Direct syscall-level optimizations available

## Section 13: Reader-Based Search

```rust
/// Executes a search on the given reader, which may or may not correspond
/// directly to the contents of the given file path.
///
/// Generally speaking, this method should only be used when there is no
/// other choice. Searching via `search_path` provides more opportunities
/// for optimizations (such as memory maps).
fn search_reader<R: io::Read>(
    &mut self,
    path: &Path,
    rdr: &mut R,
) -> io::Result<SearchResult> {
    use self::PatternMatcher::*;

    let (searcher, printer) = (&mut self.searcher, &mut self.printer);
    match self.matcher {
        RustRegex(ref m) => search_reader(m, searcher, printer, path, rdr),
        #[cfg(feature = "pcre2")]
        PCRE2(ref m) => search_reader(m, searcher, printer, path, rdr),
    }
}
```

**When search_reader is used:** - stdin (can't be mmapped) - Preprocessor output (pipe, not file) - Decompressor output (pipe, not file)

---

## Section 14: The Free Functions

```
/// Search the contents of the given file path using the given matcher,
/// searcher and printer.
fn search_path<M: Matcher, W: WriteColor>(
    matcher: M,
    searcher: &mut grep::searcher::Searcher,
    printer: &mut Printer<W>,
    path: &Path,
) -> io::Result<SearchResult> {
    match *printer {
        Printer::Standard(ref mut p) => {
            let mut sink = p.sink_with_path(&matcher, path);
            searcher.search_path(&matcher, path, &mut sink)?;
            Ok(SearchResult {
                has_match: sink.has_match(),
                stats: sink.stats().map(|s| s.clone()),
            })
        }
        Printer::Summary(ref mut p) => {
            let mut sink = p.sink_with_path(&matcher, path);
            searcher.search_path(&matcher, path, &mut sink)?;
            Ok(SearchResult {
                has_match: sink.has_match(),
                stats: sink.stats().map(|s| s.clone()),
            })
        }
        Printer::JSON(ref mut p) => {
            let mut sink = p.sink_with_path(&matcher, path);
            searcher.search_path(&matcher, path, &mut sink)?;
            Ok(SearchResult {
                has_match: sink.has_match(),
                stats: Some(sink.stats().clone()),
            })
        }
    }
}
```

**The Sink pattern:**

```
// Printer creates a sink that knows how to handle matches
let mut sink = printer.sink_with_path(&matcher, path);

// Searcher produces match events, sink handles them
searcher.search_path(&matcher, path, &mut sink)?;

// Sink accumulates results
sink.has_match()   // Did anything match?
sink.stats()       // Aggregate statistics
```

## Quick Reference: Key Types

```
// From this module
struct SearchWorkerBuilder { config, command_builder }
struct SearchWorker<W> { config, matcher, searcher, printer, ... }
struct SearchResult { has_match, stats }
enum PatternMatcher { RustRegex(...), PCRE2(...) }
enum Printer<W> { Standard(...), Summary(...), JSON(...) }

// From grep-searcher
struct Searcher;  // File reading + line iteration
enum BinaryDetection { None, Quit, Convert }

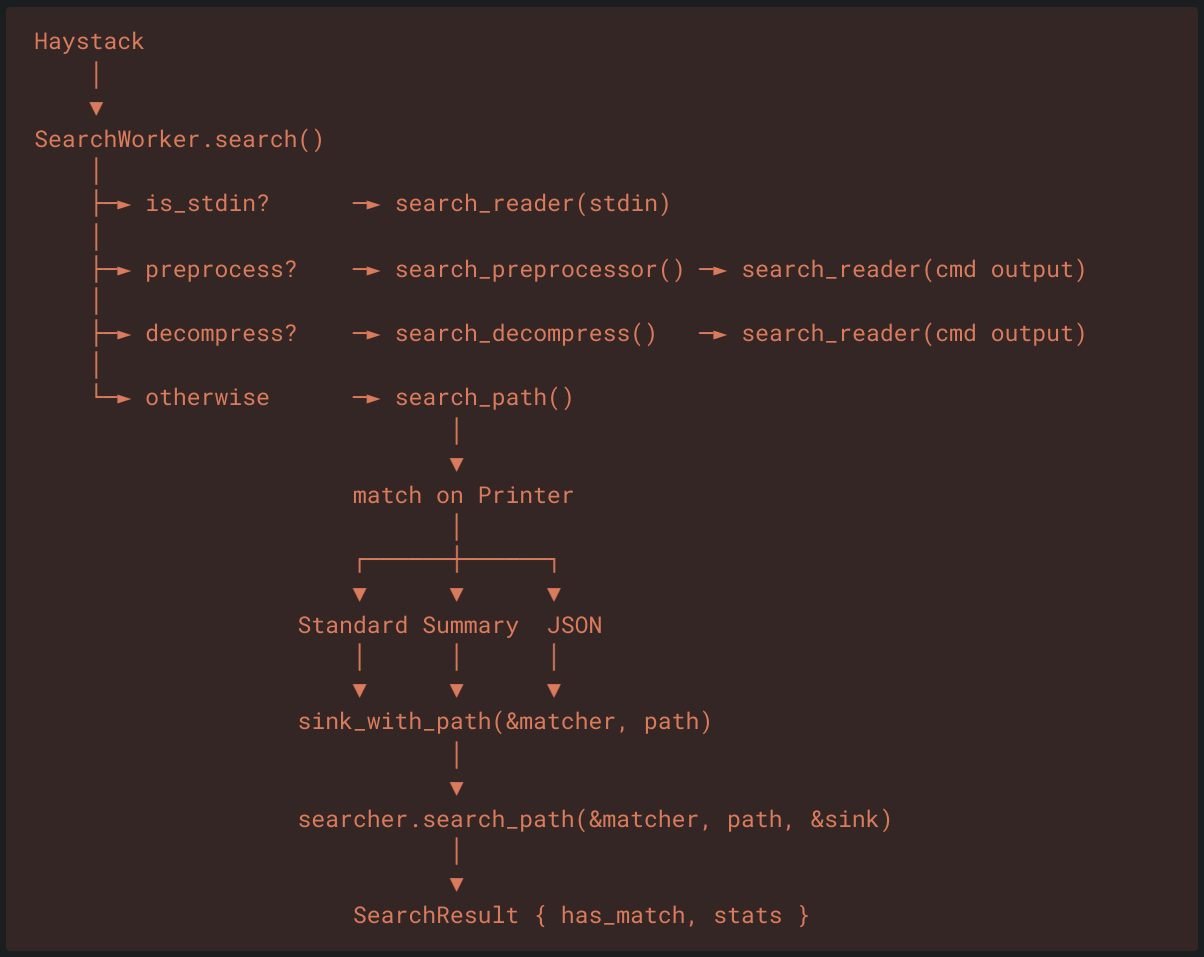
// From grep-printer
struct Standard<W>;  // Line-by-line printer
struct Summary<W>;   // Aggregate printer
struct JSON<W>;      // Structured printer
struct Stats;        // Match statistics

// From grep-matcher (trait)
trait Matcher { fn find_iter(...) }

// From grep-cli
struct CommandReaderBuilder;       // Runs external commands
struct DecompressionReaderBuilder; // Runs decompressors
```

---

## Data Flow Summary


Haystack
SearchWorker.search()
is_stdin?
search_reader(stdin)
preprocess?
search_preprocessor()
search_reader(cmd output)
decompress?
search_decompress()
search_reader(cmd output)
otherwise
search_path()
match on Printer
Standard
Summary
JSON
sink_with_path(&matcher, path)
searcher.search_path(&matcher, path, &sink)
SearchResult { has_match, stats }

# Ripgrep search.rs: The Search Coordinator

## What This File Does

The search.rs file defines the SearchWorker, which coordinates the three core components of a search operation: the matcher (regex engine), the searcher (file I/O strategy), and the printer (output formatting). It's surprisingly compact at around 450 lines because it delegates the actual work to library crates. Its job is routing and coordination.

The module doc comment captures this perfectly: "A search worker manages the high level interaction points between the matcher, the searcher, and the printer." This is the glue that connects configuration to execution.

---

## Section 1: The Config Struct

The Config struct holds five fields that control search behavior at a high level. The preprocessor field optionally holds a path to an external command that transforms file contents before searching. The preprocessor_globs field determines which files should run through the preprocessor. The search_zip boolean enables automatic decompression of compressed files.

The two binary detection fields are particularly interesting. Binary_implicit controls detection for files discovered during directory traversal. Binary_explicit controls detection for files explicitly named by the user. This distinction matters because explicitly requested files should never be silently skipped, even if they appear binary.

Default values disable all these features. No preprocessor, no decompression, no binary detection. Each must be explicitly enabled through the builder.

See: Companion Code Section 1

---

## Section 2: The Builder Pattern

SearchWorkerBuilder follows Rust's builder pattern. It starts with default configuration and provides methods to modify each setting before final construction.

The build method takes the three core components as arguments: a PatternMatcher, a Searcher, and a Printer. These come from HiArgs, which already configured them based on command-line flags. The builder adds the coordination layer.

One subtle detail: the DecompressionReaderBuilder is only constructed when search_zip is enabled. This lazy initialization avoids unnecessary work, particularly on Windows where resolving decompression binary paths can be expensive.

The async_stderr(true) calls on command builders are worth noting. When running external commands like preprocessors or decompressors, their stderr output gets forwarded asynchronously. This prevents a hung child process from blocking the search.

See: Companion Code Section 2

---

## Section 3: The Builder Methods

Each builder method configures one aspect of search behavior.

The preprocessor method validates the binary path before storing it. The resolve_binary function handles platform differences in executable lookup. This validation happens at configuration time, not search time, so errors surface early.

The preprocessor_globs method sets which files should be preprocessed. When empty (the default), all files go through the preprocessor. When populated, only matching files get preprocessed. The Override type comes from the ignore crate and supports gitignore-style patterns.

Binary detection methods demonstrate the explicit versus implicit distinction. Implicit detection typically uses "quit" mode to skip binary files entirely. Explicit detection uses "convert" mode to replace NUL bytes with something displayable, or "none" to show raw bytes.

See: Companion Code Section 3

---

## Section 4: SearchResult

The SearchResult struct captures the outcome of a single search operation. It's minimal: just whether matches were found and optional statistics.

The has_match field drives exit code determination. A search with matches returns exit code zero. A search without matches returns exit code one.

Statistics are optional because they're expensive to compute. Only when explicitly requested through flags like --stats does the printer actually collect them. The stats field contains aggregate information like match counts, line counts, and timing data.

Both fields are accessed through methods rather than direct field access. This encapsulation lets the implementation change without affecting callers.

See: Companion Code Section 4

---

## Section 5: PatternMatcher Enum

The PatternMatcher enum abstracts over regex engine implementations. Currently two variants exist: RustRegex wrapping Rust's regex crate, and PCRE2 wrapping the PCRE2 library.

PCRE2 is conditionally compiled via the pcre2 feature flag. When disabled, only RustRegex is available. This keeps binary size small for users who don't need PCRE2's advanced features.

The enum enables runtime polymorphism without trait objects. Each search method matches on the enum and calls the appropriate implementation. This approach has two advantages over dynamic dispatch: the compiler can inline through the match, and there's no vtable indirection at each call.

The Clone and Debug derives work because both inner types implement those traits. Cloning a PatternMatcher is cheap because the compiled regex is internally reference-counted via Arc.

See: Companion Code Section 5

---

## Section 6: Printer Enum

The Printer enum abstracts over output formats. Three variants cover the major use cases: Standard for grep-like line output, Summary for aggregate counts and file lists, and JSON for structured machine-readable output.

The type parameter W represents the underlying writer. In practice, this is either stdout or a BufferWriter for parallel output. The WriteColor bound ensures color escape sequences can be written.

The get_mut method provides mutable access to the underlying writer. This is essential for parallel search, where each thread writes to a private buffer that gets cleared between files.

Each printer variant wraps a type from the grep-printer crate. The Standard printer handles line-by-line output with context. The Summary printer handles counts and file-only modes. The JSON printer emits structured data conforming to a documented schema.

See: Companion Code Section 6

---

## Section 7: The SearchWorker Struct

SearchWorker brings together everything needed to execute searches. It holds the configuration, command builders, the optional decompression builder, and the three core components: matcher, searcher, and printer.

The struct is generic over W, the writer type. This allows the same SearchWorker to write to stdout, files, or memory buffers.

The Clone derive is significant. In parallel search, each thread needs its own SearchWorker. Cloning creates independent instances that can operate concurrently. The expensive parts (compiled regex) are internally shared via Arc, so cloning is cheap.

The Debug derive aids troubleshooting. When something goes wrong, being able to print the SearchWorker's state helps identify misconfiguration.

See: Companion Code Section 7

---

## Section 8: The Search Method

The search method is the entry point for searching a single haystack. It implements a decision tree that routes to the appropriate search strategy.

First, it selects binary detection based on whether the haystack is explicit or implicit. An explicit file was directly named by the user. An implicit file was discovered during directory traversal. This distinction ensures user-requested files are never silently skipped.

Then comes the routing logic. If the haystack is stdin, search_reader handles it directly. If the file should be preprocessed (based on preprocessor configuration and glob matching),

search_preprocessor transforms it first. If the file should be decompressed, search_decompress handles that. Otherwise, search_path reads the file directly.

The log::trace call provides debugging visibility. With the appropriate log level, you can see exactly which binary detection mode applies to each file.

See: Companion Code Section 8

---

## Section 9: Helper Predicates

Two helper methods determine whether special processing applies to a given path.

The should_decompress method checks whether decompression is enabled and whether the path matches a known compressed format. The DecompressionReaderBuilder maintains a mapping from file extensions to decompression commands. If no command matches, the file gets searched directly.

The should_preprocess method has three-way logic. If no preprocessor is configured, return false. If preprocessor globs are empty, preprocess everything. If globs are specified, only preprocess files that match. The is_ignore() check handles negated globs properly.

These predicates are checked in order during search routing. Preprocessing takes priority over decompression because the user explicitly requested it.

See: Companion Code Section 9

---

## Section 10: Preprocessor Search

The search_preprocessor method runs an external command and searches its output. This enables searching files that ripgrep can't read directly, like PDFs or proprietary formats.

The method constructs a Command with the file path as an argument. The file itself becomes stdin for the command. This two-path approach lets preprocessors either read the file directly or process stdin.

Error handling wraps the underlying errors with context about which command failed. This makes debugging much easier when a preprocessor misbehaves.

The close() call after searching is critical. It waits for the child process to exit and returns any errors. Without this, zombie processes could accumulate.

See: Companion Code Section 10

---

## Section 11: Decompression Search

The search_decompress method handles compressed files transparently. It recognizes formats like gzip, bzip2, xz, and others based on file extension.

The implementation mirrors search_preprocessor but uses the DecompressionReaderBuilder instead of a user-specified command. The builder knows which decompression command to use for each format.

If the decomp_builder is None (meaning search_zip is disabled), this method falls back to search_path. This shouldn't happen in practice because should_decompress would have returned false, but the defensive coding ensures correctness.

See: Companion Code Section 11

---

## Section 12: Direct File Search

The search_path method searches a file by path, allowing the searcher to use optimizations like memory mapping. This is the fast path for most searches.

The method matches on the PatternMatcher enum and delegates to a free function. This pattern avoids code duplication between RustRegex and PCRE2 variants while letting each use its own optimized implementation.

The conditional compilation attribute on PCRE2 ensures it only appears when that feature is enabled. Without this, the code wouldn't compile when PCRE2 is disabled.

See: Companion Code Section 12

---

## Section 13: Reader-Based Search

The search_reader method searches arbitrary Read implementations. It's used for stdin, preprocessor output, and decompression output.

The doc comment explains why this is less preferred than search_path: "Searching via search_path provides more opportunities for optimizations (such as memory maps)." Memory-mapped search avoids copying file contents into userspace, which can significantly speed up large file searches.

The method has the same enum matching structure as search_path. Both delegate to free functions that handle the actual search mechanics.

See: Companion Code Section 13

---

## Section 14: The Free Functions

Two free functions, search_path and search_reader, contain the actual search logic. They're generic over the Matcher trait, allowing any regex engine implementation.

The functions match on the Printer enum to create an appropriate sink. A sink is an object that receives match events and handles them appropriately. Each printer type produces its own sink type.

The sink_with_path method creates a sink that knows which file is being searched. This path appears in output and statistics.

After searching completes, the function extracts results from the sink: whether any matches occurred and optional statistics. The clone on stats is necessary because the sink owns the stats and we need to return them.

See: Companion Code Section 14

---

## Key Takeaways

First, SearchWorker is pure coordination. It holds components created elsewhere and routes searches to the right handler.

Second, the explicit versus implicit distinction pervades the design. User-requested files get different treatment than discovered files.

Third, external commands integrate cleanly. Preprocessors and decompressors are just readers that wrap child processes.

Fourth, enum-based polymorphism avoids dynamic dispatch costs. Matching on variants can be inlined and optimized.

Fifth, the sink pattern decouples searching from printing. The searcher produces match events; the sink decides what to do with them.

---

## What to Read Next

Understanding search.rs raises questions about the components it coordinates:

How does the grep-searcher crate actually find matches in files? Read the searcher implementation.

How does the Sink trait work? Read grep-printer's sink documentation.

How does memory mapping interact with the searcher? Read grep-searcher's mmap handling.

What does the DecompressionReaderBuilder support? Read grep-cli's decompression module.

# Rust Memory Safety Examples: A Comprehensive Course

## Understanding Memory Safety Through Rust's Type System

### Welcome to the Course

Memory safety vulnerabilities have been responsible for approximately 70% of security bugs in systems software over the past decade. Buffer overflows, use-after-free errors, and data races have plagued C and C++ codebases since their inception, leading to countless security exploits and system crashes. Rust was designed specifically to eliminate these classes of bugs at compile time, and this course will show you exactly how it accomplishes this remarkable feat.

The `rust-memory-safety-examples` codebase is not a typical application—it's a teaching laboratory. Every module, every function, and every type definition exists to demonstrate a specific memory safety principle. By studying this codebase systematically, you'll develop an intuition for Rust's ownership model that goes far beyond memorizing rules. You'll understand *why* the borrow checker rejects certain patterns and *how* to design your own code to work harmoniously with Rust's safety guarantees.

This course takes you from your first encounter with Rust's bounds checking through advanced concurrent programming patterns. Along the way, you'll see how Rust's seemingly strict rules actually provide tremendous flexibility once you understand their underlying logic.

### Prerequisites

Before beginning this course, you should have:

**Programming Fundamentals** - Comfort with at least one systems programming language (C, C++, Go, or similar) - Understanding of basic memory concepts: stack vs. heap, pointers, allocation and deallocation - Familiarity with common data structures: arrays, vectors, hash maps

**Basic Rust Syntax** - Ability to read Rust code (variables, functions, structs, enums) - Understanding of Rust's basic types (`i32`, `String`, `Vec<T>`, `Option<T>`, `Result<T, E>`) - Experience running Rust programs with `cargo run` and `cargo test`

**Helpful but Not Required** - Prior exposure to concepts like RAII (Resource Acquisition Is Initialization) - Basic understanding of threading and concurrency challenges - Experience debugging memory-related bugs in other languages

If you're completely new to Rust, we recommend completing the first half of "The Rust Programming Language" book before starting this course. However, if you're an experienced programmer comfortable reading new syntax, you can likely follow along while referencing documentation as needed.

---

## Learning Objectives

By the end of this course, you will be able to:

### Conceptual Understanding

- **Explain** why buffer overflows, use-after-free bugs, and data races cannot occur in safe Rust
- **Describe** the ownership model and how it differs from garbage collection and manual memory management
- **Articulate** the relationship between lifetimes, references, and memory safety
- **Compare** Rust's approach to thread safety with traditional locking strategies

### Practical Skills

- **Identify** common memory safety patterns in Rust code and understand their purpose
- **Design** data structures that leverage the type system to prevent misuse
- **Write** concurrent code that is guaranteed free of data races
- **Debug** borrow checker errors by understanding what the compiler is protecting against

### Architectural Thinking

- **Evaluate** trade-offs between different safe abstraction patterns
- **Choose** appropriate synchronization primitives for different concurrent scenarios

- **Structure** APIs that guide users toward safe usage patterns

---

## How to Use This Course

This course consists of paired documents for each lesson:

### Lecture Documents

Each lecture document (like this one) provides conceptual context and explanation. These documents are designed to be read sequentially or listened to as audio. They explain *why* code is structured certain ways, connect patterns to broader Rust principles, and highlight design decisions worth understanding.

Lecture documents reference specific code sections but don't reproduce entire files. They're meant to build your mental model before you examine the implementation details.

### Code Companion Documents

Each lesson's code companion contains annotated source code with detailed inline explanations. These documents show you *exactly* what the code does, with commentary on interesting lines and patterns. Use these as references while reading the lectures or as standalone study materials.

### Suggested Approach

**First Pass: Orientation** 1. Read the lecture document to understand the concepts 2. Run any associated examples to see the behavior 3. Skim the code companion to see how concepts map to implementation

**Second Pass: Deep Dive** 1. Read the code companion carefully, typing out key sections yourself 2. Experiment with modifications—what happens if you change something? 3. Return to the lecture document to reinforce conceptual understanding

**Third Pass: Integration** 1. Try to explain the patterns to someone else (or write your own summary) 2. Look for these patterns in other Rust codebases 3. Apply the patterns in your own projects

### Pacing Recommendations

- **Intensive Track** (2-3 days): One lesson every 2-3 hours with breaks

- **Standard Track** (1 week): One lesson per day with time for experimentation
- **Deep Learning Track** (2-3 weeks): One lesson every 2-3 days with extensive hands-on practice

---

## Learning Threads Overview

Several conceptual threads weave through this curriculum, appearing in multiple lessons with increasing depth:

### Thread 1: The Bounds Checking Journey

*Appears in: Lessons 1, 2, 3*

This thread follows Rust's approach to preventing buffer overflows. You'll start by running code that demonstrates bounds checking in action, then explore the module architecture that organizes these protections, and finally dive deep into the various bounds checking patterns available—from iterator-based approaches to explicit bounds checking with detailed error information.

The key insight of this thread: Rust doesn't just check bounds at runtime; it provides patterns that often eliminate the need for bounds checks entirely through iterator abstractions and slice patterns.

### Thread 2: Ownership and Lifetimes Mastery

*Appears in: Lessons 2, 4, 5*

This is perhaps the most important thread for understanding Rust's unique value proposition. You'll see how ownership rules prevent use-after-free bugs, how lifetimes encode validity requirements in the type system, and how these concepts enable zero-cost abstractions that are both safe and fast.

The key insight of this thread: Lifetimes are not about how long something lives—they're about proving to the compiler that references are valid for as long as they're used.

### Thread 3: Fearless Concurrency

*Appears in: Lessons 2, 6, 7*

Concurrent programming is notoriously difficult in most languages, but Rust's type system provides compile-time guarantees against data races. This thread builds from basic shared-state

concurrency with `Arc<Mutex<T>>` through advanced patterns using channels, atomic types, and the `Send` / `Sync` traits.

The key insight of this thread: Rust doesn't prevent all concurrency bugs (deadlocks are still possible), but it guarantees freedom from data races—a guarantee no other mainstream language provides without runtime overhead.

### Thread 4: Type-Driven Design

*Appears in: All lessons*

Running throughout the course is the philosophy of encoding invariants in the type system. Rather than relying on documentation or programmer discipline, Rust's patterns make invalid states unrepresentable. This thread shows how careful type design prevents entire categories of bugs.

The key insight of this thread: The best bug is one that cannot be written. Type-driven design moves error detection from runtime to compile time.

---

## Course Outline

### Lesson 1: First Encounter — Running Safe Code

**File:** `examples/buffer_overflow_prevention.rs`

Your journey begins with executable examples that demonstrate Rust's memory safety in action. This lesson focuses on buffer overflow prevention—a vulnerability class responsible for countless security exploits throughout computing history. You'll run code that would crash or be exploitable in C, and see how Rust handles the same scenarios safely.

*Key concepts:* Runtime bounds checking, panic behavior, safe APIs vs. unsafe operations

*Duration:* 30-45 minutes

### Lesson 2: The Big Picture — Memory Safety Taxonomy

**File:** `src/lib.rs`

Before diving deep into any single topic, this lesson provides an architectural overview of the entire codebase. You'll understand how the three main categories of memory safety (buffer overflows,

use-after-free, data races) relate to each other and why they're organized into separate modules. This lesson also introduces the documentation philosophy that makes Rust code self-describing.

*Key concepts:* Module organization, documentation as specification, the three pillars of memory safety

*Duration:* 45-60 minutes

---

## Lesson 3: Bounds Checking Deep Dive

**File:** `src/buffer_overflow_prevention.rs`

Now we examine bounds checking patterns comprehensively. You'll learn multiple approaches to preventing buffer overflows: iterator-based patterns that eliminate indexing entirely, safe indexing methods that return `Option<T>`, bounds-checked wrappers that provide detailed error information, and stack-allocated buffers with compile-time size guarantees.

*Key concepts:* Iterators vs. indexing, `Option` for fallible operations, newtype patterns for additional safety, const generics

*Duration:* 60-90 minutes

---

## Lesson 4: Ownership in Action

**File:** `examples/use_after_free_prevention.rs`

This lesson makes ownership tangible through runnable examples. You'll see move semantics in action, understand why certain patterns that seem reasonable are rejected by the compiler, and develop intuition for how Rust tracks value ownership. The examples demonstrate both what works and why alternatives wouldn't be safe.

*Key concepts:* Move semantics, borrowing rules, the relationship between ownership and memory deallocation

*Duration:* 45-60 minutes

---

## Lesson 5: Ownership and Lifetimes Mastery

**File:** `src/use_after_free_prevention.rs`

The deepest dive into Rust's most distinctive feature. This lesson covers lifetime annotations in detail: what they mean, when they're required, and how to read complex lifetime signatures. You'll understand lifetime elision rules, see patterns for structs that contain references, and learn how lifetimes enable safe APIs for borrowed data.

*Key concepts:* Lifetime annotations, lifetime elision, struct lifetimes, lifetime bounds on generics

*Duration:* 90-120 minutes

---

### Lesson 6: Concurrent Code Introduction

**File:** `examples/data_race_prevention.rs`

Your entry point to Rust's concurrency story. This lesson demonstrates the fundamental patterns for sharing data between threads: `Arc` for shared ownership across threads, `Mutex` for mutual exclusion, and how these types compose to provide safe concurrent access. You'll run examples that would have data races in other languages and see how Rust prevents them.

*Key concepts:* `Arc<T>`, `Mutex<T>`, thread spawning, shared mutable state

*Duration:* 45-60 minutes

---

### Lesson 7: Thread Safety Through Types

**File:** `src/data_race_prevention.rs`

The culmination of the concurrency thread. This lesson examines the `Send` and `Sync` traits that form the foundation of Rust's thread safety guarantees. You'll understand when types are automatically `Send` or `Sync`, how to work with types that aren't, and advanced patterns including channels for message passing, atomic types for lock-free programming, and reader-writer locks for read-heavy workloads.

*Key concepts:* `Send` and `Sync` traits, channels, atomics, `RwLock`, choosing the right synchronization primitive

*Duration:* 90-120 minutes

---

## Next Steps

After completing this course, you'll have a solid foundation in Rust's memory safety guarantees. Here are paths for continued learning:

### Deepen Your Understanding

- **The Rustonomicon**: The official guide to unsafe Rust—understanding unsafe helps you appreciate what safe Rust provides
- **Rust for Rustaceans** by Jon Gjengset: Advanced patterns and idioms for experienced Rust developers
- **"Learn Rust With Entirely Too Many Linked Lists"**: A deep dive into ownership through implementing various linked list types

### Apply Your Knowledge

- **Contribute to open source**: Find Rust projects that interest you and start with documentation or test contributions
- **Rewrite something in Rust**: Take a small tool you've written in another language and port it to Rust
- **Build a concurrent application**: Create something that exercises the patterns from Lessons 6-7

### Explore Related Topics

- **Async Rust**: The `async`/`await` story builds on ownership concepts in fascinating ways
- **Embedded Rust**: Memory safety on resource-constrained devices
- **WebAssembly with Rust**: Bring memory safety to the browser

### Stay Connected

- Join the Rust community on Discord, Reddit, or the official forum
- Follow This Week in Rust for ecosystem updates
- Attend local Rust meetups or RustConf

## Ready to Begin?

Start with **Lesson 1: First Encounter** to see Rust's memory safety in action. Run the examples, observe the behavior, and prepare to understand *how* Rust achieves what other languages cannot.

The journey from "fighting the borrow checker" to "thinking with the borrow checker" is transformative. By the end of this course, you won't just know Rust's rules—you'll understand their purpose and appreciate their elegance.

Let's begin.

# rust-memory-safety-examples examples/buffer_overflow_prevention.rs: First Encounter: Running Safe Code

## What This File Does

This file serves as your entry point into understanding how Rust prevents one of the most dangerous classes of security vulnerabilities in software: buffer overflows. Rather than containing complex logic itself, this example file orchestrates a demonstration of Rust's safety guarantees by calling into the library's `buffer_overflow` module.

Think of this file as your tour guide. It introduces each safety concept with a clear label, executes the demonstrations, and then summarizes what you've witnessed. The pattern here—a simple `main` function that sequences educational examples—is common in Rust learning codebases, and understanding how it works will prepare you for the more intricate safety mechanisms we'll explore in later lessons.

---

## Section 1: The Anatomy of a Rust Entry Point

Let's begin with the very first line of executable code: `fn main()`. In Rust, `main` is special—it's where your program begins execution, and unlike many other languages, it carries implicit safety guarantees from the moment it starts running.

Notice that this `main` function takes no arguments and returns nothing (the default unit type `()`). This is the simplest form of a Rust entry point. We could return a `Result` type for error handling, but for a demonstration program that's meant to show panics as a feature rather than a problem, the simple form works perfectly.

What's particularly important here is what you don't see: there's no memory allocation ceremony, no null pointer checks, no buffer initialization. Rust's runtime handles these concerns before `main` ever executes. When your code starts running, you're already standing on safe ground.

See: Companion Code Section 1

## Section 2: Module Imports and the Crate Ecosystem

The line `use rust_memory_safety_examples::buffer_overflow;` deserves careful attention because it demonstrates how Rust organizes code into **crates** and **modules**. A crate is Rust's unit of compilation—think of it as a package or library. The `use` statement brings the `buffer_overflow` module into scope from our crate.

This import pattern—`crate_name::module_name`—is fundamental to Rust. When you write `buffer_overflow::safe_array_access()`, you're calling a function from that module with its full path. This explicitness serves two purposes: it makes dependencies clear when reading code, and it avoids the naming collisions that plague languages with more permissive import systems.

The double-colon `::` is Rust's path separator, analogous to the dot in many other languages but used specifically for navigating through modules and their contents. You'll see this pattern constantly throughout Rust codebases.

See: Companion Code Section 2

---

## Section 3: Documentation Comments and Intent

The file begins with `//!` rather than the usual `//`. This is an **inner documentation comment**, and it's a small detail with big implications. Inner doc comments describe the item that contains them—in this case, the entire file. When you run `cargo doc` to generate documentation, this comment becomes the module's description.

This convention reflects Rust's philosophy that documentation is not an afterthought but a first-class citizen of the language. The tooling is designed to make good documentation habits easy. In production codebases, you'll find these comments containing examples, safety notes, and detailed explanations that become browsable documentation.

For our learning purposes, the comment "Buffer overflow prevention example" tells anyone opening this file exactly what they're looking at. In larger projects, these comments often span multiple paragraphs and include code examples that are actually tested by the compiler.

See: Companion Code Section 3

## Section 4: Sequential Demonstration Structure

The body of `main` follows a deliberate pattern: print a header, call a function, print the next header, call the next function. This isn't accidental—it's a teaching structure that makes the output readable and helps you connect what you see in the terminal to what's happening in the code.

Each `println!` macro call outputs formatted text to standard output. The `println!` is a macro (indicated by the `!`), not a function. Macros in Rust are expanded at compile time and can do things functions cannot, like accepting a variable number of arguments with different types. The `\n` within strings creates newlines for visual separation.

The numbered sections—"1. Safe Array Access", "2. Safe String Handling", "3. C vs Rust Comparison"—create a progression from simple to complex. You'll first see basic array bounds checking, then move to string handling (which involves heap allocation and dynamic sizing), and finally compare how the same operations would behave in C, a language without these safety guarantees.

See: Companion Code Section 4

---

## Section 5: Function Calls as Safety Boundaries

When you call `buffer_overflow::safe_array_access()`, you're crossing a module boundary. But more importantly, you're invoking code that will demonstrate Rust's core safety promise: memory access violations are impossible in safe Rust.

These function calls return nothing (they return the unit type `()`), which means they exist purely for their side effects—printing output and demonstrating behavior. This is common in example code but less common in production Rust, where you typically want functions to return values that can be tested and composed.

What happens inside these functions is what we'll explore in the next lesson. For now, understand that each call represents a contained experiment. If any of these functions were to attempt an actual buffer overflow, Rust would catch it—either at compile time (refusing to build) or at runtime (panicking safely rather than corrupting memory).

See: Companion Code Section 5

## Section 6: The Summary Output Pattern

The final block of the program prints a summary using checkmark symbols (✓) to reinforce the key lessons. This is a pedagogical choice, but it also demonstrates something subtle about Rust's string handling: those Unicode checkmarks work without any special encoding declarations or library imports.

Rust strings are UTF-8 by default, always. This isn't just a convenience—it's a safety feature. UTF-8 is self-synchronizing, meaning you can't accidentally split a string in the middle of a character and end up with garbage. The `String` type and `&str` slices are guaranteed to contain valid UTF-8, and the compiler enforces this.

Each println statement in this summary reinforces a core concept: compile-time prevention, runtime bounds checking, safe panics instead of undefined behavior, and automatic string resizing. These four points are the foundation of Rust's memory safety story.

See: Companion Code Section 6

---

## Section 7: Connecting to the Library

This example file sits in the `examples/` directory of a Cargo project. That location is meaningful—Cargo treats files in this directory specially. When you run `cargo run --example buffer_overflow_prevention`, Cargo compiles this file as a separate binary that links against your main library.

This architecture separates concerns elegantly. The library (`lib.rs` and its modules) contains the reusable code—the actual safety demonstrations. The example files are consumers of that library, showing how to use it. This pattern scales: a library might have dozens of examples, each demonstrating different features.

The connection to `lib.rs` is where the deeper implementation lives. When we call `buffer_overflow::safe_array_access()`, we're calling into code defined in the library. In the next lesson, we'll follow this thread and see how `use_after_free_prevention.rs` connects to the same library infrastructure.

See: Companion Code Section 7

## Section 8: What Running This Code Teaches

When you actually execute this program with `cargo run --example buffer_overflow_prevention`, several things happen that demonstrate Rust's safety philosophy in action. The program runs through each demonstration, showing both successful operations and intentional attempts to violate safety boundaries.

The key insight is what you won't see: undefined behavior. In C or C++, a buffer overflow might silently corrupt memory, work fine during testing, and then crash in production—or worse, become a security vulnerability. In Rust, any attempt to access memory out of bounds results in a **panic**, which is a controlled, safe termination with a clear error message.

Panics in Rust are not the same as crashes in C. A panic unwinds the stack (by default), running destructors for any values in scope, and then terminates. Memory is never in an inconsistent state. You can even catch panics in certain scenarios using `std::panic::catch_unwind`, though this is typically reserved for special cases like thread pool implementations.

See: Companion Code Section 8

---

## Key Takeaways

First, this file demonstrates the standard structure of a Rust example program: a simple `main` function that orchestrates demonstrations by calling into library code, with clear output that guides the reader through each concept.

Second, the `use` statement and module path syntax (`crate::module::function`) form the foundation of Rust's code organization, making dependencies explicit and preventing naming collisions.

Third, documentation comments with `//!` are part of Rust's philosophy that documentation belongs in the code and should be processed by tools into browsable reference material.

Fourth, Rust strings are UTF-8 by default, eliminating an entire class of encoding bugs and buffer vulnerabilities common in other systems languages.

Fifth, the example directory structure in Cargo projects creates a clean separation between library code and demonstration programs, a pattern you'll see throughout the Rust ecosystem.

Sixth, panics in Rust are safe—they provide controlled termination with clear diagnostics rather than the undefined behavior that plagues memory-unsafe languages.

## What to Read Next

How does the library actually implement these safety demonstrations? Read `lib.rs` to see how the `buffer_overflow` module is exposed and how Rust's module system connects this example to its implementation.

What other memory safety guarantees does Rust provide? Read `use_after_free_prevention.rs` to see how Rust's **ownership** system prevents another devastating class of bugs: accessing memory after it has been freed.

How do these examples compare to real vulnerabilities? The `buffer_overflow` module itself contains the actual demonstrations—that's where we'll go in the next lesson to see Rust's safety guarantees in action.

# rust-memory-safety-examples src/lib.rs: The Big Picture: Memory Safety Taxonomy

## What This File Does

This file serves as the architectural heart of the rust-memory-safety-examples crate, establishing a comprehensive taxonomy of memory safety vulnerabilities and their Rust-based prevention strategies. Rather than implementing complex algorithms, it functions as an educational catalog—organizing eight distinct categories of memory safety issues that plague C and C++ systems, and demonstrating how Rust's ownership model, type system, and language guarantees eliminate each one at compile time.

The structure here is deliberate: each vulnerability category exists as its own inline module, making the library both a teaching resource and a reference implementation. The file also declares three external modules (`buffer_overflow_prevention`, `data_race_prevention`, `use_after_free_prevention`) that provide deeper, runnable examples. This two-tier organization lets learners start with conceptual overviews here, then dive into hands-on examples in the companion files.

---

## Section 1: The Crate-Level Documentation Philosophy

The file opens with extensive documentation comments using Rust's `//!` syntax—the inner doc comment form that describes the enclosing item rather than the following one. This distinction matters: regular `///` comments document what comes next, while `//!` comments document the module or crate you're currently inside. Here, they establish the library's purpose before any code appears.

Notice how the documentation explicitly mentions "2024 CISA/FBI guidance" and "70% of security vulnerabilities." This isn't just marketing copy—it grounds the educational examples in real regulatory context. Financial systems, critical infrastructure, and government software increasingly mandate memory-safe languages. Understanding why Rust prevents these vulnerabilities isn't academic; it's becoming a compliance requirement.

The `#![allow(...)]` attributes that follow serve a pedagogical purpose. Normally, Clippy would warn about patterns like `vec_init_then_push` or unused code. In a production crate, you'd want

those warnings. But in a teaching library, you sometimes need to show suboptimal patterns to illustrate a point, or leave functions uncalled because they exist purely as documentation. The allows make the examples compile cleanly without cluttering the learning experience with warnings about intentionally simplified code.

See: Companion Code Section 1

---

## Section 2: Module Organization and the Prevention Taxonomy

The file declares both public inline modules and external file-based modules. The three `pub mod` statements at the top—`buffer_overflow_prevention`, `data_race_prevention`, and `use_after_free_prevention`—tell Rust to look for corresponding `.rs` files in the same directory. These modules contain the runnable examples you explored in Lesson 1.

The inline modules that follow use a different pattern: they're defined directly within `lib.rs` using the `pub mod name { ... }` syntax. This approach works well for smaller, self-contained examples that don't warrant their own files. The taxonomy covers eight vulnerability classes: buffer overflow, use-after-free, data races, integer overflow, null pointer dereference, double-free, uninitialized memory, and type confusion. Each represents a distinct failure mode in memory-unsafe languages.

This organizational split—external modules for deep dives, inline modules for quick references—reflects a common Rust pattern. Small utilities stay inline for convenience; substantial implementations get their own files for maintainability. The choice affects nothing about the compiled code, only developer experience.

See: Companion Code Section 2

---

## Section 3: Buffer Overflow Prevention Through Bounds Checking

The `buffer_overflow` module demonstrates Rust's first line of defense: automatic bounds checking on all collection accesses. In C, accessing `array[10]` when the array has only 5 elements produces undefined behavior—the program might crash, return garbage, or worse, appear to work while corrupting adjacent memory. This is the foundation of countless security exploits.

Rust offers two access patterns with fundamentally different guarantees. The `get()` method returns an `Option<&T>`—either `Some(&value)` if the index is valid, or `None` if it's out of bounds. This forces the programmer to handle both cases explicitly. The direct indexing syntax `data[10]`

also performs bounds checking, but instead of returning an Option, it panics with a clear error message when the index is invalid.

The string handling example illustrates dynamic sizing. In C, you must pre-allocate a fixed buffer and carefully track its capacity. Overflow that buffer, and you've got a security vulnerability. Rust's `String` type grows automatically—no fixed buffer to overflow, no manual capacity tracking, no vulnerability. The comparison function makes this contrast explicit, showing how C's `strcpy` into a fixed buffer becomes Rust's safe truncation using iterators.

See: Companion Code Section 3

---

## Section 4: Use-After-Free Prevention Through Ownership

The `use_after_free` module showcases Rust's signature innovation: the ownership system. In C and C++, you can free memory and then continue using the pointer that once referred to it. The freed memory might be reallocated to something else entirely, meaning your dangling pointer now corrupts unrelated data. This class of bugs is notoriously difficult to debug and frequently exploitable.

Rust prevents this at compile time through move semantics. When you assign `let owned_data = data;`, ownership transfers completely. The original binding `data` becomes invalid—not at runtime, but according to the type checker. Any subsequent attempt to use `data` produces a compile error, not a runtime crash or silent corruption.

The borrowing example demonstrates the reference side of this guarantee. When you create a reference with `let reference = &data;`, the borrow checker tracks that reference's lifetime. You cannot drop `data` while `reference` exists because the borrow checker knows that would create a dangling reference. The compile error isn't a helpful suggestion—it's an absolute prevention of the bug.

See: Companion Code Section 4

## Section 5: Data Race Prevention Through Type-System Enforcement

The `data_race` module addresses concurrent programming, where Rust's guarantees become even more valuable. A data race occurs when two threads access the same memory, at least one

access is a write, and there's no synchronization. In C++, this is undefined behavior—literally anything can happen. In Rust, it cannot compile.

The safe concurrent access example uses `Arc<Mutex<T>>`, which encodes thread-safety requirements in the type system. `Arc` (Atomic Reference Counted) provides thread-safe shared ownership—multiple threads can hold references to the same data. `Mutex` provides mutual exclusion—only one thread can access the inner value at a time. The combination ensures safe concurrent mutation.

The type system enforcement example hints at something deeper: the **Send** and **Sync** traits. `Rc` (Reference Counted) is the single-threaded version of `Arc`. It's faster because it doesn't use atomic operations, but it's not safe to share between threads. Rust encodes this constraint in the type system—`Rc` doesn't implement `Send`, so the compiler rejects any attempt to send it to another thread. You don't need to remember which types are thread-safe; the compiler knows and enforces it.

See: Companion Code Section 5

---

## Section 6: Integer Overflow Protection Through Explicit Arithmetic

The `integer_overflow` module addresses a subtler class of bugs. In C, signed integer overflow is undefined behavior, and unsigned overflow silently wraps. Both can cause security vulnerabilities—imagine an allocation size that overflows to a small value, followed by a write that assumes the original large size.

Rust takes a different approach depending on build mode. In debug builds, integer overflow panics immediately with a clear error. In release builds, it wraps (for performance), but the language provides explicit methods when you need different behavior. The `checked_add` method returns `Option<T>`—`None` on overflow, forcing you to handle the case explicitly.

The `saturating_sub` example shows another approach: saturating arithmetic. Instead of wrapping or panicking, operations clamp at the type's minimum or maximum value. For financial calculations where a balance can't go negative, `saturating_sub` ensures you never accidentally create a massive positive value through underflow. Rust gives you the vocabulary to express exactly what you intend.

See: Companion Code Section 6

---

## Section 7: Null Pointer Prevention Through Option Types

The `null_pointer` module demonstrates how Rust eliminates the "billion dollar mistake"—Tony Hoare's description of null references. In C, a null pointer and a valid pointer are the same type. You must remember to check for null before dereferencing, and the compiler won't help you if you forget.

Rust has no null pointers in safe code. Instead, optional values use `Option<T>`—an enum that's either `Some(value)` or `None`. The type system distinguishes between "definitely has a value" (`T`) and "might have a value" (`Option<T>`). You cannot accidentally treat an `Option` as if it definitely contains a value; the type checker requires you to handle both cases.

The `find_user` function returns `Option<String>`. Callers must pattern match or use one of Option's methods to extract the value. There's no way to accidentally call a method on `None`—the type doesn't have methods, only the wrapped value does. This moves null-handling from runtime discipline to compile-time enforcement.

See: Companion Code Section 7

---

## Section 8: Double-Free Prevention Through Single Ownership

The `double_free` module shows how ownership prevents another classic memory corruption bug. In C, calling `free()` twice on the same pointer corrupts the memory allocator's internal data structures. The consequences range from crashes to exploitable security vulnerabilities.

In Rust, memory is freed exactly once: when its owner goes out of scope. The `drop` function (which you can call explicitly with `drop(value)`) takes ownership of its argument. After that call, the value is moved—using it again is a compile error. You literally cannot double-free because you cannot use a value after dropping it.

The `Box` example reinforces this with heap allocation. `Box::new(42)` allocates an integer on the heap and returns a smart pointer that owns it. When `moved_box` goes out of scope, the heap memory is freed. If you tried to use `boxed_value` after moving it to `moved_box`, the compiler would reject your code. Single ownership means single deallocation—by construction, not discipline.

See: Companion Code Section 8

---

## Section 9: Uninitialized Memory Prevention Through Compiler Enforcement

The `uninitialized_memory` module addresses another category of undefined behavior in C: reading variables before they're assigned. C compilers don't require initialization, so a variable contains whatever garbage happened to be in that memory location. This can leak sensitive data or cause unpredictable behavior.

Rust's compiler performs definite initialization analysis. It tracks whether each variable is initialized on every possible code path. If there's any path where you might read a variable before assigning to it, the compiler rejects the program. You cannot forget to initialize—the build fails if you try.

The struct initialization example shows this applies to composite types too. You cannot create a struct with some fields initialized and others left undefined. Every field must have a value, or the program doesn't compile. This is stricter than C++, where you might accidentally leave members uninitialized in a constructor.

See: Companion Code Section 9

---

## Section 10: Memory Leak Prevention Through RAII

The `memory_leak` module demonstrates Resource Acquisition Is Initialization—the pattern where resources are automatically cleaned up when their owning objects go out of scope. While RAII originated in C++, Rust enforces it more consistently through the ownership system.

The file handling example shows RAII in practice. When `File::create` succeeds, it returns a `File` object that owns the file handle. When that object goes out of scope—whether normally or through an early return or panic—the file is automatically closed. You don't need to remember `fclose()`; cleanup happens by construction.

The `Drop` trait lets you customize this behavior. When you implement `Drop` for a type, Rust calls your `drop` method when values of that type go out of scope. The `DatabaseConnection` example prints a message, but real implementations would release database handles, network connections, or other system resources. Unlike C++ destructors, Rust's ownership rules ensure `drop` is called exactly once.

See: Companion Code Section 10

## Section 11: Type Confusion Prevention Through Strong Typing

The `type_confusion` module addresses bugs where data of one type is incorrectly interpreted as another. In C, casting between pointer types is common and dangerous. A function expecting a network packet might receive a string, interpreting bytes in completely wrong ways.

Rust's type system is strict about conversions. You cannot add an `i32` to an `f64` directly—the types don't match. This might seem restrictive, but it prevents subtle bugs where implicit conversions produce unexpected results. The explicit `as` cast documents your intent and makes conversions visible in code review.

The newtype pattern takes this further. By wrapping `u32` in `struct UserId(u32)` and `struct ProductId(u32)`, you create distinct types that cannot be accidentally mixed. Even though both contain a `u32`, passing a `UserId` where a `ProductId` is expected produces a type error. This pattern is zero-cost at runtime but catches logic errors at compile time.

See: Companion Code Section 11

---

## Section 12: The Test Suite as Documentation

The final section of the file contains a test module that exercises every example function. While the tests don't assert specific values—they verify that functions run without panicking—they serve an equally important purpose: demonstrating that all the example code actually works.

Each test function groups related examples together, matching the module structure. Running `cargo test` compiles and executes every example, ensuring the educational code stays in sync with current Rust. This is particularly valuable for a teaching library: students can trust that the examples they're studying actually compile and run correctly.

The `#[cfg(test)]` attribute means this module is only compiled when testing. In a library crate, tests live alongside the code they test, and the test module has access to private items. This co-location makes tests easy to maintain and signals that testing is a first-class concern, not an afterthought.

See: Companion Code Section 12

## Key Takeaways

First, Rust's memory safety guarantees form a complete taxonomy covering eight distinct vulnerability categories: buffer overflows, use-after-free, data races, integer overflow, null pointer dereference, double-free, uninitialized memory, and type confusion.

Second, the ownership system provides the foundation for most of these guarantees—single ownership prevents double-free, move semantics prevent use-after-free, and drop semantics ensure cleanup.

Third, the type system extends safety to domains like concurrency (Send/Sync traits), optional values (Option instead of null), and domain modeling (newtype pattern).

Fourth, Rust distinguishes between "cannot happen at all" (use-after-free, double-free) and "happens safely" (bounds checking panics instead of corrupting memory).

Fifth, compile-time prevention is fundamentally different from runtime detection—bugs are caught before the code ever runs, not after they've caused damage in production.

Sixth, the library's structure mirrors real Rust organization: inline modules for small examples, external modules for substantial implementations, and comprehensive tests ensuring everything actually works.

---

## What to Read Next

How does Rust prevent use-after-free in more complex scenarios? Read `use_after_free_prevention.rs` for extended examples with functions, closures, and the borrow checker's role in longer-lived references.

What happens when you need to share data between threads? Read `data_race_prevention.rs` for deeper exploration of `Arc`, `Mutex`, and the Send/Sync traits.

How do these patterns apply to realistic input handling? Read `buffer_overflow_prevention.rs` to see bounds checking in parsing scenarios that mirror production code.

# rust-memory-safety-examples src/buffer_overflow_prevention.rs: Bounds Checking Deep Dive

## What This File Does

This file demonstrates how Rust prevents buffer overflow vulnerabilities—one of the most dangerous and historically prevalent classes of security bugs in systems programming. Buffer overflows have been responsible for countless critical vulnerabilities, from the infamous Heartbleed bug to privilege escalation exploits in sudo. The file provides side-by-side comparisons of vulnerable C patterns and their safe Rust equivalents.

What makes this file particularly educational is its approach: rather than simply showing safe Rust code, it explicitly documents the dangerous C patterns that Rust's design eliminates. This context matters because buffer overflows aren't just theoretical concerns—they're the reason many organizations are migrating security-critical code to Rust. By understanding what Rust prevents and how it prevents it, you'll gain insight into the language's core design philosophy.

## Section 1: The Vulnerability Rust Eliminates

Before diving into Rust's protections, we need to understand the problem. The file opens with a documented C example showing the classic `strcpy` vulnerability. In C, the `strcpy` function copies characters from a source string to a destination buffer until it encounters a null terminator. Critically, it performs no checking whatsoever on whether the destination buffer is large enough to hold the source data.

When you copy a 46-character string into a 10-byte buffer, those extra 36 bytes don't disappear—they overwrite whatever memory comes after your buffer. On the stack, this typically means corrupting the return address, which tells the CPU where to jump when the function completes. An attacker who controls the overflowed data can redirect execution to malicious code. This isn't a theoretical concern: the CVE examples mentioned (sudo's CVE-2021-3156 and Heartbleed's CVE-2014-0160) affected millions of systems worldwide.

The fundamental issue is that C trusts the programmer completely. The language provides no mechanism to associate a buffer with its size, and standard library functions like `strcpy` have no way to know when they've written past the end. Rust takes the opposite approach: buffers carry their length information, and the compiler enforces that this information is used correctly.

See: Companion Code Section 1

---

## Section 2: Rust's Safe Copy Pattern

The `safe_copy` function demonstrates Rust's approach to the same problem. It takes a mutable slice for the destination and an immutable slice for the source. The key insight is that **slices are fat pointers**—they contain both a pointer to the data and the length of the data. This length information isn't optional metadata; it's built into the type system.

The function's first action is comparing these lengths. If the destination is smaller than the source, it returns an error rather than proceeding with a dangerous operation. This explicit check might seem obvious, but in C, you'd need to pass the buffer size as a separate parameter and remember to check it—something programmers routinely forget or get wrong.

The actual copy uses `copy_from_slice`, which performs its own bounds checking. But notice the slice syntax: `dest[..src.len()]`. This creates a sub-slice of exactly the right size, guaranteeing that the copy operation has matching source and destination lengths. The return type `Result<(), &'static str>` forces callers to handle the potential failure case—you cannot accidentally ignore a size mismatch.

This pattern illustrates a broader Rust philosophy: make the safe thing easy and the dangerous thing impossible. There's no way to call `safe_copy` and accidentally overflow the buffer, because the type system won't let you.

See: Companion Code Section 2

---

## Section 3: The Two Faces of Indexed Access

The `safe_array_access` function reveals something subtle about Rust's bounds checking strategy. Rust actually provides two different mechanisms for accessing array elements, each with different trade-offs.

Direct indexing with the bracket operator `array[0]` performs bounds checking at runtime. If the index is valid, you get the value directly. If it's out of bounds, the program panics immediately. This might seem harsh—why crash the program?—but panic is vastly preferable to the alternative. In C, out-of-bounds access is undefined behavior, which means the program might crash, might return garbage data, or might allow an attacker to read arbitrary memory. Rust's panic is deterministic and debuggable.

The safer alternative is the `get` method, which returns `Option<&T>`. When the index is valid, you get `Some(&value)`. When it's invalid, you get `None`. This forces you to handle both cases explicitly. The `match` expression in the example shows idiomatic handling: check for `Some`, do something with the value; check for `None`, handle the error gracefully.

The choice between these approaches depends on your context. If an out-of-bounds access represents a programming bug that should never happen, direct indexing with its panic behavior is appropriate—you want to crash loudly during testing. If out-of-bounds access is a normal case your program should handle (like user-provided input), use `get`.

See: Companion Code Section 3

---

## Section 4: Vectors and Dynamic Safety

The `safe_vector_usage` function demonstrates that Rust's safety guarantees extend to dynamically-sized collections. Unlike fixed-size arrays, vectors can grow and shrink at runtime. This dynamic sizing creates opportunities for bugs in other languages—what happens when you access index 5 of a vector that only has 3 elements?

In Rust, the answer is consistent with arrays: you either get a panic (with direct indexing) or a `None` (with `get`). The vector knows its current length at all times, and every access is checked against that length. There's no way to accidentally read uninitialized memory or data from a previous allocation.

The iteration pattern `for item in &vec` deserves special attention. This creates an iterator that yields references to each element in order. The iterator is guaranteed to produce exactly `vec.len()` elements—no more, no less. You cannot accidentally iterate past the end of the vector because the iterator itself enforces the bounds. This is safer and often faster than manual indexing because the bounds check happens once during iterator creation rather than on every access.

This connects to a broader Rust pattern: prefer iteration over indexing when possible. Iterators express intent more clearly, eliminate off-by-one errors, and often enable better compiler

optimizations.

See: Companion Code Section 4

---

## Section 5: Strings as Growable Buffers

The `safe_string_operations` function tackles one of C's most dangerous areas: string handling. In C, strings are just arrays of characters with a null terminator, and nearly every string operation is a potential buffer overflow waiting to happen. Concatenation, formatting, user input—all require careful manual buffer management.

Rust's `String` type is fundamentally different. It's a growable, heap-allocated buffer that manages its own memory. When you call `push_str`, the string checks whether it has enough capacity for the new data. If not, it automatically allocates a larger buffer and copies the existing content. You never need to pre-calculate how much space you'll need.

The example intentionally uses `String::with_capacity(10)` and then pushes a much longer string. In C, this would be a catastrophic overflow. In Rust, the capacity is just a hint for optimization—the string will grow as needed. The final length is whatever the content requires, safely managed by the type itself.

This automatic growth isn't magic; it has performance implications. Reallocation requires copying all existing data to a new location. If you know approximately how much data you'll store, `with_capacity` helps you avoid repeated reallocations. But the key point is that getting the capacity wrong never causes a safety problem—just potentially suboptimal performance.

See: Companion Code Section 5

---

## Section 6: Compile-Time Size Enforcement

The `demonstration_bounds_checking` function shows Rust's most powerful protection: compile-time size checking. When you have two fixed-size arrays and try to copy between them with mismatched sizes, the compiler rejects the code entirely. No runtime check needed—the program cannot be built with this bug.

The commented line `buffer.copy_from_slice(&data)` would fail to compile because `copy_from_slice` requires the source and destination slices to have identical lengths. With a 10-element destination and 20-element source, the types don't match. This is caught during compilation, before any code runs.

The safe alternative demonstrates the correct pattern: explicitly slice the source data to match the destination size with `&data[..buffer.len()]`. This makes the programmer's intent clear and creates correctly-sized slices that `copy_from_slice` accepts. The compiler can verify the safety of this operation statically.

Notice the need for `let mut mutable_buffer = buffer`. The original `buffer` binding is immutable, so we create a mutable copy. This might seem verbose, but it forces you to think about mutability explicitly. You cannot accidentally modify data you didn't intend to modify.

See: Companion Code Section 6

---

## Section 7: Testing Boundary Conditions

The test module demonstrates how to verify that safety mechanisms work correctly. The tests cover three distinct scenarios: successful copies, prevented overflows, and controlled panics.

The `test_safe_copy_success` test verifies that the function works correctly when the destination is large enough. It copies 5 bytes into a 10-byte buffer and confirms that the first 5 bytes contain the source data. This is standard unit testing, but it's important to verify the happy path works.

The `test_safe_copy_overflow_prevented` test is more interesting. It intentionally tries to copy 10 bytes into a 5-byte buffer—exactly the scenario that would cause a buffer overflow in C. The test asserts that the result is an error, verifying that the overflow was caught and prevented. The destination buffer remains untouched.

The `test_out_of_bounds_panic` test demonstrates Rust's testing infrastructure for expected panics. The `#[should_panic]` attribute tells the test framework that this test passes only if the code panics. The test accesses `array[10]` when the array only has 3 elements, which triggers a bounds check panic. This verifies that Rust's runtime safety mechanisms are working—out-of-bounds access causes controlled failure, not silent corruption.

See: Companion Code Section 7

---

## Section 8: The Safety Spectrum

Looking at this file holistically, we can see that Rust provides a spectrum of safety mechanisms, each appropriate for different situations. At one end, compile-time checking catches size mismatches before code ever runs. In the middle, runtime bounds checking catches dynamic

errors with panics. At the safer end, `Option`-returning methods let you handle bounds errors gracefully.

This spectrum reflects a design philosophy: provide the right tool for each situation rather than a one-size-fits-all solution. Sometimes you want compilation to fail. Sometimes you want a runtime panic to catch bugs early. Sometimes you want to handle out-of-bounds access as a normal case. Rust supports all three approaches.

The common thread is that undefined behavior is never an option. Whether through compiler errors, panics, or explicit error handling, Rust ensures you always know what happened and why. This predictability is crucial for security: attackers exploit undefined behavior, but they cannot exploit a program that always behaves predictably.

This connects to what you learned in the library file about Rust's memory safety taxonomy. Buffer overflows are just one category of memory safety bugs, but they share a root cause with use-after-free and data races: the program accesses memory it shouldn't. Rust's ownership system, borrowing rules, and bounds checking work together to eliminate these access violations systematically.

See: Companion Code Section 8

## Key Takeaways

First, buffer overflows occur when programs write beyond allocated memory boundaries, and they remain one of the most exploited vulnerability classes in systems software. Rust eliminates this entire category by design.

Second, Rust slices are fat pointers that carry length information, making bounds checking possible without separate size parameters. This is fundamentally different from C's approach of null-terminated strings and unchecked pointer arithmetic.

Third, Rust provides multiple access patterns with different trade-offs: direct indexing panics on bounds violations, while `get` returns `Option` for graceful error handling. Choose based on whether out-of-bounds access is a bug or a normal case.

Fourth, compile-time size checking catches many overflow bugs before code runs, but dynamic checks protect against runtime-determined indices. Both layers work together.

Fifth, `String` and `Vec` manage their own memory and grow automatically, eliminating the need for manual buffer size calculations that cause so many C vulnerabilities.

Sixth, the `#[should_panic]` test attribute lets you verify that safety mechanisms trigger correctly, turning would-be vulnerabilities into testable, controlled failures.

---

## What to Read Next

How does Rust prevent accessing memory after it's been freed? Read `use_after_free_prevention.rs` to see how ownership and borrowing eliminate another critical vulnerability class.

Want to see these bounds checking patterns in action? Return to `examples/buffer_overflow_prevention.rs` to run the demonstration code and observe the safety mechanisms yourself.

Curious how all these safety mechanisms fit together conceptually? Review `lib.rs` to see the complete taxonomy of memory safety issues that Rust addresses.

# rust-memory-safety-examples examples/use_after_free_prevention.rs: Ownership in Action

## What This File Does

This example file serves as a demonstration harness for one of memory safety's most notorious vulnerabilities: use-after-free. Where our previous lesson on bounds checking showed how Rust prevents spatial memory errors (accessing memory outside valid ranges), this file showcases Rust's prevention of temporal memory errors—accessing memory at the wrong time in its lifecycle.

The file itself is intentionally minimal, acting as a runner that invokes three demonstration functions from the library module. This design pattern separates the "what to demonstrate" from "how to demonstrate it," making the educational intent clear while keeping the executable focused on presentation rather than implementation.

See: Companion Code Section 1

---

## Section 1: The Structure of a Demonstration Runner

Notice how this example follows the same organizational pattern we saw in the buffer overflow prevention example. The `main` function serves purely as an orchestrator, calling into the library module where the actual safety demonstrations live. This isn't accidental—it's a deliberate architectural choice that mirrors how production Rust code separates binary entry points from library logic.

The `use` statement at the top imports the `use_after_free` module from our crate's library root. This is the same pattern you'd use in any Rust project: keep your logic in `src/lib.rs` and its submodules, then have thin binaries in `src/main.rs` or `examples/` that wire things together. The benefit is testability—library code can be unit tested directly, while binary code often requires integration testing.

See: Companion Code Section 2

## Section 2: Understanding Use-After-Free

Before diving into Rust's prevention mechanisms, let's understand what use-after-free actually means. In languages with manual memory management, you allocate memory, use it, and eventually free it. The vulnerability occurs when code continues to access memory after it has been freed. The memory location might now contain garbage, might have been reallocated for something else entirely, or might trigger a crash.

Use-after-free bugs are particularly insidious because they don't always manifest immediately. The freed memory might still contain valid-looking data for a while, making the bug appear to work correctly during testing but fail catastrophically in production. Security researchers prize these bugs because they can often be exploited to achieve arbitrary code execution—if an attacker can control what gets allocated in the freed space, they can potentially hijack program control flow.

The reason C and C++ struggle with this class of bug is that the type system has no concept of memory validity over time. A pointer is just an address—the compiler cannot know whether that address currently points to valid, allocated memory. Rust's insight is that preventing use-after-free requires tracking not just what type of data lives at an address, but when that data is valid.

See: Companion Code Section 3

## Section 3: Ownership as Temporal Safety

The first demonstration, `ownership_prevents_uaf()`, shows Rust's most fundamental protection: the **ownership** system. In Rust, every value has exactly one owner at any time, and when that owner goes out of scope, the value is automatically dropped (deallocated).

This might sound like a simple garbage collection scheme, but the crucial difference is determinism. Rust doesn't periodically scan for unreachable memory—it knows exactly when memory becomes unreachable because ownership transfer is explicit in the code. When you move a value to a new owner, the previous owner can no longer access it. The compiler enforces this statically.

Consider what this means for use-after-free. In C, you might free memory and still have a pointer to it—the pointer doesn't know the memory is gone. In Rust, if you transfer ownership (the equivalent of "this value will be freed by someone else"), you lose the ability to use the value. You cannot use it after it's freed because you cannot use it after you've given it away. The potential for temporal errors is eliminated by making ownership transfer visible and enforced.

See: Companion Code Section 4

---

## Section 4: Borrowing and the Lifetime Connection

The second demonstration, `borrowing_prevents_dangling()`, addresses a subtler problem. Ownership prevents use-after-free when you're transferring values, but what about when you just want to temporarily access a value owned by someone else? This is where **borrowing** and **lifetimes** enter the picture.

A reference in Rust is a borrow—you're temporarily accessing someone else's data. The compiler tracks the lifetime of every reference to ensure it never outlives the data it points to. If you try to hold onto a reference after the owner has dropped the value, the compiler will reject your code.

This is the same lifetime system we glimpsed in the bounds checking lesson, but applied to a different problem. There, lifetimes ensured slice references couldn't outlive their backing arrays. Here, lifetimes ensure any reference cannot outlive its referent. Same mechanism, different application—this is the elegance of Rust's unified approach to memory safety.

The key insight is that dangling references are just use-after-free in disguise. When you have a reference to freed memory, you have a dangling reference. Rust's lifetime system prevents both by treating them as the same underlying problem: references that outlive their referents.

See: Companion Code Section 5

---

## Section 5: The Compile-Time Advantage

Look at the print statements in `main`: they emphasize that errors are "caught before execution." This is not mere marketing language—it represents a fundamental shift in how we think about memory safety.

Traditional approaches to use-after-free fall into two categories. Runtime detection tools like AddressSanitizer can catch these bugs during testing, but they add overhead and only catch bugs in code paths that are actually exercised. Garbage collection prevents the bug entirely but imposes runtime costs and unpredictable pauses.

Rust's compile-time approach achieves the prevention of garbage collection without the runtime costs. The ownership and borrowing rules are checked during compilation, producing a binary that

simply cannot contain use-after-free bugs (in safe Rust). The final program has no additional checks, no garbage collector, no overhead—just correct code.

This is what Rust means by **zero-cost abstraction**. The safety guarantees cost nothing at runtime because they're entirely resolved at compile time. You pay for the analysis during compilation, but the resulting binary is as efficient as hand-written C that happens to be correct.

See: Companion Code Section 6

---

## Section 6: C vs Rust Mental Models

The third demonstration, `compare_c_vs_rust()`, likely shows equivalent operations in both languages to highlight the difference in guarantees. This comparison is valuable because many Rust learners come from C or C++ backgrounds and need to understand not just what Rust does differently, but why.

In C, a pointer is a promise: "I believe valid data lives at this address." The compiler trusts you. If you're wrong—if the data has been freed, if the pointer was never initialized, if you're accessing beyond allocated bounds—the compiler won't stop you. You'll get undefined behavior, which might mean a crash, might mean corrupted data, or might mean everything appears to work until it doesn't.

In Rust, a reference is a guarantee: "The compiler has verified that valid data lives at this address for as long as this reference exists." The compiler doesn't trust you—it proves your code is correct, or it refuses to compile. This is a higher bar to clear when writing code, but it means you can trust the resulting program.

The mental model shift is from "I manage memory correctly" to "the compiler verifies I manage memory correctly." This might feel like losing control, but it's actually gaining confidence. You trade the freedom to write bugs for the assurance that certain classes of bugs are impossible.

See: Companion Code Section 7

## Section 7: Connection to the Safety Taxonomy

Looking back at our lesson on `lib.rs`, we established a taxonomy of memory safety issues. Use-after-free falls into the category of **temporal** safety—accessing memory at an invalid time rather than at an invalid location. This file's demonstrations explore one branch of that taxonomy in depth.

The module path `rust_memory_safety_examples::use_after_free` mirrors this taxonomy. Just as `buffer_overflow` had its own module demonstrating spatial safety, `use_after_free` has its module for temporal safety. This organizational pattern makes the codebase navigable by concept, not just by file.

Notice how both types of safety—spatial and temporal—ultimately rely on similar mechanisms. Bounds checking uses runtime length information to validate array access. Lifetime checking uses compile-time analysis to validate reference validity. Both ensure that when you access memory, you're accessing memory that's valid for that access. The distinction is whether "valid" means "within bounds" or "still allocated."

See: Companion Code Section 8

---

## Section 8: The Educational Output Design

The print statements in this file follow a clear pedagogical structure. Each demonstration is numbered and titled, making the output easy to follow. The final "Key Takeaways" section uses checkmarks to summarize the core lessons.

This isn't just good presentation—it's good educational design. By structuring output this way, the program serves as its own documentation. A learner running this example sees not just that Rust prevents use-after-free, but why it matters and how it's achieved.

The four takeaways printed at the end align with the four key properties of Rust's approach: impossibility (ownership makes it impossible), timing (compile-time, not runtime), mechanism (lifetimes), and cost (zero-cost abstraction). Each represents a different angle on the same fundamental protection.

See: Companion Code Section 9

---

## Key Takeaways

First, use-after-free is a temporal memory safety bug—accessing memory at the wrong time in its lifecycle—and Rust prevents it through ownership and borrowing rules that are checked at compile time.

Second, ownership ensures that when a value is transferred or dropped, previous accessors can no longer reach it, making it impossible to use memory after it's been freed by the owner.

Third, borrowing with lifetimes extends this protection to references, ensuring that no reference can outlive the data it points to, preventing dangling references.

Fourth, the compile-time nature of these checks means zero runtime overhead—the resulting binary has no garbage collector, no reference counting, and no validity checks for references.

Fifth, this example runner pattern—thin binary calling into library functions—demonstrates idiomatic Rust project structure that separates executable concerns from library logic.

Sixth, the conceptual connection between spatial safety (bounds checking) and temporal safety (use-after-free prevention) shows how Rust's safety mechanisms work together as a coherent system.

---

## What to Read Next

How does the library implement these demonstrations? Read `src/use_after_free.rs` to see the actual ownership and borrowing examples with commented code showing what the compiler prevents.

What other memory safety issues does Rust address? Revisit `src/lib.rs` to review the complete taxonomy and see how use-after-free fits into the broader picture.

How do these safety guarantees work in concurrent code? The upcoming lessons will explore data race prevention, where ownership and borrowing rules extend to multi-threaded scenarios.

# rust-memory-safety-examples src/use_after_free_prevention.rs: Ownership and Lifetimes Mastery

## What This File Does

This file demonstrates how Rust's ownership system and lifetime rules prevent use-after-free vulnerabilities at compile time. Use-after-free is one of the most dangerous classes of memory safety bugs—it occurs when a program accesses memory after that memory has been freed, leading to undefined behavior, crashes, and exploitable security vulnerabilities. The C code at the top of this file illustrates exactly this pattern: allocating memory, freeing it, then returning a dangling pointer that points to memory no longer owned by the program.

What makes this file particularly valuable is that it doesn't just show safe Rust code—it shows the unsafe patterns that *would* exist if Rust didn't have its ownership system, marked as comments that deliberately won't compile. This approach lets you see both sides of the safety boundary: the dangerous pattern you might write in C, and how Rust's compiler stops you from making the same mistake.

---

## Section 1: The Vulnerability We're Preventing

Before diving into Rust's solutions, we need to understand what use-after-free actually looks like in vulnerable code. The module-level documentation presents a C function called `vulnerable_use_after_free` that commits this cardinal sin: it allocates memory, stores a value, frees the memory, and then returns a pointer to that freed memory. The caller then tries to read through that dangling pointer.

This pattern is deceptively simple to create in C. There's no syntax error, no warning in many compilers, and the code might even appear to work during testing. The freed memory might still contain the value 42, purely by accident—the memory hasn't been overwritten yet. But this is exactly what makes use-after-free bugs so insidious. They can lurk in code for years, only manifesting when the memory allocator reuses that freed block for something else.

The real-world impact is severe. The CVE examples mentioned—SMBGhost and the Chrome FileReader vulnerability—weren't theoretical concerns. They were exploitable bugs that allowed attackers to execute arbitrary code. When you access freed memory, an attacker who can control what gets allocated in that space can effectively control what your code reads, potentially redirecting program flow to malicious code.

See: Companion Code Section 1

---

## Section 2: Ownership as the First Line of Defense

The first safe pattern in the file demonstrates ownership preventing use-after-free in the most direct way possible. The function `ownership_prevents_uaf` creates a `Box<i32>`—a heap-allocated integer—then copies its value, explicitly drops the box, and finally uses the copied value.

The critical insight here is understanding what `drop(data)` actually does. In Rust, `drop` is how you explicitly free heap memory before a value would naturally go out of scope. After this call, the memory that `data` pointed to has been returned to the allocator. The box is gone. And here's where Rust's ownership system shines: the compiler *knows* this. The `data` variable is no longer valid after the `drop` call.

The commented-out line `println!("{}", *data)` isn't just a style suggestion—it's code that physically cannot compile. The compiler tracks ownership through every statement in your function. Once `data` is dropped, attempting to use it produces a compile-time error. There's no runtime check happening here, no garbage collector intervening. The invalid access simply cannot be expressed in valid Rust.

Notice that we can still use `value` after the drop. That's because we copied the integer value *before* dropping the box. Integers implement the `Copy` trait, meaning the `*data` expression creates a bitwise copy of the integer that exists independently of the box. This distinction between copying values and moving ownership is fundamental to understanding Rust's memory model.

See: Companion Code Section 2

---

## Section 3: Lifetimes Preventing Dangling References

The `lifetime_prevents_dangling_ref` function demonstrates a subtly different scenario: what happens when you try to hold a reference to data that goes out of scope. This is the reference

version of use-after-free—you're not freeing memory explicitly, but the automatic cleanup at scope boundaries has the same effect.

The pattern here involves declaring a reference variable in an outer scope, then trying to make it point to data created in an inner scope. The inner scope creates a `String`, and the commented-out code attempts to store a reference to that string in the outer variable. When the inner scope ends, the `String` is automatically dropped—its heap memory freed—but we'd still have a reference claiming to point to valid data.

Rust prevents this through **lifetime analysis**. Every reference in Rust has a lifetime—a region of code during which the reference is valid. The compiler analyzes these lifetimes and refuses to compile code where a reference would outlive its referent. In this case, the `String` has a lifetime limited to the inner block, while the reference variable has a lifetime extending beyond it. These lifetimes are incompatible, and the compiler rejects the code.

What makes this particularly elegant is that the analysis happens entirely at compile time. There's no runtime tracking of reference validity, no performance cost for this safety. The compiler simply won't produce an executable that contains this class of bug.

See: Companion Code Section 3

## Section 4: Borrowing Rules and Aliasing Control

The `borrowing_prevents_uaf` function reveals another dimension of Rust's safety guarantees: the interaction between immutable references and mutation. This pattern catches a class of bugs that's slightly more subtle than the previous examples—bugs where memory might be freed *indirectly* through mutation while references still exist.

The setup creates a vector and takes an immutable reference to its first element. The commented-out `data.clear()` call would, if allowed, deallocate the vector's backing storage. The reference would suddenly point to freed memory. This is use-after-free through collection invalidation—a common source of bugs in languages that allow arbitrary mutation.

Rust's borrowing rules prevent this through a simple but powerful principle: when an immutable reference exists, the underlying data cannot be mutated. The compiler sees that `reference` holds an immutable borrow of data inside the vector, and it therefore blocks any operation that would mutate the vector until that borrow ends.

The borrow ends when `reference` is last used—in this case, after the `println!` statement. After that point, `data.clear()` compiles and runs successfully. This is Rust's **Non-Lexical Lifetimes** in

action: borrows don't extend to the end of their scope, but only until their last use. This refinement, introduced in Rust 2018, makes the borrow checker more permissive while maintaining safety.

See: Companion Code Section 4

---

## Section 5: Reference Counting for Shared Ownership

Not all ownership patterns fit neatly into single-owner semantics. Sometimes multiple parts of your program legitimately need to share ownership of data, and you don't know at compile time which one will use it last. The `shared_ownership_safe` function demonstrates `Rc<T>`, Rust's reference-counted smart pointer for exactly this scenario.

`Rc` stands for "Reference Counted." Each call to `Rc::clone` doesn't copy the underlying data—it increments a counter and returns a new handle to the same allocation. The memory is freed only when the last `Rc` is dropped and the count reaches zero. This is a form of garbage collection, but it's deterministic and explicit rather than automatic and opaque.

The key safety property here is that you cannot get a dangling reference to data inside an `Rc`. The reference count ensures the data lives as long as any handle to it exists. The function deliberately drops each clone in sequence, and the memory is freed only after the final `drop(data)` call—when we're absolutely certain no references remain.

There's an important caveat: `Rc` is not thread-safe. The reference count isn't updated atomically, so sharing an `Rc` across threads would introduce data races. For thread-safe shared ownership, Rust provides `Arc<T>` (Atomic Reference Counted), which uses atomic operations for the counter. This separation keeps `Rc` efficient for single-threaded code while making the thread-safety requirements explicit in the type system.

See: Companion Code Section 5

---

## Section 6: The SafeObject Pattern - Consuming Self

The `SafeObject` struct and its `consume` method demonstrate a powerful pattern for managing resource lifecycles: methods that take ownership of `self`, preventing further use of the object. This pattern appears throughout Rust's ecosystem for operations that logically "finish" with an object.

The `consume` method's signature—`fn consume(self) -> Vec<i32>`—takes `self` by value, not by reference. This means calling `consume` moves ownership of the entire `SafeObject` into the

method. The method can then destructure the object and return its internal data. After this call, the `SafeObject` no longer exists as far as the type system is concerned.

The `safe_object_usage` function shows this in action. After calling `obj.consume()`, any attempt to use `obj` produces a compile error. The object has been consumed—its memory hasn't necessarily been freed yet (the `Vec` data is returned), but the `SafeObject` wrapper is gone. You cannot accidentally use the object after it's been logically invalidated.

This pattern is essential for implementing safe abstractions over resources that have cleanup requirements. Think of file handles that must be closed, network connections that must be terminated, or database transactions that must be committed. By consuming `self`, you make it impossible for callers to use the resource after the cleanup operation.

See: Companion Code Section 6

---

## Section 7: The Drop Trait and Deterministic Destruction

The `compile_time_safety_demo` function showcases the `Drop` trait, Rust's mechanism for running cleanup code when a value goes out of scope. This is RAII (Resource Acquisition Is Initialization) implemented at the language level, and it's the foundation for Rust's deterministic memory management.

The `Resource` struct implements `Drop` with a simple print statement. This lets us observe exactly when the destructor runs. When `res` goes out of scope at the end of the function, Rust automatically calls `drop`, which runs our custom cleanup code and then frees the memory.

The deterministic nature of Rust's destruction is crucial for understanding how use-after-free is prevented. Unlike garbage-collected languages where objects might be collected at any time (or never, until memory pressure), Rust values are destroyed at precisely predictable points. When a variable goes out of scope, when a value is explicitly dropped, when ownership is transferred and the new owner doesn't use the value—in all these cases, the destructor runs immediately.

This predictability is what makes the ownership model work. The compiler can reason about when values exist and when they don't because the rules are entirely deterministic. There's no uncertainty about whether a value might still be live somewhere. If the borrow checker says a value is gone, it's gone.

See: Companion Code Section 7

---

## Section 8: Testing Ownership Semantics

The test module provides executable verification of the ownership patterns discussed throughout the file. These tests don't just check that code works—they document the expected behavior of Rust's ownership system in a way that can be mechanically verified.

The `test_ownership_transfer` test demonstrates move semantics. When we write `let moved_data = data`, ownership transfers from `data` to `moved_data`. The original `data` binding becomes invalid—we can't even mention it in the assert. This test would fail to compile if we tried to use `data` after the move.

The `test_rc_shared_ownership` test is particularly illuminating because it uses `Rc::strong_count` to observe the reference count at runtime. We can see the count start at 1, increase to 2 when we clone, then decrease back to 1 when we drop the clone. This provides concrete evidence of the reference counting mechanism that keeps shared data alive.

What these tests demonstrate is that Rust's safety properties aren't just theoretical—they're testable and observable. You can write code that explores the boundaries of the ownership system and verify that it behaves as documented. This testability builds confidence that the safety guarantees hold in practice, not just in theory.

See: Companion Code Section 8

---

## Section 9: Connecting to the Safety Taxonomy

This file connects directly to the taxonomy established in `lib.rs`. Use-after-free is one of the four categories of memory safety vulnerabilities that the crate addresses, alongside buffer overflows, null pointer dereferences, and data races. Understanding how each category is prevented helps build a complete mental model of Rust's safety guarantees.

The pattern established here—showing vulnerable C code, explaining the vulnerability, then demonstrating Rust's prevention mechanism—recurs throughout the crate. Each vulnerability type has its own prevention mechanism, but they all share a common theme: making invalid states unrepresentable at compile time rather than checking for them at runtime.

What distinguishes use-after-free prevention from buffer overflow prevention is the nature of the compile-time analysis. Buffer overflow prevention relies primarily on bounds checking—runtime checks that Rust inserts automatically. Use-after-free prevention relies primarily on ownership and

lifetime analysis—compile-time checks that produce no runtime overhead. Both are forms of safety, but they operate through different mechanisms.

This distinction matters for understanding Rust's performance characteristics. The ownership system adds zero runtime cost while preventing an entire class of memory safety bugs. When you hear that Rust provides "zero-cost abstractions," the ownership system is the primary example. The abstraction of safe memory management costs nothing at runtime because all the checking happens before your program runs.

See: Companion Code Section 9

---

## Key Takeaways

1. **Use-after-free vulnerabilities in C occur when code accesses memory after it's been freed, leading to undefined behavior and exploitable security bugs that have caused real-world CVEs.**
2. **Rust's ownership system prevents use-after-free at compile time by tracking when values are dropped and making it a compile error to use them afterward—there's no runtime check, just static analysis.**
3. **Lifetimes ensure references cannot outlive their referents, preventing dangling references even when data is cleaned up automatically at scope boundaries.**
4. **The borrowing rules prevent indirect use-after-free by blocking mutation while references exist, ensuring that operations like clearing a vector can't invalidate existing references to its elements.**
5. **Reference counting with `Rc<T>` provides shared ownership for cases where single-owner semantics don't fit, with the memory freed only when all owners are gone.**
6. **Methods that consume `self` explicitly transfer ownership, making it impossible to use an object after an operation that logically invalidates it—a powerful pattern for resource lifecycle management.**

## What to Read Next

How do these ownership patterns apply when building real programs? Explore `examples/use_after_free_prevention.rs` to see these concepts used in practice with more complex scenarios.

What's the relationship between use-after-free prevention and the other safety categories? Return to `src/lib.rs` to review how ownership fits into the overall taxonomy of memory safety vulnerabilities.

How does Rust prevent the next category of memory bugs? Continue to the next lesson, which covers null pointer prevention and the `Option<T>` type that makes null references impossible.

# rust-memory-safety-examples examples/data_race_prevention.rs: Concurrent Code Introduction

## What This File Does

This example file serves as the entry point for exploring Rust's data race prevention guarantees. It demonstrates how Rust achieves something remarkable: the complete elimination of data races at compile time, a category of bug that has plagued concurrent programming for decades.

The file itself is deceptively simple—just a main function that calls three demonstrations from the library. But this simplicity is the point. Where concurrent code in other languages requires careful reasoning, defensive programming, and often runtime debugging tools to catch race conditions, Rust lets you write straightforward code with the confidence that the compiler will reject anything that could race.

## Section 1: From Memory Safety to Thread Safety

In the previous lessons, you've seen how Rust's ownership system prevents use-after-free bugs. A value has one owner, and when that owner goes out of scope, the value is dropped. You've traced through how borrowing rules prevent dangling references and how lifetimes ensure references never outlive their referents.

Now we extend that model to concurrency. The insight is profound: the same ownership rules that prevent memory bugs in single-threaded code also prevent data races in multi-threaded code. If only one owner can mutate data, and the compiler tracks that owner, then the compiler can also track when data crosses thread boundaries.

This file demonstrates that principle in action. Each function call shows a different facet of Rust's thread safety guarantees, but they all flow from the same foundation you've already learned. The transition from "no use-after-free" to "no data races" is not a separate mechanism—it's the natural extension of ownership to concurrent contexts.

See: Companion Code Section 1

## Section 2: The Main Function Structure

The main function follows a pattern you'll recognize from earlier examples: it prints a header, runs demonstrations, and ends with key takeaways. This structure serves an educational purpose—each numbered call becomes a clear checkpoint for understanding.

Notice how the code reads almost like documentation. The function names `safe_concurrent_access`, `type_system_prevents_races`, and `compare_c_vs_rust` tell you exactly what each section demonstrates. This isn't accidental. When you're learning concurrent programming concepts, having clear boundaries between ideas helps prevent the mental confusion that concurrency often induces.

The sequential execution of these demonstrations also matters. We're running concurrent code examples one after another, which means each demonstration starts fresh. There's no lingering state from previous calls that could confuse the examples.

See: Companion Code Section 2

---

## Section 3: Safe Concurrent Access with Arc and Mutex

The first demonstration introduces `Arc` and `Mutex`, the two primary tools for sharing mutable data across threads. Understanding why both are needed reveals deep truths about Rust's approach to thread safety.

`Arc` stands for "Atomically Reference Counted." It's like `Rc` from the standard library, but safe to share across threads. The "atomic" part means the reference count is updated using atomic CPU operations, preventing the count itself from becoming corrupted by concurrent access. When you need multiple threads to hold a reference to the same data, `Arc` provides shared ownership across thread boundaries.

`Mutex` provides mutual exclusion—only one thread can access the wrapped data at a time. But here's the critical insight: `Mutex` in Rust doesn't just synchronize access, it *ties the synchronization to the type system*. You literally cannot access the inner data without going through the lock. This is different from C or C++ mutexes, where the mutex is just a lock sitting next to the data, and nothing stops you from forgetting to acquire it.

The combination `Arc<Mutex<T>>` means: shared ownership (`Arc`) of exclusively-accessed data (`Mutex<T>`). This pattern is so common in Rust that it becomes second nature.

See: Companion Code Section 3

---

## Section 4: The Type System as Thread Safety Enforcer

The second demonstration shows how Rust's type system prevents races without any runtime overhead. This is where we see the `Send` and `Sync` traits in action.

`Send` is a marker trait that indicates a type is safe to transfer to another thread. Most types are `Send`, but some aren't. For example, `Rc<T>` is not `Send` because its reference count isn't atomic—if two threads tried to clone or drop an `Rc` simultaneously, they'd corrupt the count.

`Sync` indicates that references to a type are safe to share across threads. A type `T` is `Sync` if `&T` is `Send`. This captures whether multiple threads can safely read from the same data simultaneously.

The brilliance is that these traits are automatically derived. The compiler looks at a struct's fields and determines whether it's `Send` and `Sync` based on its components. If you put a non-`Send` type in your struct, your struct automatically becomes non-`Send`. Try to spawn a thread with non-`Send` data, and you get a compile error, not a race condition at runtime.

See: Companion Code Section 4

---

## Section 5: Connecting to Ownership Fundamentals

This is where the connection to use-after-free prevention becomes explicit. Recall from the previous lesson that Rust's ownership rules prevent you from using memory after it's freed by tracking which code owns which values.

The thread safety rules are essentially: "ownership, but across threads." When you spawn a thread, Rust applies the same scrutiny it applies to any ownership transfer. The data moved into the thread closure must be valid for the thread's entire lifetime. The compiler ensures no other code can access data that's been moved to another thread unless that data is wrapped in proper synchronization primitives.

Consider what happens without these rules. In C, you might pass a pointer to a local variable into a thread. The parent function returns, destroying the variable, but the thread keeps running with a dangling pointer. Rust's ownership system catches this—the borrow checker sees that you're trying to give a thread a reference to data that won't outlive the thread.

This unified model means learning ownership pays dividends twice: first for memory safety, then again for thread safety.

See: Companion Code Section 5

---

## Section 6: C vs Rust Comparison

The third demonstration makes the comparison explicit. Data races are undefined behavior in C and C++, which means they can cause arbitrary program behavior, not just incorrect results. A race might work fine in development and fail catastrophically in production. It might manifest as memory corruption, security vulnerabilities, or impossible-to-debug heisenbugs.

Rust's approach is fundamentally different. Rather than making data races undefined behavior, Rust makes them impossible to write. The compiler rejects programs that could race before they ever run. This is a stronger guarantee—not "if you're careful you won't have races" but "races cannot occur."

The demonstration typically shows how Rust's design leads to code that's both safer and often clearer about its concurrency properties. When you see `Arc<Mutex<T>>`, you know exactly what's happening: shared ownership with mutual exclusion. In C, you might see a pointer and a mutex variable nearby, but nothing in the type system connects them.

See: Companion Code Section 6

---

## Section 7: The Key Takeaways Section

The printed takeaways at the end summarize what makes Rust's approach unique. Let's examine each:

"Rust prevents data races at compile time" is the headline claim. Not at runtime with sanitizers, not with code review, but as part of compilation. A Rust program that compiles either has no data races or uses `unsafe` code (which opts out of these guarantees explicitly).

"Send and Sync traits enforce thread safety" explains the mechanism. These aren't runtime checks—they're compile-time markers that the type system uses to prevent invalid thread sharing.

"Arc and Mutex provide safe concurrent access" names the primary tools. These aren't the only synchronization primitives—there are also `RwLock`, channels, atomic types—but they're the

foundational ones.

"No undefined behavior in concurrent code" is perhaps the most important. In safe Rust, even if you make logical errors in your concurrent code, you won't corrupt memory or create security vulnerabilities.

See: Companion Code Section 7

---

## Section 8: How Examples Call Library Code

This file demonstrates a common pattern in Rust projects: example files that call into library code. The single `use` statement imports the `data_race` module from the library, and then each call invokes a function from that module.

This separation serves multiple purposes. The library contains the actual implementations and the detailed explanations of what's happening. The example file provides a runnable entry point that shows how to use the library. When you run `cargo run --example data_race_prevention`, you're compiling and running this main function.

For learning, this pattern is valuable. You can first run the example to see behavior, then dive into the library code to understand implementation. The example serves as the "what" while the library provides the "how" and "why."

See: Companion Code Section 8

---

## Key Takeaways

First, Rust's thread safety guarantees emerge from the same ownership system that prevents use-after-free bugs. Learning ownership well gives you memory safety and thread safety together.

Second, the `Send` and `Sync` traits are the mechanism behind compile-time data race prevention. They're marker traits that the compiler uses to track what can safely cross thread boundaries.

Third, `Arc<Mutex<T>>` is the fundamental pattern for sharing mutable state across threads. Unlike C-style mutexes, Rust's `Mutex` is inseparable from the data it protects—you cannot access the data without going through the lock.

Fourth, Rust converts data races from runtime undefined behavior (as in C/C++) to compile-time errors. This is a qualitatively different level of safety.

Fifth, example files like this one serve as entry points for exploration. The real depth is in the library code they call, which we'll explore in the next lesson.

---

## What to Read Next

How does `safe_concurrent_access` actually implement thread-safe sharing? Read `src/data_race.rs` to see the `Arc<Mutex<T>>` pattern in detail.

What compile errors do you get when you violate thread safety? The library code shows examples of what the compiler catches and why.

How do Send and Sync work at the type level? The next lesson on `src/data_race.rs` will show the trait system in action.

# rust-memory-safety-examples src/data_race_prevention.rs: Thread Safety Through Types

## What This File Does

This file demonstrates how Rust prevents data races—one of the most insidious bugs in concurrent programming—entirely at compile time through its type system. While the previous lesson introduced the concepts of `Send` and `Sync` traits through examples, this module dives deeper into the mechanisms that make thread safety a compile-time guarantee rather than a runtime hope.

The file presents six distinct patterns for safe concurrent programming, each addressing different use cases: mutex-protected shared state, read-write locks for read-heavy workloads, the `Send` and `Sync` marker traits, message passing channels, lock-free atomic operations, and scoped threads for borrowed data. Together, these patterns form a complete toolkit for writing concurrent Rust code that simply cannot have data races.

## Section 1: The Data Race Problem in C

Before examining Rust's solutions, it's worth understanding the problem we're solving. The file opens with a C code example that demonstrates a classic data race: two threads incrementing a shared counter without synchronization. This isn't just bad practice—it's undefined behavior according to the C standard, meaning the compiler can do literally anything.

The C example uses `pthread_create` to spawn two threads, each incrementing a global `counter` variable 100,000 times. You might expect the final value to be 200,000, but it won't be. The increment operation `counter++` isn't atomic—it's actually a read, modify, and write sequence. When two threads interleave these operations, increments get lost. Run this program multiple times and you'll get different results each time.

What makes data races particularly dangerous is their non-determinism. The bug might not manifest during testing, only to appear in production under different timing conditions. Security vulnerabilities can arise when attackers exploit these race conditions to corrupt memory or bypass

checks. The C code compiles without warning, runs without immediate failure, and produces subtly wrong results that may go unnoticed for years.

See: Companion Code Section 1

---

## Section 2: Mutex-Protected Shared State

Rust's most direct solution to the shared counter problem is the `Mutex<T>` type. Unlike C's pthread mutexes, which are separate from the data they protect, Rust's `Mutex` wraps the data it guards. This design makes it impossible to access the data without first acquiring the lock—the type system enforces what C leaves to programmer discipline.

The `safe_concurrent_counter` function demonstrates this pattern. The counter is wrapped in `Mutex::new(0)`, which is then wrapped in `Arc` (atomic reference counting) for shared ownership across threads. Each thread clones the `Arc`, not the data itself—they all share the same underlying mutex-protected counter.

Inside each thread, calling `counter_clone.lock().unwrap()` returns a `MutexGuard`, which is a smart pointer that provides access to the inner value. The guard implements `Deref` and `DerefMut`, so you can use `*num += 1` to modify the counter. When the guard goes out of scope—at the end of each loop iteration—it automatically releases the lock. This RAII pattern makes it impossible to forget to unlock, eliminating an entire class of deadlock bugs.

The result is deterministic: ten threads each incrementing 1,000 times always produces exactly 10,000. The type system has transformed a runtime data race into a compile-time impossibility.

See: Companion Code Section 2

---

## Section 3: Read-Write Locks for Concurrent Readers

Mutexes provide exclusive access, but many workloads involve far more reads than writes. A mutex forces readers to wait for each other, even though concurrent reads are perfectly safe. `RwLock<T>` addresses this by distinguishing between shared read access and exclusive write access.

The `safe_read_write_access` function creates five reader threads and one writer thread. Each reader calls `data_clone.read().unwrap()`, obtaining a `RwLockReadGuard` that allows shared access to the data. Multiple readers can hold read guards simultaneously—the lock only blocks when a writer is waiting or active.

The writer thread calls `data_clone.write().unwrap()`, obtaining a `RwLockWriteGuard` that provides exclusive mutable access. While this guard exists, no readers can acquire access, and no other writers can proceed. This matches the fundamental rule from earlier lessons: you can have many shared references or one mutable reference, but never both.

The type system encodes these semantics through the guard types. A `RwLockReadGuard` only implements `Deref`, not `DerefMut`, so you literally cannot mutate the data through a read guard. A `RwLockWriteGuard` implements both, but the lock mechanism ensures exclusivity. The reader-writer pattern that requires careful manual implementation in C becomes a type-safe abstraction in Rust.

See: Companion Code Section 3

---

## Section 4: Send and Sync — The Marker Traits

We've seen `Send` and `Sync` mentioned throughout this codebase, but this file makes their roles explicit. These are **marker traits**—they have no methods, only semantic meaning. `Send` indicates a type can be safely transferred to another thread. `Sync` indicates a type can be safely shared (via references) between threads.

The file defines `SendableData`, a simple struct containing an `i32`. Rust automatically implements `Send` for this type because `i32` is `Send`, and structs composed entirely of `Send` types are themselves `Send`. You can move `SendableData` into a `thread::spawn` closure without issue.

Contrast this with `NotSendable`, which contains a raw pointer `*mut i32`. Raw pointers are explicitly not `Send` because the compiler cannot verify what they point to or how that data might be accessed from another thread. If you uncomment the code attempting to spawn a thread with `NotSendable`, you get a compile error, not a runtime bug or a data race that appears only under load.

This is where Rust's thread safety guarantee becomes concrete. The `thread::spawn` function signature requires its closure to be `Send`. The compiler automatically determines whether types implement `Send` based on their composition. You don't write `impl Send for SendableData`—the compiler infers it. And for types like `NotSendable` where thread transfer would be unsafe, the compiler automatically withholds the trait.

See: Companion Code Section 4

---

## Section 5: Message Passing as an Alternative Model

Shared state isn't the only way to coordinate between threads. The **message passing** model, embodied in Rust's `mpsc` (multiple producer, single consumer) channels, offers a different philosophy: instead of sharing data and synchronizing access, transfer ownership of data between threads.

The `safe_message_passing` function creates a channel with `mpsc::channel()`, which returns a transmitter (`tx`) and receiver (`rx`). The transmitter is moved into a spawned thread, which sends integers 0 through 4. The main thread iterates over the receiver, which blocks waiting for messages until the channel closes.

When `tx.send(i)` executes, ownership of the value transfers from the sending thread to the channel, and then to the receiving thread when it calls `recv()`. There's no shared mutable state—each value has exactly one owner at any time. The `Send` trait ensures that only thread-safe values can be transmitted.

This pattern aligns with Rust's ownership philosophy. Rather than fighting over shared resources, threads communicate by explicitly passing values. The channel handles synchronization internally, and the type system ensures you can't accidentally access sent data from the original thread. It's a fundamentally different concurrent programming model that avoids shared-state bugs by eliminating shared state.

See: Companion Code Section 5

---

## Section 6: Lock-Free Programming with Atomics

For performance-critical code, even mutex overhead can be too much. **Atomic types** like `AtomicUsize` provide lock-free operations that the CPU executes as single, indivisible instructions. No mutex, no blocking, no deadlock possibility—but with subtle semantics that require careful reasoning.

The `safe_atomic_operations` function mirrors the mutex counter example but uses `AtomicUsize` instead. The key operation is `fetch_add(1, Ordering::SeqCst)`, which atomically increments the counter and returns the previous value. This operation cannot be interrupted or interleaved—the CPU guarantees it completes as a single unit.

The `Ordering` parameter specifies memory ordering constraints. `SeqCst` (sequentially consistent) is the strongest ordering, ensuring all threads see operations in a consistent global

order. It's the safest choice but also the most expensive. Weaker orderings like `Relaxed`, `Acquire`, and `Release` offer better performance but require deeper understanding of memory models to use correctly.

Atomics are `Sync` because their operations are inherently thread-safe—that's the point of atomic operations. Combined with `Arc` for shared ownership, they enable high-performance concurrent counters, flags, and other primitives. But they're not a general replacement for mutexes; they only work for values that fit in CPU atomic operations (typically up to 64 or 128 bits) and for operations simple enough to express atomically.

See: Companion Code Section 6

---

## Section 7: Scoped Threads and Borrowed Data

Standard `thread::spawn` requires its closure to have a `'static` lifetime—it can't borrow local data because there's no guarantee the data outlives the thread. This often forces you to use `Arc` even when you don't conceptually need shared ownership. **Scoped threads** solve this elegantly.

The `scoped_threads_safe` function uses `thread::scope`, introduced in Rust 1.63, to spawn threads that can borrow from their enclosing scope. The key insight is that `thread::scope` guarantees all spawned threads are joined before the scope exits. This lifetime guarantee lets the compiler verify that borrowed data outlives all threads that use it.

Inside the scope, two threads borrow `data` immutably—no `Arc` needed. After the scope completes, all threads have definitely finished, so the main thread can safely push to `data` again. The scoped threads borrowed the vector; they didn't own it.

This connects directly to the ownership principles from lesson four. Scoped threads extend Rust's borrowing rules across thread boundaries. Multiple threads can share immutable borrows, or one thread can have a mutable borrow, but the scope mechanism ensures the borrows don't outlive the data. It's the same safety guarantees, applied to concurrent code.

See: Companion Code Section 7

---

## Section 8: Testing Concurrent Code

The test module demonstrates an important point: Rust's thread safety guarantees make concurrent code testable in ways that are simply impossible in languages with data races. When the type system eliminates undefined behavior, tests become deterministic.

The `test_mutex_correctness` test spawns ten threads, each incrementing a shared counter 100 times, then asserts the final value is exactly 1,000. This test will pass every time, on every machine, under any load. The equivalent test in C would be flaky—sometimes passing, sometimes failing, depending on timing and scheduling.

Similarly, `test_atomic_correctness` verifies that atomic operations produce correct results under concurrent access. `test_message_passing` confirms that sent values arrive correctly. `test_rwlock` verifies that read locks provide access to the correct data.

These tests are valuable not because they might catch data races—they can't, since Rust prevents data races—but because they verify the logic of the concurrent algorithms. When you write concurrent Rust, you're testing your algorithm, not hoping to catch timing-dependent bugs that might or might not manifest.

See: Companion Code Section 8

---

## Section 9: The Hierarchy of Concurrent Primitives

Looking at all six patterns together, a hierarchy emerges based on overhead and flexibility. At the top is message passing—the safest model with no shared state at all, but with communication overhead. Next come scoped threads, which enable borrowing without ownership costs but require structured lifetimes.

`RwLock` offers better read performance than `Mutex` for read-heavy workloads, but with more complex semantics. `Mutex` provides simpler exclusive access with moderate overhead. At the bottom, atomics offer the lowest overhead but the least flexibility—only simple operations on small values.

The choice between these primitives depends on your specific needs. A web server's request counter might use atomics for minimal overhead. A cache that's read frequently but updated rarely benefits from `RwLock`. A complex data structure with many fields needs `Mutex`. Completely independent tasks communicate best through channels.

What's remarkable is that Rust makes all these options type-safe. You can't accidentally use the wrong synchronization for your data. You can't forget to lock before accessing. You can't send non-thread-safe types across threads. The type system guides you toward correct concurrent code.

See: Companion Code Section 9

---

## Key Takeaways

First, Rust prevents data races at compile time through the `Send` and `Sync` marker traits, which the compiler automatically derives based on type composition and automatically enforces at thread boundaries.

Second, `Mutex<T>` ties the lock to the data it protects, making it impossible to access the data without acquiring the lock—a design pattern that eliminates the "forgot to lock" class of bugs.

Third, `RwLock<T>` encodes the familiar "many readers or one writer" pattern into types, with `RwLockReadGuard` and `RwLockWriteGuard` providing appropriately restricted access.

Fourth, message passing through channels offers an alternative to shared state, transferring ownership between threads instead of sharing data with synchronized access.

Fifth, atomic types enable lock-free programming for simple values, with memory ordering parameters that let you trade safety for performance when you understand the implications.

Sixth, scoped threads extend borrowing rules across thread boundaries by guaranteeing all threads complete before the scope exits, enabling borrowed data in concurrent code without lifetime complications.

---

## What to Read Next

How does this file fit into the broader memory safety taxonomy? Return to `src/lib.rs` to see how data race prevention connects to the other safety guarantees Rust provides.

Want to see these patterns in action with more context? The `examples/data_race_prevention.rs` file from lesson five shows these same concepts with additional commentary and simpler examples.

How do ownership concepts from single-threaded code relate to thread safety? Review `src/use_after_free_prevention.rs` to see how the same ownership principles that prevent use-after-free also underpin thread safety.

Ready to explore more advanced concurrent patterns? Look into the `crossbeam` crate, which provides additional primitives like lock-free queues and more sophisticated scoped threading utilities that build on these foundations.

# First Encounter: Running Safe Code: Code Companion

Reference code for the First Encounter: Running Safe Code lecture. Sections correspond to the lecture document.

## Section 1: The Anatomy of a Rust Entry Point

```rust
fn main() {
    // The simplest entry point: no arguments, returns unit type ()
    // Rust's runtime has already initialized safely before this runs

    println!("=== Buffer Overflow Prevention in Rust ===\n");

    // ... demonstration code follows
}
```

The implicit return type is `()` (unit). For error-handling entry points, you could use `fn main() -> Result<(), E>` or `fn main() -> ExitCode`.

## Section 2: Module Imports and the Crate Ecosystem

```rust
// Import the buffer_overflow module from our crate
use rust_memory_safety_examples::buffer_overflow;

fn main() {
    // Call functions using the module path
    buffer_overflow::safe_array_access();
    //     ^-- module      ^-- function

    buffer_overflow::safe_string_handling();

    buffer_overflow::compare_c_vs_rust();
}
```

The `::` path separator navigates the module hierarchy. The `use` statement brings the module into scope, allowing `buffer_overflow::function()` instead of the full path `rust_memory_safety_examples::buffer_overflow::function()`.

---

## Section 3: Documentation Comments and Intent

```
//! Buffer overflow prevention example
//  ^-- The `!` makes this an INNER doc comment
//      It documents the containing item (the whole file)

// Regular comments use just //
// Doc comments for items below them use ///

use rust_memory_safety_examples::buffer_overflow;
```

Inner doc comments (`//!`) become module-level documentation when you run `cargo doc`. Outer doc comments (`///`) document the item that follows them.

---

## Section 4: Sequential Demonstration Structure

```
fn main() {
    // Header with visual separator
    println!("=== Buffer Overflow Prevention in Rust ===\n");
    //                                              ^-- \n adds newline

    // Section 1: Basic array safety
    println!("1. Safe Array Access");
    buffer_overflow::safe_array_access();

    // Section 2: Heap-allocated string safety
    println!("\n2. Safe String Handling");
    //        ^-- leading newline separates from previous output
    buffer_overflow::safe_string_handling();

    // Section 3: Cross-language comparison
    println!("\n3. C vs Rust Comparison");
    buffer_overflow::compare_c_vs_rust();

    // ... summary follows
}
```

The `println!` macro (note the `!`) accepts format strings with escape sequences. The numbered progression creates a teachable output structure.

---

## Section 5: Function Calls as Safety Boundaries

```
// These functions return () - they exist for side effects
buffer_overflow::safe_array_access();      // -> ()
buffer_overflow::safe_string_handling();   // -> ()
buffer_overflow::compare_c_vs_rust();      // -> ()

// Each call is a contained experiment:
// - If bounds are violated, Rust panics safely
// - No memory corruption can occur
// - No undefined behavior possible
```

Functions returning `()` are common in example code. Production code typically returns `Result<T, E>` for testability and error propagation via the `?` operator.

---

## Section 6: The Summary Output Pattern

```
    println!("\n=== Key Takeaways ===");

    // Unicode works without imports - Rust strings are UTF-8 by default
    println!("✓ Rust prevents buffer overflows at compile time and runtime");
    //        ^-- Unicode checkmark: U+2713

    println!("✓ Bounds checking is automatic and cannot be bypassed");
    println!("✓ No undefined behavior - panics are safe and recoverable");
    println!("✓ Strings automatically resize - no fixed buffer vulnerabilities");
```

Rust's `String` and `&str` types guarantee valid UTF-8 encoding. The compiler enforces this invariant, making string handling both safe and Unicode-aware by default.

## Section 7: Connecting to the Library

```
Project structure:

rust-memory-safety-examples/
├── Cargo.toml
├── src/
│   ├── lib.rs              <- Library root, exports modules
│   └── buffer_overflow.rs  <- Contains the actual demonstrations
└── examples/
    └── buffer_overflow_prevention.rs  <- THIS FILE (you are here)
```

```
// In lib.rs (not shown), something like:
pub mod buffer_overflow;

// This example links against the library via Cargo:
// $ cargo run --example buffer_overflow_prevention
```

The `examples/` directory has special meaning to Cargo. Files here compile as separate binaries that depend on the main library crate.

---

## Section 8: What Running This Code Teaches

```rust
// The complete file - notice what's ABSENT:
//! Buffer overflow prevention example

use rust_memory_safety_examples::buffer_overflow;

fn main() {
    println!("=== Buffer Overflow Prevention in Rust ===\n");

    println!("1. Safe Array Access");
    buffer_overflow::safe_array_access();

    println!("\n2. Safe String Handling");
    buffer_overflow::safe_string_handling();

    println!("\n3. C vs Rust Comparison");
    buffer_overflow::compare_c_vs_rust();

    println!("\n=== Key Takeaways ===");
    println!("✓ Rust prevents buffer overflows at compile time and runtime");
    println!("✓ Bounds checking is automatic and cannot be bypassed");
    println!("✓ No undefined behavior - panics are safe and recoverable");
    println!("✓ Strings automatically resize - no fixed buffer vulnerabilities");
}

// What you DON'T see:
// - No manual memory allocation
// - No null checks
// - No buffer size tracking
// - No cleanup code
// - No error-prone pointer arithmetic
```

The absence of defensive code is the point. Rust's type system and ownership model handle these concerns, letting you focus on logic rather than memory management.

---

## Quick Reference

| Element | Syntax | Purpose |
|---|---|---|
| Entry point | `fn main()` | Program execution starts here |
| Module import | `use crate::module` | Brings module into scope |
| Path separator | `::` | Navigates module hierarchy |

| | | |
|---|---|---|
| Inner doc comment | `//!` | Documents containing item |
| Outer doc comment | `///` | Documents following item |
| Macro call | `name!()` | Compile-time code expansion |
| Unit type | `()` | Empty return / no value |

## Common Entry Point Signatures

```
fn main() { }                           // Simple, no error handling
fn main() -> Result<(), Error> { }      // Can use ? operator
fn main() -> ExitCode { }               // Explicit exit codes (Rust 1.61+)
```

## Running Examples with Cargo

```
# Run this specific example
cargo run --example buffer_overflow_prevention

# Run with release optimizations
cargo run --example buffer_overflow_prevention --release

# List all available examples
cargo run --example
```

# The Big Picture: Memory Safety Taxonomy: Code Companion

Reference code for the The Big Picture: Memory Safety Taxonomy lecture. Sections correspond to the lecture document.

## Section 1: The Crate-Level Documentation Philosophy

```
//! # Rust Memory Safety Examples
//!
//! Educational examples demonstrating memory-safe programming patterns in Rust
//! for financial systems and critical infrastructure.
//!
//! ## Alignment with Federal Guidance
//!
//! These examples align with 2024 CISA/FBI guidance recommending memory-safe
//! languages for critical infrastructure to eliminate 70% of security
vulnerabilities.

// Inner doc comments (//!) describe the enclosing module/crate
// Regular doc comments (///) describe the following item

#![allow(clippy::empty_line_after_doc_comments)]
#![allow(clippy::mixed_attributes_style)]
#![allow(dead_code)]                    // Examples may not all be called
#![allow(clippy::vec_init_then_push)]   // Pedagogical patterns allowed
#![allow(clippy::approx_constant)]
#![allow(clippy::useless_vec)]
```

The `#![...]` syntax (with `!`) applies attributes to the enclosing item (the crate), while `#[...]` applies to the following item. These allows silence warnings that would distract from the educational content.

## Section 2: Module Organization and the Prevention Taxonomy

```rust
// External modules - Rust looks for these as separate files:
//   src/buffer_overflow_prevention.rs
//   src/data_race_prevention.rs
//   src/use_after_free_prevention.rs
pub mod buffer_overflow_prevention;
pub mod data_race_prevention;
pub mod use_after_free_prevention;

// Inline modules - defined directly in this file
pub mod buffer_overflow {
    //! Buffer overflow prevention through bounds checking
    // ... module contents ...
}

pub mod use_after_free {
    //! Use-after-free prevention through ownership
    // ... module contents ...
}

pub mod data_race { /* ... */ }
pub mod integer_overflow { /* ... */ }
pub mod null_pointer { /* ... */ }
pub mod double_free { /* ... */ }
pub mod uninitialized_memory { /* ... */ }
pub mod memory_leak { /* ... */ }
pub mod type_confusion { /* ... */ }
```

The taxonomy covers eight vulnerability classes. External modules (`pub mod name;`) reference separate files; inline modules (`pub mod name { }`) are self-contained within `lib.rs`.

---

## Section 3: Buffer Overflow Prevention Through Bounds Checking

```rust
pub mod buffer_overflow {
    /// Safe array access - Rust prevents buffer overflows
    pub fn safe_array_access() {
        let data = vec![1, 2, 3, 4, 5];

        // Option-returning access: explicitly handles out-of-bounds
        if let Some(&value) = data.get(10) {
            println!("Value: {}", value);
        } else {
            println!("Index out of bounds - safely handled!");
        }

        // Direct indexing panics on invalid index (not undefined behavior)
        // let value = data[10]; // Would panic with clear error message
    }

    /// Safe string handling - no buffer overflows
    pub fn safe_string_handling() {
        let mut buffer = String::new();

        // String grows automatically - no fixed buffer to overflow
        for i in 0..1000 {
            buffer.push_str(&format!("Item {}, ", i));
        }

        println!("Buffer safely holds {} bytes", buffer.len());
    }

    /// Comparing C vs Rust buffer handling
    pub fn compare_c_vs_rust() {
        // C: strcpy(buffer, "This is way too long"); // Buffer overflow!

        // Rust: Safe truncation using iterators
        let buffer = "This is way too long".to_string();
        let truncated: String = buffer.chars().take(10).collect();
        // chars() handles UTF-8 correctly; take(10) limits to 10 characters
    }
}
```

The `get()` method returns `Option<&T>`, forcing explicit handling of the out-of-bounds case. Direct indexing with `[]` performs bounds checking and panics rather than producing undefined behavior.

## Section 4: Use-After-Free Prevention Through Ownership

```rust
pub mod use_after_free {
    /// Ownership prevents use-after-free
    pub fn ownership_prevents_uaf() {
        let data = vec![1, 2, 3, 4, 5];

        // Ownership transfers completely - 'data' becomes invalid
        let owned_data = data;

        // This would not compile:
        // println!("{:?}", data); // ERROR: value used after move

        println!("Data safely owned: {:?}", owned_data);
    }

    /// Borrowing prevents dangling references
    pub fn borrowing_prevents_dangling() {
        let data = vec![1, 2, 3, 4, 5];

        // Create a reference (borrow)
        let reference = &data;

        // Cannot drop data while reference exists:
        // drop(data); // ERROR: cannot move out of `data` because it is borrowed

        println!("Reference is valid: {:?}", reference);
        // data dropped here, after reference's last use
    }

    /// Comparing C vs Rust lifetime management
    pub fn compare_c_vs_rust() {
        // C code (UNSAFE):
        // int* ptr;
        // {
        //     int x = 42;
        //     ptr = &x;
        // }
        // printf("%d", *ptr); // Use-after-free!

        // Rust equivalent (won't compile):
        /*
        let ptr: &i32;
        {
            let x = 42;
            ptr = &x; // ERROR: `x` does not live long enough
        }
        */
    }
}
```

Move semantics make the original binding invalid at compile time. The borrow checker tracks reference lifetimes and prevents any reference from outliving its referent.

## Section 5: Data Race Prevention Through Type-System Enforcement

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

    /// Arc and Mutex for safe concurrent access
    pub fn safe_concurrent_access() {
        // Arc: Atomic Reference Counting (thread-safe shared ownership)
        // Mutex: Mutual exclusion (only one thread accesses at a time)
        let counter = Arc::new(Mutex::new(0));
        let mut handles = vec![];

        for _ in 0..10 {
            let counter_clone = Arc::clone(&counter);  // Clone the Arc, not the
data
            let handle = thread::spawn(move || {
                // lock() returns a guard that auto-unlocks when dropped
                let mut num = counter_clone.lock().unwrap();
                *num += 1;
            });
            handles.push(handle);
        }

        for handle in handles {
            handle.join().unwrap();
        }

        println!("Final count (safe): {}", *counter.lock().unwrap());
    }

    /// Send and Sync traits prevent data races
    pub fn type_system_prevents_races() {
        // This would not compile (Rc is not Send):
        // use std::rc::Rc;
        // let data = Rc::new(vec![1, 2, 3]);
        // thread::spawn(move || {
        //     println!("{:?}", data); // ERROR: Rc cannot be sent between
threads
        // });

        // Rc lacks atomic operations, so it's not thread-safe
        // The compiler knows this via the Send trait bound
    }
}
```

`Arc<Mutex<T>>` combines thread-safe reference counting with mutual exclusion. The `Send` and `Sync` marker traits encode thread-safety requirements in the type system—types that aren't safe to share across threads simply don't implement these traits.

---

## Section 6: Integer Overflow Protection Through Explicit Arithmetic

```
pub mod integer_overflow {
    /// Checked arithmetic prevents silent overflows
    pub fn checked_arithmetic() {
        let a: u32 = 4_000_000_000;
        let b: u32 = 1_000_000_000;

        // checked_add returns Option<u32>: Some(result) or None on overflow
        match a.checked_add(b) {
            Some(result) => println!("Result: {}", result),
            None => println!("Overflow detected and handled safely!"),
        }

        // Debug builds: a + b would panic on overflow
        // Release builds: wraps by default (use checked_ methods for safety)
    }

    /// Saturating arithmetic for financial calculations
    pub fn saturating_arithmetic() {
        let balance: u32 = 1000;
        let withdrawal: u32 = 2000;

        // saturating_sub clamps at 0 instead of wrapping to u32::MAX - 1000
        let new_balance = balance.saturating_sub(withdrawal);

        println!("Balance after withdrawal: {} (saturated)", new_balance);
        // Prints: 0 (not 4294966296)
    }
}
```

Rust provides four arithmetic variants: default (panics in debug, wraps in release), `checked_` (returns `Option`), `saturating_` (clamps to min/max), and `wrapping_` (explicit wrap). Choose based on your domain requirements.

---

## Section 7: Null Pointer Prevention Through Option

```rust
pub mod null_pointer {
    /// Option<T> eliminates null pointer dereferences
    pub fn option_prevents_null() {
        fn find_user(id: u32) -> Option<String> {
            if id == 1 {
                Some("Alice".to_string())
            } else {
                None  // No null pointer, just the None variant
            }
        }

        // Must explicitly handle both cases
        match find_user(1) {
            Some(name) => println!("Found user: {}", name),
            None => println!("User not found"),
        }

        // Can't accidentally dereference - this won't compile:
        // let name = find_user(99);
        // println!("{}", name); // ERROR: Option<String> doesn't implement
Display

        // Must unwrap explicitly (acknowledging the risk):
        // find_user(99).unwrap() // Panics on None
    }
}
```

`Option<T>` is an enum with two variants: `Some(T)` and `None`. The type system forces you to handle the `None` case—there's no way to accidentally dereference a null.

---

## Section 8: Double-Free and RAII Patterns

```rust
pub mod double_free {
    /// Ownership ensures single free
    pub fn ownership_prevents_double_free() {
        let data = vec![1, 2, 3, 4, 5];

        // Attempting to free twice won't compile:
        // drop(data);
        // drop(data); // ERROR: use of moved value

        println!("Data will be freed exactly once: {:?}", data);
    } // data freed here automatically via Drop trait

    /// Box demonstrates single ownership
    pub fn box_single_ownership() {
        let boxed_value = Box::new(42);  // Heap allocation

        let moved_box = boxed_value;     // Ownership transfers

        // This would not compile:
        // drop(boxed_value); // ERROR: value used after move

        println!("Boxed value: {}", moved_box);
    } // moved_box freed here, exactly once
}

pub mod memory_leak {
    use std::fs::File;
    use std::io::Write;

    /// Drop trait for custom cleanup (RAII pattern)
    pub fn drop_trait_cleanup() {
        struct DatabaseConnection {
            id: u32,
        }

        impl Drop for DatabaseConnection {
            fn drop(&mut self) {
                // Called automatically when value goes out of scope
                println!("Closing database connection: {}", self.id);
            }
        }

        {
            let _conn = DatabaseConnection { id: 1 };
            println!("Database connection open");
        } // Drop::drop() called here automatically
    }
}
```

The `Drop` trait enables RAII—resources are cleaned up automatically when values go out of scope. Combined with ownership rules, this guarantees exactly-once cleanup without manual memory

management.

## Section 9: Type Safety and the Newtype Pattern

```
pub mod type_confusion {
    /// Newtype pattern for type safety
    pub fn newtype_pattern() {
        // Wrap primitive types to create distinct types
        struct UserId(u32);      // Tuple struct wrapping u32
        struct ProductId(u32);   // Different type, same underlying data

        let user = UserId(123);
        let product = ProductId(456);

        // This would not compile - different types!
        // if user == product { } // ERROR: mismatched types
        // fn process_user(id: UserId) - won't accept ProductId

        // Access inner value with .0
        println!("User ID: {}, Product ID: {}", user.0, product.0);
    }

    /// Enum prevents invalid states
    pub fn enum_prevents_invalid_states() {
        enum ConnectionState {
            Disconnected,
            Connecting,
            Connected { session_id: String },  // Data only exists when Connected
            Error { message: String },
        }

        let state = ConnectionState::Connected {
            session_id: "abc123".to_string(),
        };

        // Pattern matching forces handling of all variants
        match state {
            ConnectionState::Connected { session_id } => {
                // session_id only accessible in this branch
                println!("Connected with session: {}", session_id);
            }
            ConnectionState::Disconnected => println!("Disconnected"),
            ConnectionState::Connecting => println!("Connecting..."),
            ConnectionState::Error { message } => println!("Error: {}", message),
        }
    }
}
```

The newtype pattern creates distinct types from primitives, preventing accidental mixing of semantically different values. Enums with associated data ensure that data is only accessible in valid states.

---

## Quick Reference

### Vulnerability Prevention Summary

| Vulnerability Class | Rust Prevention Mechanism | Key Types/Traits |
|---|---|---|
| Buffer Overflow | Bounds checking | `Vec::get()`, `slice[i]` |
| Use-After-Free | Ownership + borrowing | Move semantics, `&T`, `&mut T` |
| Data Races | Type system enforcement | `Send`, `Sync`, `Arc<Mutex<T>>` |
| Integer Overflow | Explicit arithmetic | `checked_add()`, `saturating_sub()` |
| Null Pointer | Option type | `Option<T>`, `Some`, `None` |
| Double-Free | Single ownership | Drop trait, move semantics |
| Uninitialized Memory | Initialization required | Compiler enforcement |
| Type Confusion | Strong typing | Newtypes, enums |

### Arithmetic Method Variants

| Method Pattern | Behavior on Overflow | Return Type |
|---|---|---|
| `a + b` | Panic (debug) / Wrap (release) | `T` |
| `a.checked_add(b)` | Returns `None` | `Option<T>` |

| | | |
|---|---|---|
| `a.saturating_add(b)` | Clamps to `T::MAX` | `T` |
| `a.wrapping_add(b)` | Wraps around | `T` |
| `a.overflowing_add(b)` | Wraps, returns flag | `(T, bool)` |

## Module Organization Pattern

```
src/lib.rs
├── External modules (pub mod name;)     → src/name.rs
└── Inline modules (pub mod name { })    → defined in lib.rs
```

# Bounds Checking Deep Dive: Code Companion

Reference code for the Bounds Checking Deep Dive lecture. Sections correspond to the lecture document.

---

## Section 1: The Vulnerability Rust Eliminates

```c
#include <string.h>

void vulnerable_copy(char* dest, const char* src) {
    strcpy(dest, src);  // NO BOUNDS CHECKING!
    // strcpy copies until null terminator, ignoring dest size
}

int main() {
    char buffer[10];  // Only 10 bytes allocated on stack
    char* long_string = "This is a very long string that will overflow";
    // This string is 46 characters - 36 bytes overflow!

    vulnerable_copy(buffer, long_string);  // BUFFER OVERFLOW!
    // Those extra 36 bytes overwrite the stack frame,
    // potentially corrupting the return address
    return 0;
}
```

This C code represents the class of vulnerability that Rust eliminates by design. The `strcpy` function has no way to know the destination buffer's size—it's just a raw pointer with no length information.

---

## Section 2: Rust's Safe Copy Pattern

```rust
/// Rust's safe equivalent to strcpy
/// Note: slices are "fat pointers" containing (ptr, length)
pub fn safe_copy(dest: &mut [u8], src: &[u8]) -> Result<(), &'static str> {
    // Length information travels WITH the slice - can't be forgotten
    if dest.len() < src.len() {
        return Err("Destination buffer too small");
    }

    // dest[..src.len()] creates a sub-slice of exact size needed
    // copy_from_slice performs its own bounds verification
    dest[..src.len()].copy_from_slice(src);
    Ok(())  // Result type forces caller to handle potential failure
}
```

The signature `&mut [u8]` and `&[u8]` are slices—fat pointers that carry their length. The `Result` return type makes error handling explicit and non-ignorable.

## Section 3: The Two Faces of Indexed Access

```rust
pub fn safe_array_access() {
    let array = [1, 2, 3, 4, 5];

    // Direct indexing: bounds-checked at runtime, panics if invalid
    let _first = array[0];  // OK - returns i32 directly

    // This would panic with a clear error message:
    // let _invalid = array[10];  // thread 'main' panicked at 'index out of
bounds'

    // Safe alternative: get() returns Option<&T>
    match array.get(10) {
        Some(value) => println!("Value: {}", value),  // Never reached here
        None => println!("Index out of bounds (safely handled)"),
    }
}
```

| Access Method | Return Type | Out-of-Bounds Behavior | Use When |
|---|---|---|---|
| `array[i]` | `T` | Panics immediately | Bug if index invalid |
| `array.get(i)` | `Option<&T>` | Returns `None` | Invalid index is expected |

## Section 4: Vectors and Dynamic Safety

```rust
pub fn safe_vector_usage() {
    let mut vec = Vec::new();  // Dynamically-sized, heap-allocated
    vec.push(1);
    vec.push(2);
    vec.push(3);

    // Iterator-based access: bounds checking happens once at creation
    // Cannot iterate past the end - iterator knows exact length
    for item in &vec {
        println!("{}", item);
    }

    // get() works identically to arrays - same Option<&T> pattern
    if let Some(value) = vec.get(5) {
        println!("Value at index 5: {}", value);
    } else {
        println!("Index 5 doesn't exist");  // This branch executes
    }
}
```

The iterator `&vec` yields exactly `vec.len()` references. The `if let` syntax is a concise way to handle the `Some` case when you don't need the `None` branch to do anything special.

---

## Section 5: Strings as Growable Buffers

```rust
pub fn safe_string_operations() {
    // Initial capacity is just a hint - not a hard limit
    let mut dest = String::with_capacity(10);
    let src = "This is a very long string that would overflow a fixed buffer";

    // In C: catastrophic overflow. In Rust: automatic reallocation
    dest.push_str(src);  // String grows to accommodate 61 characters

    println!("String length: {} (automatically managed)", dest.len());
    // Prints: String length: 61 (automatically managed)
}
```

`String::with_capacity(10)` pre-allocates 10 bytes but imposes no limit. When `push_str` needs more space, `String` automatically reallocates to a larger buffer.

---

## Section 6: Compile-Time Size Enforcement

```
pub fn demonstration_bounds_checking() {
    let buffer: [u8; 10] = [0; 10];  // Fixed-size array: 10 bytes
    let data: [u8; 20] = [1; 20];    // Fixed-size array: 20 bytes

    // This would NOT compile - size mismatch caught at compile time:
    // buffer.copy_from_slice(&data);
    // error: source slice length (20) does not match destination slice length
(10)

    // Safe alternative: explicitly slice to matching size
    let safe_copy = &data[..buffer.len()];  // Take first 10 bytes only
    let mut mutable_buffer = buffer;
    mutable_buffer.copy_from_slice(safe_copy);  // Now sizes match

    println!("Safe copy completed: {:?}", mutable_buffer);
    // Prints: Safe copy completed: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
}
```

When array sizes are known at compile time, the compiler catches mismatches before the code ever runs. The slice expression `&data[..buffer.len()]` creates a view of exactly 10 elements.

## Section 7: Testing Bounds Behavior

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

    #[test]
    fn test_safe_copy_success() {
        let mut dest = [0u8; 10];
        let src = [1, 2, 3, 4, 5];

        assert!(safe_copy(&mut dest, &src).is_ok());
        assert_eq!(&dest[..5], &src);  // First 5 bytes copied
    }

    #[test]
    fn test_safe_copy_overflow_prevented() {
        let mut dest = [0u8; 5];
        let src = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

        // Overflow attempt returns Err, buffer unchanged
        assert!(safe_copy(&mut dest, &src).is_err());
    }

    #[test]
    #[should_panic]  // Test expects a panic - controlled failure
    #[allow(unconditional_panic, clippy::out_of_bounds_indexing)]
    fn test_out_of_bounds_panic() {
        let array = [1, 2, 3];
        let _ = array[10];  // Panics - deterministic, debuggable failure
    }

    #[test]
    fn test_safe_get() {
        let array = [1, 2, 3];
        assert_eq!(array.get(1), Some(&2));   // Valid index
        assert_eq!(array.get(10), None);       // Invalid index - no panic
    }
}
```

The `#[should_panic]` attribute lets you verify that dangerous operations fail predictably. Unlike C's undefined behavior, Rust's panics are consistent and testable.

## Quick Reference

### Slice Types

| Type | Description | Length Known |
|---|---|---|
| `[T; N]` | Fixed-size array | Compile time |
| `&[T]` | Immutable slice (fat pointer) | Runtime |
| `&mut [T]` | Mutable slice (fat pointer) | Runtime |
| `Vec<T>` | Growable vector | Runtime |
| `String` | Growable UTF-8 string | Runtime |

## Access Methods

```
// Direct indexing - panics on out-of-bounds
let value: T = collection[index];

// Safe access - returns Option
let maybe_value: Option<&T> = collection.get(index);

// Slice creation - panics if range invalid
let sub: &[T] = &collection[start..end];

// Safe slice - returns Option
let maybe_sub: Option<&[T]> = collection.get(start..end);
```

## Key Copy Operations

```
// Requires exact size match - panics otherwise
dest.copy_from_slice(src);

// Clone elements (requires T: Clone)
dest.clone_from_slice(src);

// Safe copying via slicing
dest[..src.len()].copy_from_slice(src);
```

## C vs Rust Comparison

| C Pattern | Rust Equivalent | Safety Mechanism |
|---|---|---|

| | | |
|---|---|---|
| `strcpy(dest, src)` | `dest.copy_from_slice(src)` | Size match required |
| `array[i]` | `array.get(i)` | Returns `Option<&T>` |
| `malloc` + manual tracking | `Vec::new()` | Length stored with data |
| `char buffer[N]` | `String` | Automatic growth |

# rust-memory-safety-examples examples/use_after_free_prevention.rs: Code Companion

Reference code for the Ownership in Action lecture. Sections correspond to the lecture document.

## Section 1: The Structure of a Demonstration Runner

```rust
//! Use-after-free prevention example

use rust_memory_safety_examples::use_after_free;

fn main() {
    println!("=== Use-After-Free Prevention in Rust ===\n");

    // Each call delegates to library module functions
    println!("1. Ownership Prevents Use-After-Free");
    use_after_free::ownership_prevents_uaf();

    println!("\n2. Borrowing Prevents Dangling References");
    use_after_free::borrowing_prevents_dangling();

    println!("\n3. C vs Rust Comparison");
    use_after_free::compare_c_vs_rust();

    // Summary output reinforces educational goals
    println!("\n=== Key Takeaways ===");
    println!("✓ Rust's ownership system makes use-after-free impossible");
    println!("✓ Compile-time prevention - errors caught before execution");
    println!("✓ Lifetimes ensure references are always valid");
    println!("✓ No runtime overhead - zero-cost abstraction");
}
```

The entire `main()` function is an orchestrator—no memory safety logic lives here. All demonstrations are imported from `use_after_free` module in the library crate.

## Section 2: Understanding Use-After-Free

```
// What use-after-free looks like in C (CONCEPTUAL - this is unsafe code)
//
// char* ptr = malloc(100);    // Allocate memory
// strcpy(ptr, "hello");       // Use memory
// free(ptr);                  // Free memory
// printf("%s", ptr);          // BUG: Use after free!
//                             // ptr still holds the address, but memory is gone

// The freed memory might:
// - Still contain "hello" (appears to work)
// - Contain garbage (crashes or wrong output)
// - Be reallocated for something else (security vulnerability)
```

This conceptual C code shows why use-after-free is dangerous: the pointer `ptr` remains valid syntactically even after `free()`, but the memory it points to is no longer owned by the program.

## Section 3: Ownership as Temporal Safety

```
// How Rust's ownership prevents use-after-free
fn ownership_prevents_uaf() {
    let data = String::from("hello");  // data owns the String

    let moved_data = data;  // Ownership transfers to moved_data
                            // data is now invalid - compiler enforces this

    // println!("{}", data);  // COMPILE ERROR: value borrowed after move
    //                        // ^^^^^^ This line would not compile

    println!("{}", moved_data);  // Only the current owner can access
}   // moved_data goes out of scope, memory is freed
    // No dangling pointers possible - the only handle is gone
```

The `move` semantic is Rust's key insight: when ownership transfers, the previous variable becomes unusable. The compiler tracks this statically—there's no runtime check.

## Section 4: Borrowing and the Lifetime Connection

```rust
// Borrowing with lifetime enforcement
fn borrowing_prevents_dangling() {
    let reference: &String;  // Declare a reference (no value yet)

    {
        let data = String::from("temporary");
        // reference = &data;  // COMPILE ERROR: `data` does not live long enough
        //                     // The borrow would outlive the owner
    }   // data dropped here - if reference pointed to it, it would dangle

    // Correct approach: reference must not outlive referent
    let data = String::from("lives long enough");
    reference = &data;  // OK: data lives as long as reference needs
    println!("{}", reference);
}   // Both data and reference go out of scope together - safe

// The compiler error would look like:
// error[E0597]: `data` does not live long enough
//    |     reference = &data;
//    |                 ^^^^^ borrowed value does not live long enough
//    | }
//    | - `data` dropped here while still borrowed
```

Lifetimes are the compiler's way of tracking "how long does this reference need to be valid?" When a reference would outlive its referent, compilation fails.

---

## Section 5: The Compile-Time Advantage

```rust
// These print statements emphasize the compile-time nature
println!("✓ Compile-time prevention - errors caught before execution");
println!("✓ No runtime overhead - zero-cost abstraction");

// Comparison of safety approaches:
//
// | Approach           | When Detected | Runtime Cost | Coverage    |
// |--------------------|---------------|--------------|-------------|
// | AddressSanitizer   | Runtime       | 2-3x slower  | Tested paths|
// | Garbage Collection| Runtime       | GC pauses    | Complete    |
// | Rust Ownership     | Compile-time  | Zero         | Complete    |
```

The key distinction: Rust's safety checks happen entirely during compilation. The resulting binary contains no ownership checks, no reference counting, no garbage collector—just the logic you wrote.

## Section 6: C vs Rust Mental Models

```rust
// Mental model comparison

// C pointer: "I believe valid data lives here"
// - The compiler trusts you
// - No verification of belief
// - Undefined behavior if wrong

// Rust reference: "The compiler verified valid data lives here"
// - The compiler proves correctness
// - Static analysis of all code paths
// - Won't compile if unverifiable

fn compare_c_vs_rust() {
    // In Rust, you cannot create an invalid reference in safe code

    let valid_ref: &i32;
    let owner = 42;
    valid_ref = &owner;  // Compiler verifies: owner lives long enough

    // There is no Rust equivalent to:
    //   int* ptr = (int*)0xDEADBEEF;  // C: trust me, this is valid
    //   *ptr = 42;                    // Undefined behavior

    println!("Reference always valid: {}", valid_ref);
}
```

Safe Rust makes it impossible to construct a reference to invalid memory. Every reference creation is verified at compile time.

---

## Quick Reference: Ownership vs Borrowing

| Concept | Symbol | Meaning | Example |
|---|---|---|---|
| Ownership | `T` | Owns the value, controls its lifetime | `let s = String::new()` |
| Shared borrow | `&T` | Read-only access, owner retains control | `fn len(s: &String)` |

| | | | |
|---|---|---|---|
| Mutable borrow | `&mut T` | Exclusive read-write access | `fn push(s: &mut String)` |
| Move | (implicit) | Ownership transfers, source invalidated | `let s2 = s1` |

## Key Compile Errors

| Error | Meaning |
|---|---|
| "value borrowed after move" | Tried to use a value after ownership transferred |
| "does not live long enough" | Reference would outlive the data it points to |
| "cannot borrow as mutable" | Tried to mutate through a shared reference |

## The Ownership Rules

1. Each value has exactly one owner
2. When the owner goes out of scope, the value is dropped
3. Ownership can be transferred (moved) or temporarily lent (borrowed)
4. References must never outlive their referent

# rust-memory-safety-examples src/use_after_free_prevention.rs: Code Companion

Reference code for the Ownership and Lifetimes Mastery lecture. Sections correspond to the lecture document.

---

## Section 1: The Vulnerability We're Preventing

```c
// C code demonstrating the use-after-free vulnerability
// This is the pattern Rust's ownership system prevents

int* vulnerable_use_after_free() {
    int* ptr = malloc(sizeof(int));  // Allocate heap memory
    *ptr = 42;                       // Store a value
    free(ptr);                       // Memory freed - ptr is now dangling
    return ptr;                      // DANGER: returning invalid pointer!
}

void exploit() {
    int* p = vulnerable_use_after_free();
    printf("%d\n", *p);  // USE-AFTER-FREE: undefined behavior
    // Memory might still contain 42... or anything else
    // Attacker could control what's now in that memory location
}
```

This C code compiles without warnings but contains a critical security vulnerability. The pointer `ptr` becomes invalid after `free()`, yet nothing stops us from returning and dereferencing it. Rust makes this pattern impossible to express.

## Section 2: Ownership as the First Line of Defense

```rust
pub fn ownership_prevents_uaf() {
    let data = Box::new(42);  // Box<i32>: heap-allocated integer
    let value = *data;        // Copy the i32 value (i32 implements Copy)

    drop(data);  // Explicitly free the heap memory

    // This would NOT compile:
    // println!("{}", *data);  // Error: use of moved value `data`

    println!("Copied value: {}", value);  // Safe: value is independent copy
}
```

The `drop()` function takes ownership of `data`, consuming it. After this point, the compiler marks `data` as invalid. The `*data` dereference copies the integer because `i32: Copy`—primitive types are bitwise copied rather than moved.

## Section 3: Lifetimes Preventing Dangling References

```rust
pub fn lifetime_prevents_dangling_ref() {
    let _reference: Option<&String> = None;  // Reference lives in outer scope

    {
        let _data = String::from("temporary");  // String lives in inner scope

        // This would NOT compile:
        // reference = Some(&_data);
        // Error: `_data` does not live long enough
        //        borrowed value does not live long enough
    }
    // _data is dropped here - its memory freed

    // If the assignment above compiled, _reference would be dangling here
}
```

The lifetime of `_data` is limited to the inner block (the curly braces). The compiler performs lifetime analysis and sees that `_reference` would outlive what it references. This is a compile-time check with zero runtime cost.

## Section 4: Borrowing Rules and Aliasing Control

```rust
pub fn borrowing_prevents_uaf() {
    let mut data = vec![1, 2, 3];  // Vec<i32> with heap-allocated storage

    let reference = &data[0];  // Immutable borrow of first element

    // This would NOT compile while `reference` exists:
    // data.clear();
    // Error: cannot borrow `data` as mutable because it is also
    //        borrowed as immutable

    println!("First element: {}", reference);  // Last use of `reference`
    // Non-Lexical Lifetimes: borrow ends here, not at scope end

    data.clear();  // Now mutation is allowed
    println!("Vector cleared");
}
```

The `clear()` method requires `&mut self`, but an immutable borrow (`reference`) is active. Rust's rule: you can have either one mutable reference OR any number of immutable references, never both. The borrow of `reference` ends after its last use, enabling the subsequent mutation.

---

## Section 5: Reference Counting for Shared Ownership

```rust
use std::rc::Rc;

pub fn shared_ownership_safe() {
    let data = Rc::new(vec![1, 2, 3]);  // Rc<Vec<i32>>: reference-counted
    let clone1 = Rc::clone(&data);       // Increments ref count, doesn't clone
data
    let clone2 = Rc::clone(&data);       // Ref count now 3

    println!("Data from clone1: {:?}", clone1);
    println!("Data from clone2: {:?}", clone2);

    // Memory freed only when ALL Rc handles are dropped
    drop(clone1);  // Ref count: 2
    drop(clone2);  // Ref count: 1
    drop(data);    // Ref count: 0 → memory freed
}
```

`Rc::clone()` is cheap—it increments an integer counter, not a deep copy. The underlying `Vec` is deallocated only when `strong_count` reaches zero. Note: `Rc` is not thread-safe; use `Arc` for multi-threaded scenarios.

---

## Section 6: Consuming Self for Ownership Transfer

```
pub struct SafeObject {
    data: Vec<i32>,
}

impl SafeObject {
    pub fn new(data: Vec<i32>) -> Self {
        Self { data }
    }

    // Borrows self immutably - object still usable after call
    pub fn get_data(&self) -> &[i32] {
        &self.data
    }

    // Takes ownership of self - object consumed, cannot be used again
    pub fn consume(self) -> Vec<i32> {
        self.data  // Moves data out of the struct
    }
}

pub fn safe_object_usage() {
    let obj = SafeObject::new(vec![1, 2, 3]);
    let data_ref = obj.get_data();  // Borrows obj
    println!("Data: {:?}", data_ref);

    let owned_data = obj.consume();  // Moves obj - takes ownership

    // This would NOT compile:
    // obj.get_data();  // Error: value used after move

    println!("Owned data: {:?}", owned_data);
}
```

The method signature tells the whole story: `&self` borrows, `self` consumes. When `consume(self)` takes ownership, the caller loses access to `obj`. This pattern is common for builder types and resource cleanup.

## Section 7: Drop Trait and Deterministic Cleanup

```rust
pub fn compile_time_safety_demo() {
    struct Resource {
        id: i32,
    }

    impl Drop for Resource {
        fn drop(&mut self) {
            println!("Resource {} freed", self.id);
            // Custom cleanup logic runs here
        }
    }

    let res = Resource { id: 1 };
    println!("Resource created: {}", res.id);

    // `res` automatically dropped at scope end
    // Drop::drop() called deterministically - not garbage collected
}
// Output:
// Resource created: 1
// Resource {} freed: 1
```

The `Drop` trait provides a destructor that runs when a value goes out of scope. Unlike garbage collection, this is deterministic—you know exactly when cleanup occurs. The compiler inserts the `drop` call automatically; you cannot call `drop()` method directly (use `std::mem::drop()` to drop early).

## Quick Reference

| Ownership Pattern | Signature | Effect |
|---|---|---|
| Take ownership | `fn foo(x: T)` | Caller loses access to `x` |
| Borrow immutably | `fn foo(x: &T)` | Caller retains access, cannot mutate |
| Borrow mutably | `fn foo(x: &mut T)` | Caller retains access, exclusive mutation |
| Return owned | `fn foo() -> T` | Caller receives ownership |
| Return reference | `fn foo(&self) -> &T` | Lifetime tied to `self` |

## Smart Pointer Comparison

| Type | Ownership | Thread-Safe | Use Case |
|---|---|---|---|
| `Box<T>` | Single owner | N/A (moved) | Heap allocation, recursive types |
| `Rc<T>` | Shared, counted | No | Single-threaded shared ownership |
| `Arc<T>` | Shared, counted | Yes | Multi-threaded shared ownership |

## Key Compiler Errors

```
// "use of moved value"
let a = Box::new(1);
let b = a;
println!("{}", a);  // Error: a was moved to b

// "does not live long enough"
let r;
{ let x = 5; r = &x; }  // Error: x dropped while r borrows it

// "cannot borrow as mutable because also borrowed as immutable"
let mut v = vec![1];
let first = &v[0];
v.push(2);  // Error: push needs &mut, but first holds &
```

# rust-memory-safety-examples examples/data_race_prevention.rs: Code Companion

Reference code for the Concurrent Code Introduction lecture. Sections correspond to the lecture document.

---

## Section 1: From Memory Safety to Thread Safety

```
//! Data race prevention example

// This single use statement brings in all the demonstration functions
// The module system ensures clean separation of concerns
use rust_memory_safety_examples::data_race;
```

The `use` statement imports from a library crate, meaning these demonstrations live in `src/lib.rs` or a submodule. This separation allows the examples to be tested independently from the binary.

---

## Section 2: The Main Function Structure

```rust
fn main() {
    println!("=== Data Race Prevention in Rust ===\n");

    // Each demonstration is clearly numbered and labeled
    println!("1. Safe Concurrent Access with Arc and Mutex");
    data_race::safe_concurrent_access();

    println!("\n2. Type System Prevents Races");
    data_race::type_system_prevents_races();

    println!("\n3. C vs Rust Comparison");
    data_race::compare_c_vs_rust();

    // Summary section reinforces key concepts
    println!("\n=== Key Takeaways ===");
    // ... takeaways follow
}
```

The function calls execute sequentially, ensuring each concurrent demonstration completes before the next begins. This prevents any cross-contamination between examples.

---

## Section 3: Safe Concurrent Access with Arc and Mutex

```rust
// Conceptual structure (from the library module):
// Arc<Mutex<T>> combines two guarantees:

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

// Arc: Atomic Reference Counting - safe to clone across threads
// Mutex: Mutual Exclusion - only one thread accesses inner data at a time
let shared_data: Arc<Mutex<Vec<i32>>> = Arc::new(Mutex::new(Vec::new()));

// Clone Arc to share ownership with spawned thread
let data_for_thread = Arc::clone(&shared_data);

thread::spawn(move || {
    // .lock() returns a Result<MutexGuard<T>, PoisonError>
    // The guard provides exclusive access until dropped
    let mut guard = data_for_thread.lock().unwrap();
    guard.push(42);  // Safe: we hold the lock
});  // Guard drops here, releasing the lock
```

The `MutexGuard` returned by `.lock()` implements `Deref` and `DerefMut`, so you can use it like `&mut T`. The lock is automatically released when the guard goes out of scope.

---

## Section 4: The Type System as Thread Safety Enforcer

```
// Send and Sync are auto-traits - the compiler derives them automatically

// This compiles: i32 is Send
thread::spawn(move || {
    let x: i32 = 42;
    println!("{}", x);
});

// This would NOT compile: Rc is not Send
// use std::rc::Rc;
// let rc = Rc::new(42);
// thread::spawn(move || {
//     println!("{}", rc);  // ERROR: Rc<i32> cannot be sent between threads
safely
// });

// The thread::spawn signature enforces Send:
// pub fn spawn<F, T>(f: F) -> JoinHandle<T>
// where
//     F: FnOnce() -> T + Send + 'static,  // <-- F must be Send
//     T: Send + 'static,                  // <-- Return type must be Send
```

The compiler error for non-`Send` types is precise: "Rc cannot be sent between threads safely." This specificity helps developers understand exactly what needs to change.

---

## Section 5: Connecting to Ownership Fundamentals

```rust
// The move keyword transfers ownership into the closure
let data = vec![1, 2, 3];

let handle = thread::spawn(move || {
    // data is now owned by this thread's closure
    // The original thread can no longer access it
    println!("Thread has: {:?}", data);
});

// This would NOT compile:
// println!("{:?}", data);  // ERROR: value borrowed after move

handle.join().unwrap();

// Without move, the compiler checks lifetimes:
// let local = String::from("hello");
// thread::spawn(|| {
//     println!("{}", local);  // ERROR: local may be dropped while thread runs
// });
```

The `'static` bound on `thread::spawn` requires that all captured data either be owned (`move`) or have `'static` lifetime. This prevents dangling references across thread boundaries.

## Section 6: C vs Rust Comparison

```rust
// In C (pseudocode showing the problem):
// int shared_counter = 0;
// pthread_mutex_t lock;  // The lock exists, but...
//
// void* thread_func(void* arg) {
//     shared_counter++;  // Oops! Forgot to acquire lock
//     return NULL;       // Data race: undefined behavior
// }

// In Rust, the type system prevents this:
let counter = Arc::new(Mutex::new(0));

let counter_clone = Arc::clone(&counter);
thread::spawn(move || {
    // Cannot access the i32 without going through the Mutex
    // counter_clone += 1;  // ERROR: no += on Arc<Mutex<i32>>

    // Must acquire lock to access the inner value
    let mut num = counter_clone.lock().unwrap();
    *num += 1;  // Safe: lock is held
});
```

The `Mutex<T>` wrapper makes the protected data inaccessible without acquiring the lock. There's no way to "forget" to lock—the type system physically prevents unguarded access.

---

## Section 7: The Key Takeaways Section

```
println!("\n=== Key Takeaways ===");
println!("✓ Rust prevents data races at compile time");
println!("✓ Send and Sync traits enforce thread safety");
println!("✓ Arc and Mutex provide safe concurrent access");
println!("✓ No undefined behavior in concurrent code");
```

Each takeaway maps to a concrete mechanism: compile-time prevention via the borrow checker, `Send`/`Sync` traits for type-level enforcement, `Arc<Mutex<T>>` for shared mutable state, and the absence of undefined behavior in safe Rust.

---

## Quick Reference

| Type | Purpose | Thread Safety |
|---|---|---|
| `Rc<T>` | Single-threaded reference counting | Not `Send`, not `Sync` |
| `Arc<T>` | Multi-threaded reference counting | `Send + Sync` if `T: Send + Sync` |
| `Mutex<T>` | Mutual exclusion wrapper | `Sync` if `T: Send` |
| `RwLock<T>` | Reader-writer lock | `Sync` if `T: Send + Sync` |

| Trait | Meaning | Auto-derived? |
|---|---|---|
| `Send` | Safe to transfer to another thread | Yes |
| `Sync` | Safe to share references across threads | Yes |

```
// Common pattern: shared mutable state across threads
let shared: Arc<Mutex<T>> = Arc::new(Mutex::new(value));

// Clone for each thread
let thread_copy = Arc::clone(&shared);

// Spawn and move ownership of the clone
thread::spawn(move || {
    let mut guard = thread_copy.lock().unwrap();
    // Use guard as &mut T
});
```

# rust-memory-safety-examples src/data_race_prevention.rs: Code Companion

Reference code for the Thread Safety Through Types lecture. Sections correspond to the lecture document.

---

## Section 1: The Data Race Problem in C

```c
#include <pthread.h>
#include <stdio.h>

int counter = 0;  // Shared mutable state - no protection

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        counter++;  // DATA RACE: read-modify-write is not atomic
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Counter: %d\n", counter);  // Expected 200,000, actual varies
    return 0;
}
```

This C code compiles without warning but exhibits undefined behavior. The `counter++` operation involves three steps (load, increment, store) that can interleave between threads, causing lost updates.

---

## Section 2: Mutex-Protected Shared State

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

pub fn safe_concurrent_counter() {
    // Mutex wraps the data it protects - can't access without locking
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        // Clone the Arc, not the data - shared ownership
        let counter_clone = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            for _ in 0..1000 {
                // lock() returns MutexGuard<i32>, which derefs to &mut i32
                let mut num = counter_clone.lock().unwrap();
                *num += 1;
                // MutexGuard dropped here, lock released automatically
            }
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Final counter value: {}", *counter.lock().unwrap());
    // Always produces correct result: 10,000
}
```

The `Arc<Mutex<T>>` pattern is idiomatic for shared mutable state across threads. `Arc` provides thread-safe reference counting, while `Mutex` ensures exclusive access. The `MutexGuard` implements RAII—the lock is released when the guard goes out of scope.

## Section 3: Read-Write Locks for Concurrent Readers

```rust
use std::sync::{Arc, RwLock};
use std::thread;

pub fn safe_read_write_access() {
    let data = Arc::new(RwLock::new(vec![1, 2, 3]));
    let mut handles = vec![];

    // Multiple readers can acquire read() simultaneously
    for i in 0..5 {
        let data_clone = Arc::clone(&data);
        let handle = thread::spawn(move || {
            // RwLockReadGuard only implements Deref, not DerefMut
            let read_guard = data_clone.read().unwrap();
            println!("Reader {}: {:?}", i, *read_guard);
        });
        handles.push(handle);
    }

    // Writer requires exclusive access via write()
    let data_clone = Arc::clone(&data);
    let writer_handle = thread::spawn(move || {
        // RwLockWriteGuard implements both Deref and DerefMut
        let mut write_guard = data_clone.write().unwrap();
        write_guard.push(4);  // Mutation allowed through write guard
        println!("Writer added element");
    });
    handles.push(writer_handle);

    for handle in handles {
        handle.join().unwrap();
    }
}
```

`RwLock<T>` encodes the borrowing rules at runtime: multiple `&T` or one `&mut T`. The guard types enforce this through their trait implementations—you cannot mutate data through a `RwLockReadGuard`.

## Section 4: Send and Sync — The Marker Traits

```rust
use std::thread;

/// This type is automatically Send because i32 is Send
struct SendableData {
    value: i32,
}

/// This type is NOT Send - raw pointers opt out automatically
struct NotSendable {
    ptr: *mut i32,  // Raw pointers are !Send and !Sync
}

pub fn thread_safety_enforced() {
    let sendable = SendableData { value: 42 };

    // Works: SendableData is Send, so it can cross thread boundaries
    thread::spawn(move || {
        println!("Value in thread: {}", sendable.value);
    });

    // COMPILE ERROR if uncommented:
    // let not_sendable = NotSendable { ptr: std::ptr::null_mut() };
    // thread::spawn(move || {
    //     // error[E0277]: `*mut i32` cannot be sent between threads safely
    //     println!("{:?}", not_sendable.ptr);
    // });
}
```

The `thread::spawn` function has the bound `F: Send + 'static`. Since `NotSendable` contains `*mut i32` (which is `!Send`), the struct itself becomes `!Send`, and the compiler rejects the code. No runtime check needed.

---

## Section 5: Message Passing as an Alternative Model

```rust
use std::sync::mpsc;  // Multiple Producer, Single Consumer
use std::thread;

pub fn safe_message_passing() {
    // channel() returns (Sender<T>, Receiver<T>)
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        for i in 0..5 {
            // send() transfers ownership of i to the channel
            tx.send(i).unwrap();
        }
        // tx dropped here, closing the channel
    });

    // Receiver implements Iterator - blocks until channel closes
    for received in rx {
        println!("Received: {}", received);
    }
}
```

Message passing transfers ownership rather than sharing it. The `Sender<T>` requires `T: Send` because values cross thread boundaries. When all senders are dropped, the channel closes and the receiver's iterator ends.

---

## Section 6: Atomic Types for Lock-Free Programming

```rust
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::Arc;
use std::thread;

pub fn safe_atomic_operations() {
    // AtomicUsize provides lock-free atomic operations
    let counter = Arc::new(AtomicUsize::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter_clone = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            for _ in 0..1000 {
                // fetch_add is a single atomic operation - no lock needed
                counter_clone.fetch_add(1, Ordering::SeqCst);
            }
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    // SeqCst provides strongest ordering guarantee
    println!("Atomic counter: {}", counter.load(Ordering::SeqCst));
}
```

Atomic types provide hardware-level atomic operations without locks. `Ordering::SeqCst` (sequentially consistent) provides the strongest memory ordering guarantee, ensuring all threads see operations in the same order. Weaker orderings like `Relaxed` offer better performance for specific use cases.

## Section 7: Scoped Threads for Borrowed Data

```rust
use std::thread;

pub fn scoped_threads_safe() {
    let mut data = vec![1, 2, 3];

    // thread::scope guarantees all spawned threads join before returning
    thread::scope(|s| {
        s.spawn(|| {
            // Can borrow data - scope guarantees data outlives thread
            println!("Data length: {}", data.len());
        });

        s.spawn(|| {
            // Multiple threads can borrow immutably
            println!("Data: {:?}", data);
        });
    });
    // All threads joined here - scope has ended

    // Safe to modify: we have exclusive access again
    data.push(4);
}
```

Unlike `thread::spawn` which requires `'static` data, `thread::scope` allows borrowing from the parent stack frame. The scope ensures all spawned threads complete before returning, so the borrowed data cannot be invalidated.

---

## Quick Reference

| Synchronization Type | Use Case | Guard Type | Key Trait Bound |
|---|---|---|---|
| `Mutex<T>` | Exclusive access to shared data | `MutexGuard<T>` | `T: Send` |
| `RwLock<T>` | Read-heavy concurrent access | `RwLockReadGuard<T>` / `RwLockWriteGuard<T>` | `T: Send + Sync` |
| `mpsc::channel` | Thread communication | N/A | `T: Send` |

| | | | |
|---|---|---|---|
| `AtomicUsize` | Lock-free counters/flags | N/A | Inherently `Sync` |
| `thread::scope` | Borrow stack data in threads | N/A | Borrowed data outlives scope |

## Marker Traits Summary

```
Send: Type can be transferred to another thread (ownership moves)
Sync: Type can be shared between threads (&T is Send)

Relationship: T is Sync if and only if &T is Send
```

## Common Patterns

```
// Shared ownership + mutual exclusion
Arc<Mutex<T>>

// Shared ownership + reader-writer access
Arc<RwLock<T>>

// Thread-safe reference counting alone (immutable data)
Arc<T>

// Lock-free atomic counter
Arc<AtomicUsize>
```

## Memory Ordering (from strongest to weakest)

| Ordering | Guarantee |
|---|---|
| `SeqCst` | Total global order, all threads agree |
| `AcqRel` | Acquire on load, release on store |
| `Release` | Prior writes visible to acquiring threads |
| `Acquire` | Sees all writes from releasing threads |
| `Relaxed` | Only atomicity, no ordering guarantees |

# Advanced Rust: Ownership, Borrowing, and Encapsulation

## Introduction

Rust is often defined as "a language empowering everyone to build reliable and efficient software." While memory safety is frequently highlighted as Rust's main feature, this talk explores the broader implications of Rust's ownership system and how it enables powerful encapsulation patterns.

The core challenge Rust addresses comes from Mozilla's experience with Firefox (21 million lines of code, billions of deployments): most bugs aren't local to individual functions, but arise from how components interact with each other. This "action at a distance" problem - where changes in one component break functionality elsewhere - was an explicit design goal for Rust to solve.

## Fundamental Concepts

### Data and Functions

Rust fundamentally deals with two main constructs: - **Data structures**: structs (multiple fields) and enums (alternatives)
- **Functions**: the primary way programs work - everything is a function call

Unlike object-oriented languages, Rust has no classes, inheritance, or automatic virtual dispatch. When components communicate (like passing data between threads), it's done through function calls.

### Ownership

Every value in Rust has exactly one unique owner. This owner can: - Mutate the value - Destroy it - Pass ownership to other parts of the system

When ownership is lost or given up, the value is automatically dropped and cleaned up.

```rust
struct Point {
    x: i32,
    y: i32,
}

impl Drop for Point {
    fn drop(&mut self) {
        println!("Dropping point!");
    }
}

fn main() {
    let point = Point { x: 1, y: 2 };
    println!("{:?}", point);
    // Point is automatically dropped here when it goes out of scope
}
```

## What Can You Own?

Ownership in Rust models different types of resources:

1. **Plain data**: Numbers, simple structures in memory
2. **Heap allocations**: Vectors that allocate and manage memory
3. **Resources with lifecycle requirements**: File handles, mutex guards

```rust
use std::fs::File;

// Owning plain data
let point = Point { x: 1, y: 2 };

// Owning a resource (file handle)
let file = File::open("example.txt")?;
// When `file` goes out of scope, it's automatically closed
```

The key insight: Rust's ownership is about **resource safety**, not just memory safety. A `File` in Rust doesn't even have a `close()` method - closing happens automatically when you drop ownership.

## Borrowing

Values can be referenced through borrowing, where the owner promises not to touch the data while it's borrowed.

### Immutable References

```rust
fn takes_borrow(point: &Point) {
    println!("Point: ({}, {})", point.x, point.y);
}
```

Immutable references (`&T`): - Can be shared (multiple references can exist) - Guarantee no mutation is observable - Prevent the owner from removing the data from memory

**Mutable References**

```rust
fn modify_point(point: &mut Point) {
    point.x += 1;
}
```

Mutable references (`&mut T`): - Are unique (only one can exist at a time) - Allow mutation of the referenced data - Cannot alias with any other references

### Borrow Checking

The borrow checker uses region-based memory management, drawing regions where data lives and where it's borrowed:

```rust
let data = Point { x: 1, y: 2 };
let reference = &data;        // Borrow starts here
drop(data);                   // ❌ Error: data borrowed until...
println!("{:?}", reference);  // ...here
```

The checker ensures borrowing regions are within the data's lifetime regions.

## Advanced Patterns: Building a Mutex

Let's explore how Rust's concepts work together by implementing a mutex - a runtime expression of Rust's compile-time "unique access" guarantee.

### Using a Mutex

```rust
use std::sync::Mutex;

struct Counter {
    value: i32,
}

fn main() {
    let counter = Counter { value: 0 };
    let mutex = Mutex::new(counter); // Takes ownership

    manipulate_counter(&mutex);
}

fn manipulate_counter(mutex: &Mutex<Counter>) {
    let mut guard = mutex.lock().unwrap();
    guard.value += 1;
}
```

Notice the ownership transfer: the `Counter` has a very short lifetime (line 1-2), then the `Mutex` takes ownership and lives for the rest of the program.

## Interior Mutability

The mutex example reveals something interesting: we get mutable access (`&mut Counter`) from an immutable reference (`&Mutex<Counter>`). This is **interior mutability** - producing mutable access to immutable memory locations.

This isn't a hack - it's a core language feature that makes sense: 1. We borrow the mutex immutably (can't destroy it) 2. We call `lock()` to register our interest 3. We get a `MutexGuard` token representing unique access 4. The guard can produce a mutable reference to the inner data

## Implementing a Simple Mutex

```rust
use std::cell::UnsafeCell;

pub struct Mutex<T> {
    data: UnsafeCell<T>,
    // ... locking machinery
}

pub struct MutexGuard<'a, T> {
    mutex: &'a Mutex<T>,
}

impl<T> Mutex<T> {
    pub fn new(data: T) -> Self {
        Mutex {
            data: UnsafeCell::new(data),
        }
    }

    pub fn lock(&self) -> MutexGuard<T> {
        // Acquire lock from OS (e.g., pthread_mutex_lock)
        MutexGuard { mutex: self }
    }
}

impl<T> MutexGuard<'_, T> {
    pub fn borrow_mut(&self) -> &mut T {
        unsafe {
            let ptr = self.mutex.data.get();
            &mut *ptr
        }
    }
}

impl<T> Drop for MutexGuard<'_, T> {
    fn drop(&mut self) {
        // Unlock mutex (e.g., pthread_mutex_unlock)
    }
}
```

## The Role of Unsafe

The `unsafe` block doesn't disable language features - it enables six additional operations that could break language guarantees if misused. Here we use raw pointer dereferencing.

The key insight: the number of `unsafe` lines isn't important. What matters is the **argument** for why the unsafe code is valid. In our mutex:

1. We own the inner data through the `Mutex`
2. We have an immutable reference to the mutex (it can't be destroyed)

3. `lock()` returns a guard only when we've successfully acquired the lock
4. The guard is unique due to the locking mechanism
5. Therefore, the mutable reference we return is guaranteed to be unique

## Software Architecture Implications

### Ownership vs Borrowing as Contracts

**Ownership** is a contract forcing the caller to give up interest in data, putting the callee in charge. This is a tool for **decoupling**.

**Borrowing** is a contract where the caller gives access under conditions, but makes promises about keeping data available. This creates **strong coupling**.

```
// Three different contracts for writing to a file:

// 1. Takes ownership - maximum decoupling
fn submit_write_op(file: File, data: String) { /* ... */ }

// 2. Borrows file, owns data
fn write_to_file(file: &File, data: String) { /* ... */ }

// 3. Borrows both - tightest coupling
fn write_to_file_borrowed(file: &File, data: &str) { /* ... */ }
```

### Fighting the Borrow Checker

When developers "fight" the borrow checker, they're often fighting their software architecture. Borrowing works well for short-term access, but becomes problematic when multiple components express opinions about memory buffer management.

The solution is often to use ownership for communication between components, keeping borrowing within component boundaries.

## Key Takeaways

When coding in Rust, ask yourself:

1. **What resources do I manage?** Much more than just memory - files, network connections, locks, etc.

2. **What are their lifecycles?** Rust provides basic lifecycle management through ownership
3. **How can I encode invariants?** Use the type system to make invalid states unrepresentable
4. **How do I encapsulate complexity?** Hide internal management behind safe interfaces
5. **If unsafety is involved, what's my safety argument?** Document why unsafe code upholds Rust's guarantees

## The Power of Encapsulation

Rust's encapsulation allows hiding complex resource management behind simple interfaces. The distinction between "immutable parts you can't touch" (safety machinery) and "inner parts you can access" (actual data) is a powerful pattern for building safe abstractions.

Interior mutability and unsafe code aren't escape hatches - they're tools for building safe abstractions that hide complexity from users while maintaining Rust's safety guarantees.

# Advanced Rust Programming: Ownership, Borrowing, and Encapsulation

## Introduction

Rust is a programming language that has evolved significantly since its inception. Originally defined in 2015 as "a systems programming language that runs blazingly fast, prevents nearly all segfaults and guarantees thread safety," it was redefined in 2018 as "a language empowering everyone to build reliable and efficient software."

This guide explores advanced Rust concepts focusing on reliability through ownership, borrowing, and encapsulation—going beyond just memory safety to examine how Rust manages complex program interactions and prevents "action at distance" bugs where changes in one component unexpectedly break another.

## Core Language Philosophy

Rust fundamentally deals with two main constructs:

1. **Data** - Structures (multiple fields) and enums (alternatives)
2. **Functions** - Everything is a function call, including data conversion and component interaction

Unlike languages with classes and inheritance, Rust has no automatic virtual dispatch and follows a more functional approach to program design.

## Ownership: The Foundation

### Basic Ownership Rules

Every value in Rust has exactly one unique owner. This owner can: - Mutate the data - Destroy the data
- Pass ownership to other parts of the system

When ownership is lost or given up, the value is automatically dropped and cleaned up.

```rust
struct Point {
    x: i32,
    y: i32,
}

impl Drop for Point {
    fn drop(&mut self) {
        println!("Dropping point");
    }
}

fn main() {
    let point = Point { x: 1, y: 2 };
    println!("{:?}", point);
    // Point is automatically dropped here when it goes out of scope
}
```

## What Can You Own?

Ownership in Rust extends far beyond simple data:

- **Plain data**: Numbers, basic structures
- **Heap allocations**: Vectors, dynamically allocated memory
- **Resources with lifecycles**: File handles, network connections
- **Abstract privileges**: Mutex guards, exclusive access tokens

```rust
use std::fs::File;

// Owning a simple data structure
let point = Point { x: 1, y: 2 };

// Owning a file resource
let file = File::open("example.txt")?;
// When `file` goes out of scope, it's automatically closed
```

## Resource Management, Not Just Memory

Ownership is fundamentally about resource safety. A `File` in Rust doesn't expose its internal file descriptor because:

1. It maintains encapsulation of the resource lifecycle
2. It prevents interference with proper cleanup
3. It ensures resources are properly released

Notably, Rust's `File` has no `close()` method—the way to close it is to drop ownership.

## Borrowing: Controlled Access

Borrowing allows values to be referenced while the owner promises not to modify or destroy the data. There are two types of references:

### Immutable References (`&T`)

```
fn takes_borrow(point: &Point) {
    println!("Point: ({}, {})", point.x, point.y);
}
```

- Can be shared (multiple immutable references allowed)
- Guarantee: No mutation will be observed through these references
- Stronger than just "read-only"—nobody can mutate the data while borrowed

### Mutable References (`&mut T`)

```
fn modify_point(point: &mut Point) {
    point.x += 1;
    point.y += 1;
}
```

- Must be unique (only one mutable reference at a time)
- Cannot alias with other references
- Provides exclusive access for the duration of the borrow

### Borrow Checking

Rust uses region-based memory management to verify borrows:

```
fn main() {
    let data = Point { x: 1, y: 2 };     // data lives: line 1-4
    let reference = &data;                // borrow lives: line 2-4
    drop(data);                          // ❌ Error: trying to drop at line 3
    println!("{:?}", reference);         // reference used until line 4
}
```

The compiler draws regions and ensures borrowed data outlives all references to it.

## Ownership vs Borrowing in Software Architecture

The choice between ownership and borrowing has significant architectural implications:

### Ownership as Decoupling

```
fn submit_write_operation(file: File, data: String) {
    // Takes ownership - caller gives up control
    // Enables fire-and-forget patterns
    // Good for component boundaries
}
```

### Borrowing as Coupling

```
fn write_to_disk(file: &mut File, data: &str) -> std::io::Result<()> {
    // Borrows - creates tight coupling
    // Caller must maintain data and file
    // Good for controlled, short-term operations
}
```

**Key insight**: Fighting with the borrow checker often indicates architectural issues rather than language limitations.

## Interior Mutability and Encapsulation

Rust provides mechanisms to safely violate the borrowing rules when needed, through interior mutability patterns.

### Mutex: A Practical Example

A mutex demonstrates how Rust models runtime-enforced unique access:

```rust
use std::sync::Mutex;

struct Counter {
    value: i32,
}

fn main() {
    let data = Counter { value: 0 };
    let mutex = Mutex::new(data);  // Mutex takes ownership

    manipulate_counter(&mutex);    // Pass immutable reference to mutex
}

fn manipulate_counter(counter: &Mutex<Counter>) {
    let mut guard = counter.lock().unwrap();
    guard.value += 1;  // Mutable access through immutable reference!
}
```

## How Interior Mutability Works

The mutex provides:

1. **Encapsulation**: The inner data is hidden behind the mutex interface
2. **Runtime enforcement**: Locking ensures unique access at runtime
3. **Safe interface**: Interior mutability is contained within the mutex implementation

```rust
// Conceptual implementation
pub struct Mutex<T> {
    data: UnsafeCell<T>,  // Allows interior mutability
    // ... locking machinery
}

pub struct MutexGuard<'a, T> {
    mutex: &'a Mutex<T>,  // Tied to original mutex lifetime
}

impl<T> Mutex<T> {
    pub fn lock(&self) -> MutexGuard<T> {
        // Acquire lock from OS
        MutexGuard { mutex: self }
    }
}

impl<T> MutexGuard<'_, T> {
    pub fn borrow_mut(&mut self) -> &mut T {
        unsafe {
            // Safety: We hold the lock, so access is unique
            &mut *self.mutex.data.get()
        }
    }
}
```

## Unsafe Rust: Controlled Danger

The `unsafe` keyword doesn't disable safety checks—it enables additional capabilities that require manual verification:

1. Dereferencing raw pointers
2. Calling unsafe functions
3. Accessing mutable statics
4. Implementing unsafe traits
5. Reading from unions
6. Accessing fields of packed structs

### Safety Through Encapsulation

```rust
impl<T> MutexGuard<'_, T> {
    fn borrow_mut(&mut self) -> &mut T {
        unsafe {
            // Safety argument:
            // 1. We own the MutexGuard, proving we hold the lock
            // 2. MutexGuard cannot be cloned or copied
            // 3. Lock ensures exclusive access
            // 4. Lifetime ties reference to guard
            &mut *self.mutex.data.get()
        }
    }
}
```

The key insight: The amount of `unsafe` code isn't what matters—it's the strength of the safety argument and the encapsulation around it.

## Key Principles for Advanced Rust

When working with advanced Rust concepts, always ask:

1. **What resources do I manage?** Beyond memory, consider files, locks, network connections
2. **What are their lifecycles?** How are they acquired and released?
3. **How can I encode invariants?** Use the type system to prevent invalid states
4. **What is the relationship between resources?** Model dependencies through ownership
5. **How do I encapsulate complexity?** Hide internal management behind safe interfaces
6. **If using unsafe, what is my safety argument?** Document why the unsafe code is correct

## Conclusion

Advanced Rust programming is fundamentally about:

- **Ownership** for lifecycle management and component decoupling
- **Borrowing** for controlled access with clear contracts
- **Encapsulation** for hiding complexity and managing invariants
- **Interior mutability** for safe violations of borrowing rules
- **Unsafe code** for operations the compiler cannot verify

These features work together to prevent "action at distance" bugs and enable reliable software architecture. The strict nature of Rust's ownership system forces explicit thinking about resource

management and component boundaries, leading to more maintainable and reliable code.

# Advanced Rust Programming: Ownership, Borrowing, and Encapsulation

## Introduction

Rust is often described as "a language empowering everyone to build reliable and efficient software." While memory safety is frequently highlighted as Rust's primary feature, the language offers much more sophisticated capabilities for building robust systems through its ownership model, borrowing system, and encapsulation features.

This guide explores advanced Rust programming techniques, focusing on how Rust manages problems beyond just memory safety, and how these features enable better software architecture.

## What Problems Does Rust Tackle?

When Mozilla Research developed Rust, they identified key issues in complex codebases like Firefox (21 million lines of code):

- **Action at Distance**: Bugs that occur when you change something in one part of the code and it breaks something elsewhere
- **Component Interaction Problems**: Issues that arise not from individual functions, but from how components interact with each other
- **Evolution-Related Bugs**: Problems introduced when components evolve independently and move against each other

Rust was explicitly designed to tackle these architectural challenges.

## Rust's Core Architecture

Despite appearing complex, Rust fundamentally deals with two main constructs:

1. **Data**: Structures (things with multiple fields) and enums (alternatives)
2. **Functions**: The primary way programs work - everything in Rust is a function call

Rust is more functional than many realize: - No classes or inheritance - No automatic virtual dispatch - Component communication happens through function calls

# Ownership: The Foundation

## Basic Ownership Rules

Every value in Rust has exactly one unique owner:

```
struct Point {
    x: i32,
    y: i32,
}

impl Drop for Point {
    fn drop(&mut self) {
        println!("Dropping Point({}, {})", self.x, self.y);
    }
}

fn main() {
    let point = Point { x: 1, y: 2 };
    println!("{:?}", point);
    // Point is automatically dropped here
}
```

## What Can You Own?

Ownership in Rust extends beyond simple data:

1. **Plain data**: Numbers, structures in memory
2. **Heap allocations**: Vectors, strings
3. **Resources with lifecycle requirements**: File handles, network connections
4. **Privileges**: Access rights to shared data structures

```
use std::fs::File;

// Owning a file handle
let file = File::open("example.txt")?;
// When `file` goes out of scope, it's automatically closed
```

### Resource Safety, Not Just Memory Safety

Rust's ownership system is fundamentally about resource safety. The `File` type demonstrates this:

```
// File has no close() method!
// The way to close it is to drop it
let file = File::open("data.txt")?;
// File is automatically closed when dropped
```

This encapsulation is crucial - you cannot access the internal file descriptor, preventing interference with lifecycle management.

## Borrowing: Controlled Access

### Immutable References

```
fn takes_borrow(point: &Point) {
    println!("Point: ({}, {})", point.x, point.y);
}

let point = Point { x: 1, y: 2 };
takes_borrow(&point); // Borrow the point
// Point is still owned here
```

**Key guarantee**: Through an immutable reference, you cannot observe any mutations. The guarantee is stronger than "you can't mutate" - it's "nobody can mutate while you're watching."

### Mutable References

```
fn modify_point(point: &mut Point) {
    point.x += 1;
    point.y += 1;
}

let mut point = Point { x: 1, y: 2 };
modify_point(&mut point);
```

Mutable references are: - **Unique**: Only one can exist at a time - **Exclusive**: Cannot coexist with immutable references - **Inherently concurrency-safe**: By definition thread-safe

### Borrow Checking

Rust uses region-based memory management to track borrows:

```
let data = Point { x: 1, y: 2 };
let reference = &data;        // Borrow starts
println!("{:?}", reference);
drop(data);                   // ERROR: Can't drop while borrowed
println!("{:?}", reference);  // Borrow ends
```

The compiler draws regions and ensures borrowing regions stay within ownership regions.

## Ownership vs Borrowing: Architectural Implications

### Function Design Patterns

Consider these three approaches to the same functionality:

```
// 1. Takes ownership - decoupling
fn write_to_file_owned(file: File, data: String) {
    // Caller gives up all control
}

// 2. Borrows file, owns data - mixed approach
fn write_to_file_mixed(file: &File, data: String) {
    // Caller keeps file, gives up data
}

// 3. Borrows everything - tight coupling
fn write_to_file_borrowed(file: &File, data: &str) {
    // Caller retains all control
}
```

**Key insight**: - **Ownership = Decoupling**: Caller gives up control, callee manages lifecycle - **Borrowing = Coupling**: Caller makes promises about data lifetime

### Architectural Guidance

- Use ownership to pass data between components with clean separation
- Use borrowing for short-term access within component boundaries
- Fighting the borrow checker often indicates architectural issues

## Interior Mutability: Controlled Mutation

Sometimes you need mutable access through immutable references. This is called interior mutability:

```
use std::sync::Mutex;

let counter = Mutex::new(0);

fn increment(mutex: &Mutex<i32>) {
    let mut guard = mutex.lock().unwrap();
    *guard += 1;
}
```

### How Mutex Works

The mutex demonstrates sophisticated encapsulation:

```
// Simplified mutex implementation
struct Mutex<T> {
    data: UnsafeCell<T>,
    // locking machinery
}

struct MutexGuard<'a, T> {
    mutex: &'a Mutex<T>,
}

impl<T> Mutex<T> {
    fn lock(&self) -> MutexGuard<T> {
        // Acquire lock from OS
        MutexGuard { mutex: self }
    }
}

impl<'a, T> MutexGuard<'a, T> {
    fn borrow_mut(&mut self) -> &mut T {
        unsafe { &mut *self.mutex.data.get() }
    }
}

impl<'a, T> Drop for MutexGuard<'a, T> {
    fn drop(&mut self) {
        // Unlock the mutex
    }
}
```

The pattern: 1. Immutable access to the mutex itself 2. Locking produces a unique guard (token) 3. Guard provides mutable access to inner data 4. Dropping guard automatically unlocks

# Unsafe Rust: Managing the Unmanageable

## When Unsafe is Necessary

Unsafe Rust isn't an escape hatch - it's a core language feature for situations the compiler cannot verify:

```
impl<'a, T> MutexGuard<'a, T> {
    fn borrow_mut(&mut self) -> &mut T {
        unsafe {
            // SAFETY: We hold the lock, so we have exclusive access
            &mut *self.mutex.data.get()
        }
    }
}
```

## Unsafe is About Encapsulation

The goal is to: 1. Isolate unsafe code to small, reviewable modules 2. Build safe interfaces around unsafe operations 3. Document safety invariants clearly 4. Leverage existing, vetted unsafe implementations when possible

**Important**: The number of lines of unsafe code doesn't matter - the complexity of the safety argument does.

# Key Design Principles

## Questions to Ask When Designing Rust Code

1. **What resources do I manage?** (Beyond just memory)
2. **What are their lifecycles?**
3. **How can I encode invariants in the type system?**
4. **What relationships exist between resources?**
5. **How can I encapsulate complexity?**

6. **If using unsafe, what's my safety argument?**

### Encapsulation Strategy

- Hide internal management details
- Provide safe interfaces
- Use ownership to model resource lifecycles
- Leverage the type system to prevent invalid states
- Compartmentalize unsafe code

## Conclusion

Rust's advanced features work together to enable reliable software architecture:

- **Ownership** manages resource lifecycles automatically
- **Borrowing** provides controlled access with compile-time verification
- **Interior mutability** enables sophisticated concurrency patterns
- **Unsafe** allows system-level programming within safe boundaries
- **Strong encapsulation** hides complexity behind safe interfaces

The key insight is that these aren't just memory management features - they're tools for building better software architecture that prevents bugs through design rather than testing.

Understanding these concepts deeply allows you to leverage Rust's full power for building reliable, efficient systems that scale with complexity.

# Mastering Rust Through ripgrep: A Deep Dive into Production Systems Programming

## Course Introduction

Welcome to an in-depth exploration of one of the most respected Rust codebases in the ecosystem. ripgrep isn't just a fast grep replacement—it's a masterclass in systems programming, demonstrating how to build performant, maintainable, and user-friendly command-line tools in Rust.

Created by Andrew Gallick (BurntSushi), ripgrep has become a reference implementation for many Rust patterns. When Rust developers want to understand how to properly implement the builder pattern, design trait hierarchies, or structure a multi-crate workspace, they often turn to ripgrep. By studying this codebase, you're not just learning one tool—you're learning idioms and patterns that will serve you across your entire Rust journey.

What makes ripgrep particularly valuable for learning? It sits at the intersection of several challenging domains: high-performance text processing, cross-platform compatibility, parallel execution, and complex configuration management. Yet the code remains remarkably readable, with clear abstractions and thoughtful organization. This is production Rust at its finest.

Throughout this course, you'll discover how ripgrep achieves grep-like functionality while consistently outperforming traditional tools. You'll see how careful API design makes complex functionality accessible, how the type system prevents entire categories of bugs, and how Rust's ownership model enables both safety and performance.

---

## Prerequisites

This course assumes you have a solid foundation in programming and basic familiarity with Rust. Specifically, you should be comfortable with:

### Rust Fundamentals

- **Ownership and borrowing**: You understand why Rust has these concepts and can work with references without fighting the borrow checker constantly

- **Structs, enums, and pattern matching**: These are Rust's bread and butter for data modeling
- **Traits**: You know how to define and implement traits, even if advanced patterns like trait objects are still fuzzy
- **Error handling**: You're familiar with `Result`, `Option`, and the `?` operator
- **Modules and crates**: You understand how Rust organizes code into modules and how dependencies work

### General Programming Knowledge

- **Regular expressions**: You should understand basic regex syntax and concepts
- **Command-line tools**: Familiarity with grep, find, or similar tools helps contextualize the problem domain
- **Concurrent programming concepts**: Understanding threads and parallelism at a conceptual level is helpful, though we'll explain ripgrep's specific approaches

### Nice to Have (But Not Required)

- Experience with the builder pattern in any language
- Familiarity with memory-mapped files
- Understanding of Unicode text processing challenges

If you're still learning Rust fundamentals, consider working through the official Rust Book first. This course will be here when you're ready to see how these concepts come together in production code.

## Learning Objectives

By completing this course, you will:

### Understand Production Rust Architecture

- Recognize how to structure a multi-crate Rust workspace where each crate has a focused responsibility
- See how the facade pattern creates clean public APIs that hide internal complexity
- Learn when and why to split functionality across multiple crates

### Master the Builder Pattern Ecosystem

- Implement builders that validate configuration at construction time
- Understand the trade-offs between eager and lazy validation
- See how builders compose with other builders to configure complex systems

### Develop Trait-Based Abstraction Skills

- Design traits that define clear contracts while allowing diverse implementations
- Understand how ripgrep's `Matcher` and `Sink` traits enable extensibility
- Learn when trait objects are appropriate versus generic parameters

### Gain Performance Engineering Intuition

- Recognize patterns for memory-efficient processing of large inputs
- Understand when to use memory mapping versus buffered reading
- See how literal optimization extracts fast paths from complex patterns
- Learn how parallel iteration is implemented with work-stealing

### Appreciate Systems Programming Patterns

- Handle platform differences gracefully with conditional compilation
- Manage external processes with proper timeout and cleanup behavior
- Work with terminal capabilities and ANSI color codes
- Process compressed files transparently

### Navigate Real-World Complexity

- See how configuration flows from command-line flags through to execution
- Understand the layers involved in a seemingly simple "search for pattern in files" operation
- Appreciate the edge cases that production tools must handle

---

# How to Use This Course

This course is structured for deep, engaged learning. Each lesson consists of two companion documents that work together:

## Lecture Documents

These are the documents you're reading now—prose explanations written in a lecture style. They: - Introduce concepts before you see the code - Explain the "why" behind design decisions - Connect implementation details to broader Rust principles - Highlight patterns you'll encounter repeatedly - Discuss trade-offs and alternatives that were considered

Lecture documents work well for reading or listening (if converted to audio). They provide the conceptual framework that makes the code meaningful.

## Code Companion Documents

Each lecture has an accompanying code companion that contains: - Annotated source code excerpts - Inline comments explaining specific lines - Cross-references to related code in other lessons - Type signatures and their implications

The code companions are reference documents—you'll flip to them when the lecture mentions specific implementations.

## Suggested Approach

**For each lesson:** 1. Read the lecture document first, building mental models 2. Open the code companion alongside the actual source code 3. Trace through the code, matching it to the lecture's explanations 4. Experiment: modify the code, run tests, see what breaks

**Pacing recommendations:** - **Intensive study**: 2-3 lessons per day, with hands-on coding - **Moderate pace**: 1 lesson per day, with time for exploration - **Deep dive**: 2-3 lessons per week, implementing similar patterns yourself

**Working with the actual codebase:**

```
git clone https://github.com/BurntSushi/ripgrep
cd ripgrep
cargo build --release
```

We strongly recommend having the codebase open as you study. Reading code in your editor, with full IDE support for jumping to definitions, is far more effective than reading excerpts alone.

---

## Learning Threads Overview

Several conceptual threads weave through multiple lessons. Recognizing these threads helps you see how patterns recur and evolve throughout the codebase.

### The Builder Pattern Ecosystem

*Appears in: Lessons 7, 11, 14, 20, 25, 28*

ripgrep uses the builder pattern extensively—but not in a rote, mechanical way. You'll see builders that: - Validate configuration during building (fail-fast approach) - Allow incremental configuration with sensible defaults - Compose with other builders to configure nested systems - Use generic parameters to track configuration state at compile time

By following this thread, you'll develop intuition for when and how to apply builder patterns in your own code.

### Trait-Based Extensibility

*Appears in: Lessons 3, 5, 8, 17*

The `Matcher` and `Sink` traits are the architectural backbone of ripgrep's library ecosystem. This thread explores: - How traits define contracts between components - When to use trait objects versus generic parameters - How associated types carry additional type information - The relationship between trait design and testability

### Memory-Efficient Stream Processing

*Appears in: Lessons 12, 15, 16*

Processing potentially gigabyte-sized files with bounded memory is a fundamental challenge. This thread covers: - Line buffer design for incremental processing - When memory mapping provides benefits (and when it doesn't) - How context lines complicate streaming (and how to handle it) - Balancing memory usage against performance

### Configuration Flow

*Appears in: Lessons 33, 34, 35, 36*

Command-line tools must transform user input into program behavior. This thread follows configuration from: - Raw command-line arguments - Through low-level parsing - Into validated high-level structures - Finally becoming configured components

### Cross-Platform Considerations

*Appears in: Lessons 29, 30, 31*

Real-world tools must work across operating systems. This thread examines: - Conditional compilation for platform-specific code - Abstracting over platform differences - Handling external processes portably - Terminal capability detection

---

## Course Outline

### Part 1: The Library Ecosystem (Lessons 1-2)

**Lesson 1: The Facade Pattern** — `crates/grep/src/lib.rs` How ripgrep organizes its functionality across multiple crates while presenting a unified API. You'll see how the `grep` crate serves as a facade, re-exporting types from specialized crates without exposing internal organization.

**Lesson 2: Minimal Working Example** — `crates/grep/examples/simplegrep.rs` A complete, working grep implementation in approximately 50 lines. This lesson shows the payoff of ripgrep's API design—how all the complexity we'll study enables remarkably simple client code.

### Part 2: The Matcher Abstraction (Lessons 3-5)

**Lesson 3: The Matcher Trait** — `crates/matcher/src/lib.rs` The central abstraction that defines what it means to "match" text. You'll learn how this trait enables ripgrep to support different regex engines while maintaining a consistent interface.

**Lesson 4: Capture Group Interpolation** — `crates/matcher/src/interpolate.rs` How `$1`, `$name`, and other replacement patterns work at the byte level. This lesson demonstrates careful parsing and Rust's approach to string manipulation.

**Lesson 5: The Sink Trait** — `crates/searcher/src/sink.rs` The consumer pattern for search results. You'll see how this trait handles context lines, binary detection, and result filtering through a callback-based design.

## Part 3: Regex Integration (Lessons 6-10)

**Lesson 6: Regex Crate Entry** — `crates/regex/src/lib.rs` A brief orientation to how the regex matcher crate is organized, setting the stage for deeper exploration.

**Lesson 7: Regex Configuration** — `crates/regex/src/config.rs` Builder pattern for regex options including case sensitivity, word boundaries, and multi-line mode. See how configuration is validated and applied.

**Lesson 8: Implementing Matcher** — `crates/regex/src/matcher.rs` The concrete implementation of the `Matcher` trait for Rust's regex crate. Learn about thread-safe regex caching and how abstract interfaces meet concrete implementations.

**Lesson 9: Literal Optimization** — `crates/regex/src/literal.rs` How ripgrep extracts literal string prefixes from regex patterns to enable fast-path searching. This is a key performance optimization that makes ripgrep fast.

**Lesson 10: AST Manipulation** — `crates/regex/src/ast.rs` Parsing and transforming regex syntax trees. You'll see how ripgrep modifies regex patterns (for case-insensitive matching, for example) at the AST level.

## Part 4: The Search Engine (Lessons 11-16)

**Lesson 11: Searcher Overview** — `crates/searcher/src/lib.rs` The public API for searching. Configuration options for encoding, binary detection, line terminators, and context lines.

**Lesson 12: Efficient Line Reading** — `crates/searcher/src/line_buffer.rs` Memory-efficient buffering for line-oriented processing. You'll understand how ripgrep processes files incrementally without loading them entirely into memory.

**Lesson 13: Line Handling Utilities** — `crates/searcher/src/lines.rs` Finding line boundaries, counting lines, and handling different line terminators. The detailed work that underlies "line-oriented" searching.

**Lesson 14: Searcher Core** — `crates/searcher/src/searcher/mod.rs` The main `Searcher` and `SearcherBuilder`. How configuration options translate into search behavior.

**Lesson 15: Search Algorithm** — `crates/searcher/src/searcher/core.rs` The actual search loop. Context line handling, match iteration, and coordinating with the `Sink` trait for output.

**Lesson 16: Memory Mapping** — `crates/searcher/src/searcher/mmap.rs` When and how to use memory-mapped files. You'll learn the heuristics ripgrep uses to decide between mmap and buffered reading.

## Part 5: Output Formatting (Lessons 17-22)

**Lesson 17: Printer Architecture** — `crates/printer/src/lib.rs` Overview of the three output modes: Standard (human-readable), JSON (machine-readable), and Summary (counts only).

**Lesson 18: Terminal Colors** — `crates/printer/src/color.rs` ANSI color handling with the termcolor crate. User-configurable color schemes and terminal capability detection.

**Lesson 19: Printer Utilities** — `crates/printer/src/util.rs` Shared formatting utilities: line numbers, byte offsets, path formatting, and separator handling.

**Lesson 20: Standard Output** — `crates/printer/src/standard.rs` The main human-readable output format. Context lines, column numbers, and matching fragment highlighting.

**Lesson 21: JSON Output** — `crates/printer/src/json.rs` Machine-readable JSON output using streaming JSON generation. See how ripgrep provides structured data for tooling integration.

**Lesson 22: Summary Output** — `crates/printer/src/summary.rs` Count-only and path-only modes. Aggregation strategies for different summary styles.

## Part 6: File Discovery (Lessons 23-28)

**Lesson 23: Glob Pattern Matching** — `crates/globset/src/lib.rs` High-level glob API. `GlobSet` for efficiently matching against many patterns simultaneously.

**Lesson 24: Glob Implementation** — `crates/globset/src/glob.rs` Glob-to-regex compilation. Special cases for performance (`*.rs` is faster than `**.rs`) and correctness.

**Lesson 25: Ignore Crate Overview** — `crates/ignore/src/lib.rs` The file filtering ecosystem. How `DirEntry`, `WalkBuilder`, and ignore patterns work together.

**Lesson 26: Gitignore Parsing** — `crates/ignore/src/gitignore.rs` Full `.gitignore` implementation including negation patterns, directory-only patterns, and nested gitignore files.

**Lesson 27: File Type Definitions** — `crates/ignore/src/types.rs` Mapping file extensions and names to types. How `--type rust` knows which files to include.

**Lesson 28: Parallel Directory Walking** — `crates/ignore/src/walk.rs` The parallel walker with work-stealing. Skip logic, cycle detection, and coordination between walker threads.

### Part 7: CLI Infrastructure (Lessons 29-31)

**Lesson 29: CLI Utilities Overview** — `crates/cli/src/lib.rs` Shared CLI utilities used across ripgrep. Hostname detection, tty handling, and standard stream management.

**Lesson 30: Decompression Support** — `crates/cli/src/decompress.rs` Searching compressed files transparently. Process spawning for external decompression tools.

**Lesson 31: Process Management** — `crates/cli/src/process.rs` Running external commands with timeouts. Graceful shutdown, signal handling, and resource cleanup.

### Part 8: The Application (Lessons 32-37)

**Lesson 32: Application Entry Point** — `crates/core/main.rs` The `rg` binary. Error handling strategies, logging setup, and the structure of the main function.

**Lesson 33: Flag System Architecture** — `crates/core/flags/mod.rs` How flags are defined, documented, parsed, and converted. The metadata system that generates help text.

**Lesson 34: Low-Level Arguments** — `crates/core/flags/lowargs.rs` Direct CLI parsing with the lexopt crate. Argument normalization and early validation.

**Lesson 35: High-Level Arguments** — `crates/core/flags/hiargs.rs` Semantic validation and conversion. Creating fully-configured search components from user input.

**Lesson 36: Search Orchestration** — `crates/core/search.rs` Tying everything together. Parallel versus sequential search, result aggregation, and exit code handling.

**Lesson 37: Haystack Discovery** — `crates/core/haystack.rs` Finding files to search. Stdin handling, explicit path arguments, and integration with the walker.

---

## Next Steps

After completing this course, you'll have a deep understanding of ripgrep's architecture and the patterns that make it successful. Here are paths for continued learning:

### Extend ripgrep

- Implement a new output format (perhaps SARIF for static analysis tools)
- Add a new matcher backend for a different regex engine
- Create a new file type definition for languages you use

### Apply These Patterns

- Build your own CLI tool using ripgrep's architectural patterns
- Implement a builder-heavy API for a library you're creating
- Design trait hierarchies for extensible systems

### Explore Related Codebases

- **BurntSushi's other projects**: The `regex`, `memchr`, and `aho-corasick` crates demonstrate low-level optimization techniques
- **fd**: A fast `find` replacement that shares some patterns with ripgrep
- **tokio**: For async Rust patterns at a similar level of sophistication

### Contribute

The best way to solidify understanding is to contribute. ripgrep's issue tracker has discussions about potential improvements, and even documentation contributions help.

---

## Ready to Begin?

Turn to **Lesson 1: The Facade Pattern** to start your journey. You'll see how ripgrep presents a unified interface to its users while maintaining clean internal organization. It's an elegant introduction to the architectural thinking that pervades this codebase.

Remember: take your time, experiment with the code, and don't hesitate to re-read sections. Production codebases reward patient study. Every lesson you complete builds understanding that compounds in later lessons.

Let's dive in.

# Inside the Rust Compiler: From Source Code to Binary

This comprehensive guide explores the internal workings of the Rust compiler (`rustc`), from parsing source code to generating binaries. We'll examine the various intermediate representations (IRs) and learn how to interact with the compiler programmatically.

## What is Rust?

Rust is defined by the Rust Language team as "a language empowering everyone to build reliable and efficient software." Key characteristics include:

- **Systems-level control**: Explicit memory allocation control
- **Efficiency**: No garbage collector, compiles to small binaries for embedded systems
- **Memory safety**: Prevents buffer overflows, use-after-free, and dangling pointers
- **Thread safety**: Data race freedom in concurrent code
- **Compile-time guarantees**: All safety checks happen at compile time

### Borrowing and Lifetime Semantics

Rust's safety guarantees come from its borrowing rules: - **Shared data cannot be mutated**: Multiple immutable references are allowed - **Mutable data cannot be aliased**: Only one mutable reference at a time - **Lifetimes ensure validity**: References cannot outlive the data they point to

### The `unsafe` Keyword

The `unsafe` keyword allows you to: - Directly manipulate memory and raw pointers - Call unsafe functions - Access mutable static variables - Implement unsafe traits

When used appropriately (like in the standard library), `unsafe` enables powerful abstractions while maintaining safety at higher levels.

## The Rust Compiler Architecture

The Rust compiler (`rustc`) transforms source code through multiple intermediate representations before generating a binary:

```
Rust Source Code
      ↓
Abstract Syntax Tree (AST)
      ↓
High-level IR (HIR)
      ↓
Typed HIR (THIR)
      ↓
Mid-level IR (MIR)
      ↓
Code Generation (LLVM/GCC/Cranelift)
      ↓
Target Binary
```

## Exploring the Rust Compiler Repository

The main Rust compiler repository contains several important directories:

- **`compiler/`**: Core compiler code organized into multiple crates
- **`library/`**: Standard library and core types
- **`src/`**: Bootstrap and build scripts
- **`tests/`**: Comprehensive test suite

### Building the Compiler

To build the Rust compiler from source:

```
# Initial setup
./x.py setup
# Choose option B for compiler development

# Build the compiler
./x.py build

# Incremental builds (much faster)
./x.py build --keep-stage 1
```

The build process creates multiple compiler stages due to bootstrapping: - **Stage 0**: Minimal compiler built with existing Rust toolchain - **Stage 1**: Full compiler built with Stage 0 - **Stage 2**: Self-hosted compiler (Stage 1 building itself)

# Intermediate Representations in Detail

## Abstract Syntax Tree (AST)

The AST is created through lexing and parsing:

```
// Source code
fn main() {
    println!("Hello, world!");
}
```

View the AST using:

```
rustc -Z unpretty=ast hello.rs
```

The AST represents the program structure as a tree, with nodes for functions, expressions, and statements.

## High-level IR (HIR)

HIR is created by lowering and desugaring the AST: - Multiple loop constructs (`for`, `while`, `loop`) are normalized to `loop` - Control flow is simplified to `match` statements - The prelude is made explicit

```
# View HIR
rustc -Z unpretty=hir hello.rs

# View HIR tree structure
rustc -Z unpretty=hir-tree hello.rs
```

At the HIR level, the compiler performs: - **Type inference**: Determining types from context - **Trait solving**: Ensuring generic constraints can be satisfied - **Type checking**: Verifying type compatibility

Example of trait bounds:

```
fn log_elements<T: std::fmt::Display>(vec: Vec<T>) {
    for (i, element) in vec.iter().enumerate() {
        println!("{}: {}", i, element);
    }
}
```

### Typed HIR (THIR)

THIR contains fully elaborated types and is used for: - Unsafe code analysis - Additional type-related checks

```
rustc -Z unpretty=thir-tree hello.rs
```

### Mid-level IR (MIR)

MIR transforms the tree structure into a Control Flow Graph (CFG):

```
# View MIR (recommended flags)
rustc -Z unpretty=mir -Z mir-opt-level=0 hello.rs
```

MIR represents programs as: - **Places in memory** instead of variables - **Basic blocks** containing statements - **Terminators** that branch between blocks

Example MIR structure:

```
fn main() -> () {
    let mut _0: ();
    let _1: ();

    bb0: {
        _1 = const "Hello, world!";
        _2 = &_1;
        _3 = std::io::_print(_2);
        goto -> bb1;
    }

    bb1: {
        return;
    }
}
```

At the MIR level, the compiler performs: - **Borrow checking**: Ensuring memory safety rules - **Drop elaboration**: Inserting cleanup code - **Lifetime analysis**: Validating reference lifetimes

## Interacting with the Compiler Programmatically

You can write Rust programs that call the compiler internally using the `rustc_driver` crate:

## Basic Setup

```rust
#![feature(rustc_private)]

extern crate rustc_driver;
extern crate rustc_interface;

use rustc_driver::Compilation;
use rustc_interface::interface;

struct MyCallbacks;

impl rustc_driver::Callbacks for MyCallbacks {
    fn after_analysis<'tcx>(
        &mut self,
        compiler: &interface::Compiler,
        queries: &'tcx rustc_interface::Queries<'tcx>,
    ) -> Compilation {
        queries.global_ctxt().unwrap().peek_mut().enter(|tcx| {
            // Access compiler internals here
            println!("Analyzing crate: {}",
tcx.crate_name(rustc_span::def_id::LOCAL_CRATE));
        });
        Compilation::Continue
    }
}

fn main() {
    let args: Vec<String> = std::env::args().collect();
    rustc_driver::RunCompiler::new(&args, &mut MyCallbacks).run().unwrap();
}
```

## Callback Points

The compiler provides three main callback points:

1. `after_crate_root_parsing`: After creating the initial AST
2. `after_expansion`: After macro expansion and name resolution
3. `after_analysis`: After all type checking and borrow checking

## Accessing Compiler Internals

With a typing context (`TyCtxt`), you can call many internal compiler functions:

```
// Get all function bodies in the crate
let body_owners = tcx.hir().body_owners().collect::<Vec<_>>();

// Pretty print MIR
for def_id in body_owners {
    rustc_middle::mir::pretty::write_mir_pretty(tcx, Some(def_id), &mut
std::io::stdout()).unwrap();
}
```

## Code Generation Backends

The final step transforms MIR into target code. Rust supports multiple backends:

- **LLVM** (default): Mature, highly optimizing
- **GCC**: Alternative with different optimization characteristics
- **Cranelift**: Faster compilation, less optimization

### Custom Code Generation

You can implement custom code generation by implementing the `CodegenBackend` trait:

```
impl CodegenBackend for MyBackend {
    fn codegen_crate(&self, tcx: TyCtxt, metadata: EncodedMetadata,
need_metadata_module: bool) -> Box<dyn Any> {
        // Custom code generation logic
    }

    // ... other required methods
}
```

This enables targeting custom architectures or generating specialized output formats.

# Debugging and Optimization

### Compiler Flags for Exploration

Useful flags for examining compiler internals:

```
# List all unstable options
rustc -Z help

# Common debugging flags
rustc -Z unpretty=ast          # Show AST
rustc -Z unpretty=hir          # Show HIR
rustc -Z unpretty=mir          # Show MIR
rustc --emit=llvm-ir           # Show LLVM IR
rustc --emit=asm               # Show assembly
```

### Development Workflow

When working on the compiler:

1. Use the compiler profile: `./x.py setup` → option B
2. Make changes to compiler source
3. Rebuild incrementally: `./x.py build --keep-stage 1 library`
4. Test changes with the stage 1 compiler

This workflow reduces rebuild times from minutes to seconds.

## Practical Applications

Understanding compiler internals enables:

- **Static analysis tools**: Analyze code without executing it
- **Custom lints**: Detect project-specific issues
- **Code generation**: Generate code from high-level descriptions
- **Verification tools**: Prove program properties
- **Educational tools**: Understand how language features work

## Conclusion

The Rust compiler is a sophisticated piece of software that transforms high-level Rust code through multiple intermediate representations, each optimized for different types of analysis. By understanding these internals and learning to interact with them programmatically, you can build powerful tools that leverage the compiler's deep understanding of Rust programs.

The compiler's architecture demonstrates the careful balance between providing high-level language features and maintaining the performance and safety guarantees that make Rust unique in the systems programming landscape.

# Rust Allocation from First Principles

A deep dive into how Rust manages memory, from raw bytes to high-level collections.

## Overview

This guide takes you through Rust's allocation system by reading the actual standard library source code. Rather than learning abstractions first, we start at the bottom — how memory is requested from the operating system — and build up to the smart pointers and collections you use every day.

## Prerequisites

- Basic Rust syntax (variables, functions, structs, enums)
- Comfort with the idea of pointers (even if not with raw pointer manipulation)
- Curiosity about how things work under the hood

## Chapters

### Chapter 1: Memory Layout and Alignment

Before allocating memory, Rust must know *what* to allocate. The `Layout` type captures two pieces of information: how many bytes, and what address boundaries the memory must respect.

**Key concepts:** - Memory alignment and why CPUs care - The Layout type and its invariants - Struct padding and field ordering - The isize::MAX size limit

### Chapter 2: The Allocator Traits

With Layout defining requirements, we need interfaces for actually getting memory. Rust provides two traits: the stable `GlobalAlloc` and the future-facing `Allocator`.

**Key concepts:** - GlobalAlloc: simple, stable, production-ready - Allocator: sophisticated, handles ZSTs, returns fat pointers - Safety contracts and undefined behavior - The Unix implementation calling libc

### Chapter 3: Box — Owned Heap Allocation

`Box<T>` is Rust's simplest smart pointer: a single heap allocation with ownership. It's the foundation for understanding how Rust combines allocation with the ownership system.

**Key concepts:** - Box structure: just a pointer (plus allocator) - The allocation path: from Box::new to malloc - Drop behavior: content destructors then deallocation - Zero-sized types and dangling pointers - Raw pointer escape hatches for FFI

### Chapter 4: Vec — Dynamic Arrays

`Vec<T>` adds dynamic sizing: the ability to grow and shrink at runtime. This introduces capacity management, growth strategies, and reallocation.

**Key concepts:** - The (ptr, len, cap) triplet - RawVec: the allocation engine - Amortized O(1) push via doubling - Reallocation: in-place vs allocate-copy-free - ZST handling with infinite capacity

## Source Code References

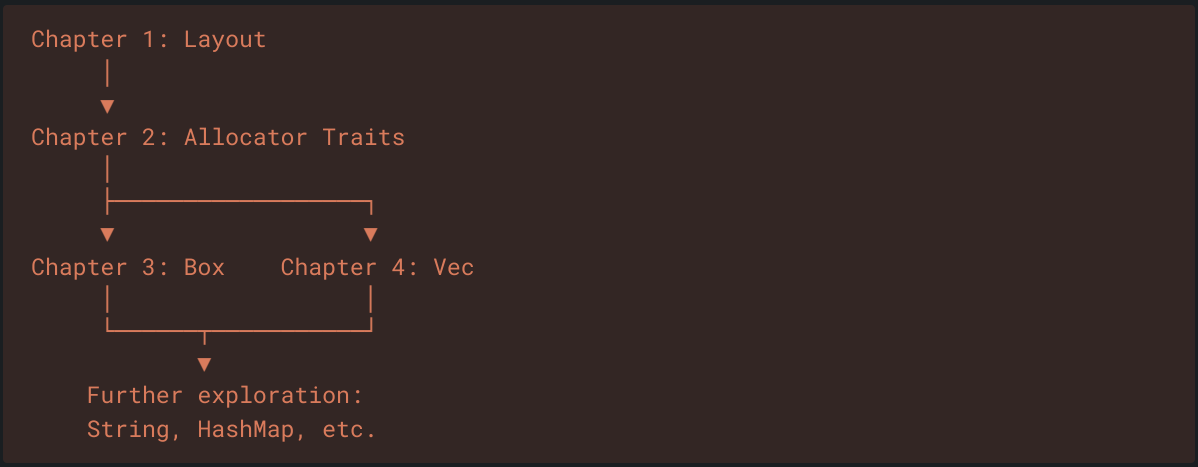
All content is based on the Rust standard library source:

| Topic | Source File |
|---|---|
| Layout | `library/core/src/alloc/layout.rs` |
| GlobalAlloc | `library/core/src/alloc/global.rs` |
| Allocator | `library/core/src/alloc/mod.rs` |
| Unix allocator | `library/std/src/sys/alloc/unix.rs` |
| Box | `library/alloc/src/boxed.rs` |
| Vec | `library/alloc/src/vec/mod.rs` |
| RawVec | `library/alloc/src/raw_vec.rs` |

## Companion Code

The `box_exploration.rs` file contains runnable examples demonstrating: - Box memory layout inspection - Layout calculations for various types - Zero-sized type behavior - Drop order visualization - Raw pointer round-trips - Manual allocation/deallocation

## Learning Path




## Key Insights

1. **Allocation is two questions**: How many bytes? What alignment?
2. **Traits abstract the allocator**: Your code doesn't care if it's malloc, jemalloc, or a custom arena.
3. **Zero-sized types are special**: They never allocate but maintain valid (dangling) pointers.
4. **Drop is two-phase**: Destructors first, then memory deallocation.
5. **Growth strategies matter**: Doubling gives O(1) amortized push; naive growth gives $O(n^2)$.
6. **Safety contracts are manual**: The compiler can't verify allocator correctness — you must uphold invariants.

## Next Steps

After completing these chapters:

- **Read the Rustonomicon**: Deeper unsafe Rust coverage

- **Implement your own allocator**: Apply what you've learned
- **Study other collections**: HashMap, BTreeMap, VecDeque
- **Explore String**: It's Vec with UTF-8 validation

---

*"Your performance intuition is useless. Run perf."*
— Rust standard library source code

# Chapter 1: Memory Layout and Alignment

> *"Your performance intuition is useless. Run perf."*
> — Comment in Rust's layout.rs

## Introduction

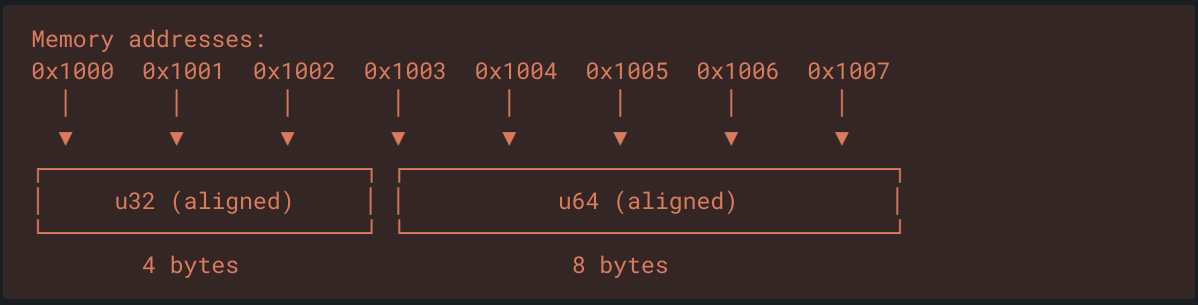
Before Rust can allocate memory, it needs to answer two fundamental questions: 1. **How many bytes do I need?** 2. **What address boundaries must the memory respect?**

The `Layout` type encapsulates these requirements. It's the foundation upon which all of Rust's allocation machinery is built.

---

## 1.1 What is Memory Alignment?

At the hardware level, CPUs access memory most efficiently when data sits at addresses that are multiples of the data's size.




A 32-bit integer (4 bytes) wants addresses like `0x1000`, `0x1004`, `0x1008` — addresses divisible by 4. A 64-bit value wants addresses divisible by 8.

### Why Does Alignment Matter?

**Hardware efficiency:** On many architectures, misaligned access requires multiple memory bus cycles instead of one. On x86, it's slower. On older ARM, it causes a hardware fault and crashes your program.

**Atomics:** Atomic operations often require alignment to work correctly. A misaligned atomic might not be atomic at all.

**SIMD:** Vector instructions typically require 16-byte or 32-byte alignment.

---

## 1.2 The Layout Type

Layout lives in `core::alloc` and has exactly two fields:

```
pub struct Layout {
    size: usize,      // How many bytes
    align: Alignment, // What address boundaries (power of 2)
}
```

That's it. Everything else is methods for constructing, validating, and manipulating these two numbers.

### The Core Invariant

This is critical and worth memorizing:

> **"The size, when rounded up to the nearest multiple of align, does not overflow isize."**

Why `isize` and not `usize`? Because pointer arithmetic in Rust uses signed offsets. The `ptr.offset(i)` method takes an `isize`. If your allocation exceeds `isize::MAX`, pointer arithmetic could overflow, causing undefined behavior.

The maximum size for a given alignment is calculated as:

```
const fn max_size_for_align(align: Alignment) -> usize {
    // max_size = (isize::MAX + 1) - align
    unsafe { unchecked_sub(isize::MAX as usize + 1, align.as_usize()) }
}
```

On a 64-bit system with 8-byte alignment, the maximum allocation is approximately **8 exabytes** (8,589,934,591 GB). You won't hit this limit.

---

## 1.3 Creating Layouts

### Safe Constructor

```
pub const fn from_size_align(size: usize, align: usize) -> Result<Self,
LayoutError>
```

This validates three things: 1. `align` must be non-zero 2. `align` must be a power of two 3. `size` must not exceed `max_size_for_align`

### Type-Based Constructors

```
// For any sized type - computed at compile time
let layout = Layout::new::<i32>();  // size=4, align=4

// For dynamically-sized types (slices, trait objects)
let layout = Layout::for_value(&some_slice);

// For arrays
let layout = Layout::array::<u32>(100)?;  // 100 u32s
```

### The Unsafe Escape Hatch

```
pub const unsafe fn from_size_align_unchecked(size: usize, align: usize) -> Self
```

Skips validation. In debug builds, it still asserts. In release builds, it trusts you completely. Use only when you can prove the invariants hold.

---

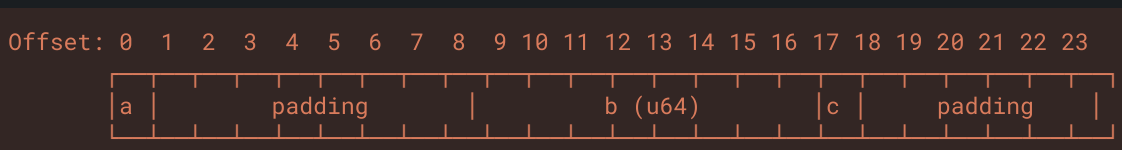
## 1.4 Struct Layout and Padding

When Rust lays out a struct, it must ensure each field is properly aligned. This often requires **padding** — unused bytes inserted between fields.

### C-Compatible Layout (`#[repr(C)]`)

```rust
#[repr(C)]
struct PaddedStruct {
    a: u8,   // offset 0, size 1
    // 7 bytes padding (to align b to 8)
    b: u64,  // offset 8, size 8
    c: u8,   // offset 16, size 1
    // 7 bytes padding (to make total size multiple of 8)
}
// Total: 24 bytes, align 8
```

Memory layout:




### Rust's Default Layout

Without `#[repr(C)]`, Rust can **reorder fields** to minimize padding:

```rust
struct RustStruct {
    a: u8,
    b: u64,
    c: u8,
}
// Rust reorders to: b, a, c
// Layout: 8 + 1 + 1 + 6(padding) = 16 bytes
```

This is why you can't reliably take pointers to struct fields and do arithmetic between them unless you use `#[repr(C)]`.

---

## 1.5 Combining Layouts

### The `extend` Method

Places one layout after another with proper padding:

```rust
let a = Layout::new::<u8>();   // size=1, align=1
let b = Layout::new::<u32>();  // size=4, align=4

let (combined, offset_of_b) = a.extend(b)?;
// combined: size=8, align=4
// offset_of_b: 4 (after 3 bytes of padding)
```

This is how you manually compute struct layouts:

```rust
pub fn repr_c(fields: &[Layout]) -> Result<(Layout, Vec<usize>), LayoutError> {
    let mut offsets = Vec::new();
    let mut layout = Layout::from_size_align(0, 1)?;

    for &field in fields {
        let (new_layout, offset) = layout.extend(field)?;
        layout = new_layout;
        offsets.push(offset);
    }

    // Don't forget trailing padding!
    Ok((layout.pad_to_align(), offsets))
}
```

### The `pad_to_align` Method

Adds trailing padding so the total size is a multiple of alignment. This is essential for arrays — without it, elements beyond the first would be misaligned.

```rust
let layout = Layout::from_size_align(5, 4)?;  // size=5, align=4
let padded = layout.pad_to_align();           // size=8, align=4
```

### The `repeat` Method

Creates a layout for an array of `n` elements:

```rust
let element = Layout::new::<u32>();
let (array_layout, stride) = element.repeat(10)?;
// array_layout: size=40, align=4
// stride: 4 (distance between elements)
```

## 1.6 The Bit Manipulation Tricks

Layout code uses fast bitwise operations because alignment is always a power of two.

### Rounding Up to Alignment

```
// Round size up to next multiple of align
fn round_up(size: usize, align: usize) -> usize {
    (size + align - 1) & !(align - 1)
}
```

How it works: - `align - 1` gives a mask of the low bits (e.g., align=8 → mask=0b111) - Adding `align - 1` ensures we round up, not down
- `!(align - 1)` inverts the mask - AND-ing clears the low bits, rounding down to alignment

Example with size=5, align=4:

```
size + align - 1 = 5 + 3 = 8 = 0b1000
!(align - 1)     = !3    = ...11111100
8 & ...11111100  = 8
```

### Calculating Padding

```
fn padding_needed(size: usize, align: usize) -> usize {
    let rounded = (size + align - 1) & !(align - 1);
    rounded - size
}
```

## 1.7 Zero-Sized Types

Rust has types with size 0:

```
let unit_layout = Layout::new::<()>();      // size=0, align=1
let empty_layout = Layout::new::<[u8; 0]>(); // size=0, align=1

struct Empty;
let empty_struct = Layout::new::<Empty>();  // size=0, align=1
```

Arrays of ZSTs also have size 0:

```rust
let million_units = Layout::array::<()>(1_000_000)?;
// size=0, align=1
// A million units take zero bytes!
```

This has important implications for allocation — you can't actually allocate zero bytes from most allocators, so Rust handles ZSTs specially (we'll see this in Chapter 3).

---

## 1.8 Key Takeaways

1. **Layout = size + alignment**, nothing more
2. **Alignment must be a power of two** — required by hardware and OS allocators
3. **Size limit is isize::MAX** — to ensure pointer arithmetic is safe
4. **Padding is invisible but real** — it affects struct sizes and memory usage
5. **Rust can reorder struct fields** — use `#[repr(C)]` for predictable layout
6. **Zero-sized types have size 0** — they're handled specially throughout the allocation system

---

## Source Files

| File | Purpose |
|---|---|
| `library/core/src/alloc/layout.rs` | The Layout type and all its methods |
| `library/core/src/alloc/mod.rs` | Module organization and re-exports |

---

## Exercises

1. Calculate the layout of this struct by hand, then verify with `Layout::new`:

   ```rust
   #[repr(C)]
   struct Mixed { a: u8, b: u32, c: u8, d: u16 }
   ```

2. Why can't alignment be zero? What would happen if you tried?

3. The `extend` method doesn't add trailing padding. Why is `pad_to_align` a separate step?
4. What's the maximum number of `u8` values you could store in a single allocation?

---

## Next Chapter

Chapter 2: The Allocator Traits →

We'll see how Layout gets used by the allocation interfaces — `GlobalAlloc` and `Allocator` — and trace the path from Rust code down to `libc::malloc`.

# Chapter 2: The Allocator Traits

> *"It's undefined behavior if global allocators unwind."*
> — Safety requirements for GlobalAlloc

## Introduction

With `Layout` defining *what* memory we need, we now examine *how* to get it. Rust provides two allocator traits:

- **`GlobalAlloc`** — Stable, simple, the current production workhorse
- **`Allocator`** — Unstable, more sophisticated, the future

Both define contracts that any memory allocator must satisfy. We'll trace the path from these abstract traits down to actual `libc::malloc` calls.

---

## 2.1 GlobalAlloc: The Stable Interface

```
pub unsafe trait GlobalAlloc {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8;
    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout);

    // Provided defaults (can be overridden for efficiency)
    unsafe fn alloc_zeroed(&self, layout: Layout) -> *mut u8;
    unsafe fn realloc(&self, ptr: *mut u8, layout: Layout, new_size: usize) ->
*mut u8;
}
```

### Why `unsafe trait`?

The trait itself is unsafe to implement because the compiler cannot verify the guarantees. Implementors must manually ensure:

1. **No unwinding** — Panicking from an allocator is undefined behavior
2. **Correct layouts** — Returned memory must match the requested size and alignment

3. **Valid pointers** — Returned pointers must be usable for reads/writes

### The Four Methods

**`alloc`** — The core allocation method - Takes a `Layout`, returns a raw pointer - Returns null on failure (never panics!) - Memory is uninitialized — contains garbage - **Zero-sized layouts are undefined behavior**

**`dealloc`** — Frees memory - Must receive a pointer from a previous `alloc` on the same allocator - Must receive the *same layout* used to allocate - Using the wrong layout is undefined behavior

**`alloc_zeroed`** — Allocation with zero-initialization - Default implementation: `alloc` then `memset` to zero - Can be overridden to use `calloc` for efficiency

**`realloc`** — Resize an allocation - May return the same pointer (grown in place) or a new one - Copies `min(old_size, new_size)` bytes to the new location - Old pointer becomes invalid after successful realloc

---

## 2.2 The Safety Contracts

These contracts are **not checked by the compiler**. Violating them causes undefined behavior — crashes, corruption, security vulnerabilities.

### Contract for `alloc`

```
unsafe fn alloc(&self, layout: Layout) -> *mut u8;
```

**Preconditions (caller must ensure):** - `layout.size() > 0` — Zero-sized allocations are UB

**Postconditions (implementor must ensure):** - Returns null OR a valid pointer that is: - Properly aligned to `layout.align()` - Valid for reads and writes of `layout.size()` bytes - Not overlapping with any other live allocation

### Contract for `dealloc`

```
unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout);
```

**Preconditions (caller must ensure):** - `ptr` was returned by a previous `alloc` or `realloc` on this allocator - `layout` is the same layout used to allocate - The memory has not already been

deallocated

**Postconditions:** - The memory is released back to the allocator - The pointer becomes invalid — any use is UB

### Contract for `realloc`

```
unsafe fn realloc(&self, ptr: *mut u8, layout: Layout, new_size: usize) -> *mut
u8;
```

**Preconditions:** - Same as `dealloc` for `ptr` and `layout` - `new_size > 0` - `new_size` rounded up to `layout.align()` must not overflow `isize`

**Postconditions:** - If returns null: original allocation is unchanged, `ptr` still valid - If returns non-null: - Original `ptr` is now invalid - Returned pointer has the new size - First `min(old_size, new_size)` bytes are preserved

---

## 2.3 The Default Implementations

GlobalAlloc provides default implementations for `alloc_zeroed` and `realloc`. Understanding these reveals optimization opportunities.

### Default `alloc_zeroed`

```
unsafe fn alloc_zeroed(&self, layout: Layout) -> *mut u8 {
    let size = layout.size();
    let ptr = unsafe { self.alloc(layout) };
    if !ptr.is_null() {
        unsafe { ptr::write_bytes(ptr, 0, size) };
    }
    ptr
}
```

This is inefficient — it allocates uninitialized memory then writes zeros. A smart implementation uses `calloc`, which: 1. May get pre-zeroed pages from the OS 2. Uses copy-on-write to share zero pages 3. Avoids touching memory until actually needed

### Default `realloc`

```
unsafe fn realloc(&self, ptr: *mut u8, layout: Layout, new_size: usize) -> *mut
u8 {
    let new_layout = unsafe { Layout::from_size_align_unchecked(new_size,
layout.align()) };
    let new_ptr = unsafe { self.alloc(new_layout) };
    if !new_ptr.is_null() {
        unsafe {
            ptr::copy_nonoverlapping(ptr, new_ptr, cmp::min(layout.size(),
new_size));
            self.dealloc(ptr, layout);
        }
    }
    new_ptr
}
```

This is the naive **allocate-copy-free** strategy: 1. Allocate new block 2. Copy old data 3. Free old block

Real allocators (like glibc) try to **grow in place** first — if there's free space after the current block, just extend it. This is much faster for growing `Vec`s.

---

## 2.4 The Unix Implementation

Now let's see how these traits connect to actual OS memory:

```rust
// From library/std/src/sys/alloc/unix.rs

unsafe impl GlobalAlloc for System {
    #[inline]
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        if layout.align() <= MIN_ALIGN && layout.align() <= layout.size() {
            unsafe { libc::malloc(layout.size()) as *mut u8 }
        } else {
            unsafe { aligned_malloc(&layout) }
        }
    }

    #[inline]
    unsafe fn dealloc(&self, ptr: *mut u8, _layout: Layout) {
        unsafe { libc::free(ptr as *mut libc::c_void) }
    }

    #[inline]
    unsafe fn alloc_zeroed(&self, layout: Layout) -> *mut u8 {
        if layout.align() <= MIN_ALIGN && layout.align() <= layout.size() {
            unsafe { libc::calloc(layout.size(), 1) as *mut u8 }
        } else {
            let ptr = unsafe { self.alloc(layout) };
            if !ptr.is_null() {
                unsafe { ptr::write_bytes(ptr, 0, layout.size()) };
            }
            ptr
        }
    }

    #[inline]
    unsafe fn realloc(&self, ptr: *mut u8, layout: Layout, new_size: usize) ->
*mut u8 {
        if layout.align() <= MIN_ALIGN && layout.align() <= new_size {
            unsafe { libc::realloc(ptr as *mut libc::c_void, new_size) as *mut u8
}
        } else {
            unsafe { realloc_fallback(self, ptr, layout, new_size) }
        }
    }
}
```

## The Two-Path Strategy

Every method has the same pattern: 1. **Simple path**: If alignment requirements are modest, use standard libc functions 2. **Aligned path**: If alignment is strict, use special aligned allocation

The check `layout.align() <= MIN_ALIGN && layout.align() <= layout.size()` determines which path:

- `MIN_ALIGN` is the guaranteed alignment from `malloc` (usually 8 or 16 bytes)
- The second condition (`align <= size`) works around jemalloc quirks where small allocations might not respect alignment

### Aligned Allocation

```
unsafe fn aligned_malloc(layout: &Layout) -> *mut u8 {
    let mut out = ptr::null_mut();
    let align = layout.align().max(size_of::<usize>());
    let ret = unsafe { libc::posix_memalign(&mut out, align, layout.size()) };
    if ret != 0 { ptr::null_mut() } else { out as *mut u8 }
}
```

`posix_memalign` is the POSIX standard for aligned allocation: - First argument: output pointer (modified by the function) - Second argument: alignment (must be ≥ `sizeof(void*)` and power of 2) - Third argument: size - Returns 0 on success, error code on failure

### Platform-Specific Bug Workarounds

```
#[cfg(target_vendor = "apple")]
{
    if layout.align() > (1 << 31) {
        return ptr::null_mut();
    }
}
```

Older macOS/iOS versions have a bug where `posix_memalign` with huge alignments (>2GB) returns a pointer that isn't actually aligned. Rather than silently corrupt memory, Rust returns null immediately.

---

## 2.5 The Allocator Trait (Unstable)

The newer `Allocator` trait fixes several limitations of `GlobalAlloc`:

```rust
pub unsafe trait Allocator {
    fn allocate(&self, layout: Layout) -> Result<NonNull<[u8]>, AllocError>;
    unsafe fn deallocate(&self, ptr: NonNull<u8>, layout: Layout);

    // Provided defaults
    fn allocate_zeroed(&self, layout: Layout) -> Result<NonNull<[u8]>,
AllocError>;
    unsafe fn grow(...) -> Result<NonNull<[u8]>, AllocError>;
    unsafe fn grow_zeroed(...) -> Result<NonNull<[u8]>, AllocError>;
    unsafe fn shrink(...) -> Result<NonNull<[u8]>, AllocError>;
}
```

## Key Differences from GlobalAlloc

| Aspect | GlobalAlloc | Allocator |
|---|---|---|
| Error handling | Null pointer | `Result<_, AllocError>` |
| Pointer type | `*mut u8` | `NonNull<[u8]>` |
| Zero-sized alloc | Undefined behavior | Allowed |
| Resize operations | Single `realloc` | Separate `grow` / `shrink` |
| Alignment changes | Not supported | Supported in grow/shrink |

## The Fat Pointer Return

`NonNull<[u8]>` is a **fat pointer** — it contains both the address and the length. This lets the allocator tell you the *actual* size allocated:

```rust
// You ask for 30 bytes, allocator gives you 32 due to internal binning
let result = allocator.allocate(Layout::from_size_align(30, 1)?)?;
let actual_size = result.len();  // Might be 32!
```

Collections like `Vec` can use this extra space for free.

## Zero-Sized Allocations

Unlike GlobalAlloc, Allocator handles ZSTs gracefully:

```
// This is fine with Allocator
let layout = Layout::new::<()>();  // size=0
let ptr = allocator.allocate(layout)?;  // Returns a valid NonNull
```

The implementation must catch this case and return a dangling-but-valid pointer without calling the underlying allocator.

---

## 2.6 The Global Allocator

By default, Rust uses the system allocator. You can change it:

```
use std::alloc::{GlobalAlloc, Layout, System};

#[global_allocator]
static ALLOCATOR: System = System;
```

Or use a custom allocator:

```
struct MyAllocator;

unsafe impl GlobalAlloc for MyAllocator {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        // Your implementation
    }

    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout) {
        // Your implementation
    }
}

#[global_allocator]
static ALLOCATOR: MyAllocator = MyAllocator;
```

### Example: A Simple Bump Allocator

From the GlobalAlloc documentation:

```
const ARENA_SIZE: usize = 128 * 1024;

#[repr(C, align(4096))]
struct SimpleAllocator {
    arena: UnsafeCell<[u8; ARENA_SIZE]>,
    remaining: AtomicUsize,
}

unsafe impl GlobalAlloc for SimpleAllocator {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        let size = layout.size();
        let align = layout.align();
        let align_mask = !(align - 1);

        if align > 4096 {
            return null_mut();
        }

        let mut allocated = 0;
        self.remaining
            .fetch_update(Relaxed, Relaxed, |mut remaining| {
                if size > remaining { return None; }
                remaining -= size;
                remaining &= align_mask;  // Round down for alignment
                allocated = remaining;
                Some(remaining)
            })
            .ok()?;

        unsafe { self.arena.get().cast::<u8>().add(allocated) }
    }

    unsafe fn dealloc(&self, _ptr: *mut u8, _layout: Layout) {
        // Bump allocators don't free individual allocations
    }
}
```

This "bump allocator" is extremely fast (just an atomic decrement) but never reuses memory. It's used in compilers, game frames, and other short-lived contexts.

---

## 2.7 The Optimizer's Freedom

A critical note from the docs:

> You must not rely on allocations actually happening, even if there are explicit heap allocations in the source.

The optimizer can: - **Eliminate allocations** — `drop(Box::new(42))` might not allocate at all - **Move to stack** — Small heap allocations might become stack allocations - **Merge allocations** — Multiple small allocations might become one

This means debugging allocators that count allocations can be misleading. The count depends on optimization level!

---

## 2.8 Key Takeaways

1. **GlobalAlloc is stable and simple** — Four methods, raw pointers, null for errors
2. **Allocator is the future** — Result types, fat pointers, ZST support
3. **Safety contracts are critical** — Violating them is undefined behavior
4. **Default implementations are naive** — Override for efficiency
5. **Platform quirks exist** — Real allocators handle bugs in underlying systems
6. **The optimizer can eliminate allocations** — Don't rely on allocation counts

---

## Source Files

| File | Purpose |
|---|---|
| `library/core/src/alloc/global.rs` | GlobalAlloc trait definition |
| `library/core/src/alloc/mod.rs` | Allocator trait and AllocError |
| `library/std/src/sys/alloc/unix.rs` | Unix implementation |
| `library/std/src/sys/alloc/windows.rs` | Windows implementation |

---

## Exercises

1. Why must allocators never panic? What would happen if `alloc` panicked inside `Box::new`?

2. The default `realloc` always allocates new memory. When would `libc::realloc` be able to grow in place?
3. Write a simple allocator that tracks total bytes allocated (hint: wrap the System allocator).
4. Why does `Allocator::allocate` return `NonNull<[u8]>` instead of just `NonNull<u8>`?

---

## Next Chapter

Chapter 3: Box — Owned Heap Allocation →

We'll see how `Box<T>` uses these allocator traits to provide safe, owned heap memory with automatic cleanup.

# Chapter 3: Box — Owned Heap Allocation

> *"Box provides the simplest form of heap allocation in Rust."*
> — Standard library documentation

## Introduction

Having understood memory layouts (Chapter 1) and allocator traits (Chapter 2), we now see how they combine into something useful: `Box<T>`, Rust's simplest smart pointer.

Box is deceptively simple — it's just a pointer to heap memory. But that simplicity hides careful engineering around allocation, deallocation, and the ownership system.

---

## 3.1 What is Box?

At its core, Box is: 1. A pointer to heap-allocated memory 2. Ownership of that memory 3. Automatic deallocation when the Box goes out of scope

```
let boxed: Box<i32> = Box::new(42);
// - 4 bytes allocated on heap
// - Value 42 written there
// - boxed holds the pointer (8 bytes on stack)

// When boxed goes out of scope:
// - Memory is automatically freed
```

### The Structure

```
pub struct Box<
    T: ?Sized,
    A: Allocator = Global,
>(Unique<T>, A);
```

Two fields: - **Field 0**: `Unique<T>` — A pointer wrapper that asserts exclusive ownership - **Field 1**: `A` — The allocator (defaults to `Global`, which is zero-sized)

For the common case of `Box<T>` with the global allocator, Box is exactly **one pointer** in size — 8 bytes on 64-bit systems.

### Why Box Exists

**1. Heap allocation for large data:**

```
// This 1MB array would overflow the stack
let huge: Box<[u8; 1_000_000]> = Box::new([0u8; 1_000_000]);
// Only 8 bytes on stack, 1MB on heap
```

**2. Recursive data structures:**

```
enum List<T> {
    Cons(T, Box<List<T>>),  // Box breaks the infinite size
    Nil,
}
// Without Box: "recursive type has infinite size"
```

**3. Trait objects:**

```
let animal: Box<dyn Animal> = Box::new(Dog { name: "Rex" });
// Box holds a fat pointer: data ptr + vtable ptr
```

**4. Transferring ownership efficiently:**

```
fn transfer(data: Box<[u8; 1_000_000]>) { ... }
// Moving Box copies 8 bytes, not 1MB
```

---

## 3.2 Box Creation: The Allocation Path

### The Simple Case: `Box::new`

```
pub fn new(x: T) -> Self {
    return box_new(x);
}

#[rustc_intrinsic]
pub fn box_new<T>(x: T) -> Box<T>;
```

`box_new` is a **compiler intrinsic** — the compiler generates specialized code for it. Why? To avoid unnecessary copies.

Without the intrinsic, the naive implementation would be:

```
fn new_naive(x: T) -> Box<T> {
    let mut uninit = Box::new_uninit();  // Allocate
    uninit.write(x);                     // Copy x to heap
    unsafe { uninit.assume_init() }
}
```

This copies `x` from wherever it starts → stack → heap. The intrinsic lets the compiler write `x` directly to heap memory.

## The Real Allocation: `try_new_uninit_in`

```
pub fn try_new_uninit_in(alloc: A) -> Result<Box<MaybeUninit<T>, A>, AllocError>
where
    A: Allocator,
{
    let ptr = if T::IS_ZST {
        NonNull::dangling()
    } else {
        let layout = Layout::new::<MaybeUninit<T>>();
        alloc.allocate(layout)?.cast()
    };
    unsafe { Ok(Box::from_raw_in(ptr.as_ptr(), alloc)) }
}
```

Step by step:

1. **ZST check**: If `T` has zero size, don't allocate — return a "dangling" pointer
2. **Calculate layout**: Get size and alignment requirements
3. **Call allocator**: `alloc.allocate(layout)` — this eventually calls `malloc`
4. **Cast pointer**: From `NonNull<[u8]>` to `NonNull<MaybeUninit<T>>`
5. **Wrap in Box**: `Box::from_raw_in` creates the Box struct

## Zero-Sized Types: No Allocation

```
let unit: Box<()> = Box::new(());
```

What happens: - `T::IS_ZST` is true for `()` - `NonNull::dangling()` returns a well-aligned, non-null pointer that points to nothing - No allocator call occurs - Box still has a valid (but meaningless) pointer

The dangling pointer is typically `0x1` for align=1 types — non-null, but obviously not from the heap.

---

## 3.3 Box Destruction: The Drop Implementation

```
unsafe impl #[may_dangle] T: ?Sized, A: Allocator> Drop for Box<T, A> {
    fn drop(&mut self) {
        // The T in the Box is dropped by the compiler before this runs

        let ptr = self.0;

        unsafe {
            let layout = Layout::for_value_raw(ptr.as_ptr());
            if layout.size() != 0 {
                self.1.deallocate(From::from(ptr.cast()), layout);
            }
        }
    }
}
```

### Drop Order

Critical detail: **T's destructor runs before Box's destructor**.

```
struct Noisy(i32);
impl Drop for Noisy {
    fn drop(&mut self) { println!("Dropping {}", self.0); }
}

{
    let b = Box::new(Noisy(42));
}  // Prints "Dropping 42", then Box::drop runs
```

The compiler inserts `drop_in_place` for the contents, *then* calls `Box::drop` which only handles memory.

### The `#[may_dangle]` Attribute

This tells the compiler that even though `T` might contain dangling references after its destructor runs, that's okay — Box won't access `T`'s data, only deallocate the memory.

### Zero-Size Check

```
if layout.size() != 0 {
    self.1.deallocate(...);
}
```

For ZSTs, we never allocated, so we must not deallocate. The dangling pointer never touched the allocator.

---

## 3.4 Raw Pointer Conversions

Box provides escape hatches for interop with raw pointers:

### `into_raw`: Box → Raw Pointer

```
pub fn into_raw(b: Self) -> *mut T {
    let mut b = mem::ManuallyDrop::new(b);
    &raw mut **b
}
```

This: 1. Wraps Box in `ManuallyDrop` to prevent Drop from running 2. Returns the raw pointer

**After calling `into_raw`:** - You own the raw pointer - You're responsible for eventually freeing the memory - The Box no longer exists

### `from_raw`: Raw Pointer → Box

```
pub unsafe fn from_raw(raw: *mut T) -> Self {
    unsafe { Self::from_raw_in(raw, Global) }
}
```

**Safety requirements:** - Pointer must have come from `Box::into_raw` (or equivalent) - Pointer must not have been freed - Must be called at most once per allocation

### The Round-Trip Pattern

```rust
let original = Box::new(String::from("hello"));
let ptr = Box::into_raw(original);

// ... do things with raw pointer ...

let recovered = unsafe { Box::from_raw(ptr) };
// Memory will be freed when recovered is dropped
```

This is essential for FFI — passing heap memory to C code and getting it back.

---

## 3.5 Memory Layout and ABI

### Box is ABI-Compatible with C Pointers

```rust
// Rust
#[no_mangle]
pub extern "C" fn create_foo() -> Box<Foo> {
    Box::new(Foo::new())
}

// C
struct Foo* create_foo(void);
```

For sized types, `Box<T>` has the exact same representation as `T*` in C. This enables zero-cost FFI.

### Box with Custom Allocators

```rust
Box<T, A: Allocator>
```

When `A` is not `Global`: - The allocator instance is stored in the Box - Box size increases by `size_of::<A>()` - The allocator must be used for deallocation

For `Global` (a zero-sized type), no overhead is added.

---

## 3.6 The Deref Magic

Box implements `Deref` and `DerefMut`:

```rust
impl<T: ?Sized, A: Allocator> Deref for Box<T, A> {
    type Target = T;

    fn deref(&self) -> &T {
        &**self
    }
}
```

This enables:

```rust
let b: Box<String> = Box::new(String::from("hello"));
println!("{}", b.len());  // Calls String::len through Deref
b.push_str(" world");     // Calls String::push_str through DerefMut
```

The `**self` syntax: - First `*` dereferences the `&self` to get `Box<T>` - Second `*` is the built-in dereference for `Box<T>` to get `T`

---

## 3.7 Box and Pinning

```rust
impl<T: ?Sized, A: Allocator> Unpin for Box<T, A> {}
```

Box is **always Unpin**, even when T is not. This is a deliberate design choice:

- Box owns its contents and can always move them
- The address of the Box can change (Box is on the stack)
- But `Pin<Box<T>>` pins the *heap* contents, not the Box itself

```rust
let pinned: Pin<Box<MyFuture>> = Box::pin(my_future);
// The Box can move, but the MyFuture on the heap cannot
```

---

## 3.8 Practical Examples

### Recursive Data Structures

```
enum BinaryTree<T> {
    Leaf(T),
    Node {
        left: Box<BinaryTree<T>>,
        right: Box<BinaryTree<T>>,
    },
}

let tree = BinaryTree::Node {
    left: Box::new(BinaryTree::Leaf(1)),
    right: Box::new(BinaryTree::Node {
        left: Box::new(BinaryTree::Leaf(2)),
        right: Box::new(BinaryTree::Leaf(3)),
    }),
};
```

## Trait Objects

```
trait Animal {
    fn speak(&self);
}

struct Dog;
impl Animal for Dog {
    fn speak(&self) { println!("Woof!"); }
}

let animal: Box<dyn Animal> = Box::new(Dog);
animal.speak();
```

`Box<dyn Animal>` is a **fat pointer**: 16 bytes containing: - Pointer to the data (8 bytes) - Pointer to the vtable (8 bytes)

## FFI with C

```
#[no_mangle]
pub extern "C" fn create_buffer(size: usize) -> *mut u8 {
    let buffer: Box<[u8]> = vec![0u8; size].into_boxed_slice();
    Box::into_raw(buffer) as *mut u8
}

#[no_mangle]
pub unsafe extern "C" fn free_buffer(ptr: *mut u8, size: usize) {
    let slice = std::slice::from_raw_parts_mut(ptr, size);
    let _ = Box::from_raw(slice as *mut [u8]);
}
```

## 3.9 Exploration Program

```
use std::alloc::Layout;
use std::mem;

fn main() {
    // Box is pointer-sized
    println!("Box<i32> size: {} bytes", mem::size_of::<Box<i32>>());
    println!("Box<[u8; 1000]> size: {} bytes", mem::size_of::<Box<[u8; 1000]>>
());

    // ZST Box has a dangling pointer
    let unit_box: Box<()> = Box::new(());
    let ptr = Box::into_raw(unit_box);
    println!("Box<()> pointer: {:p}", ptr);  // Usually 0x1
    let _ = unsafe { Box::from_raw(ptr) };

    // Drop order demonstration
    struct Loud(i32);
    impl Drop for Loud {
        fn drop(&mut self) { println!("Dropping Loud({})", self.0); }
    }

    {
        let outer = Box::new(Loud(1));
        let inner = Box::new(Loud(2));
        println!("Both alive");
    }
    println!("Both dropped (LIFO order)");
}
```

## 3.10 Key Takeaways

1. **Box is just a pointer** — 8 bytes for the common case
2. **Allocation happens through the Allocator trait** — Eventually calls malloc
3. **ZSTs don't allocate** — They get dangling pointers
4. **Drop is two-phase** — Content destructor first, then memory deallocation
5. **Box is ABI-compatible with C pointers** — Enables zero-cost FFI
6. **Deref makes Box transparent** — You can call T's methods directly

## Source Files

| File | Purpose |
|---|---|
| `library/alloc/src/boxed.rs` | Box type and all implementations |
| `library/alloc/src/boxed/convert.rs` | From/Into implementations |
| `library/alloc/src/boxed/thin.rs` | ThinBox for DSTs |

---

## Exercises

1. What's the size of `Box<Box<i32>>`? Why?
2. Why can't you call `Box::from_raw` twice on the same pointer?
3. Implement a simple smart pointer that's like Box but logs all allocations/deallocations.
4. What happens if you call `mem::forget` on a Box? Is this a memory leak?
5. Why does `Box<[T]>` (boxed slice) need to be a fat pointer, unlike `Box<[T; N]>` (boxed array)?

---

## Next Chapter

Chapter 4: Vec — Dynamic Arrays →

We'll see how `Vec<T>` builds on these primitives to provide a growable array, introducing the complexity of capacity management and reallocation strategies.

# Chapter 4: Vec — Dynamic Arrays

> *"Vec is and always will be a (pointer, capacity, length) triplet. No more, no less."*
> — Vec documentation guarantees

## Introduction

Box gives us a single heap allocation. But what if we need a collection that grows? Enter `Vec<T>`, Rust's dynamic array — the most commonly used collection type.

Vec builds on everything we've learned: Layout for calculating memory requirements, the Allocator traits for getting memory, and ownership semantics for safe resource management. But it adds crucial new complexity: **capacity management** and **growth strategies**.

---

## 4.1 The Vec Memory Model

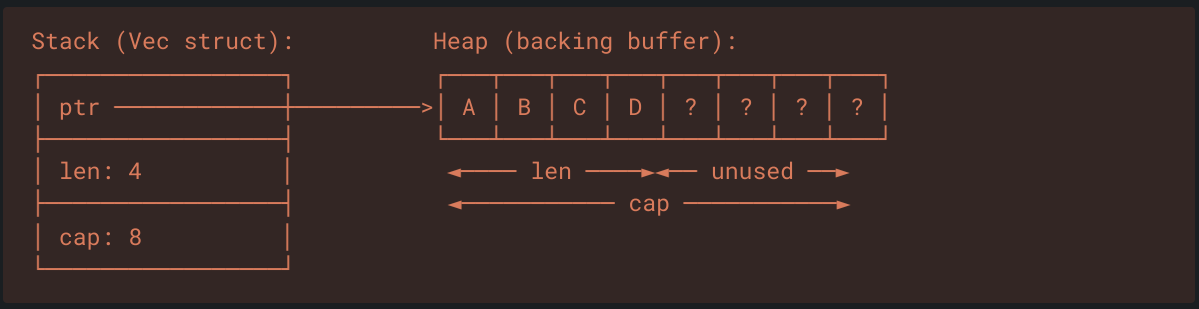
### The Three-Field Structure

```
pub struct Vec<T, A: Allocator = Global> {
    buf: RawVec<T, A>,
    len: usize,
}
```

Vec delegates all allocation to `RawVec`, keeping only the element count itself. Effectively, Vec has three pieces of state: - **ptr**: Pointer to the heap allocation - **len**: Number of elements currently stored - **cap**: Total capacity (elements that *could* be stored)

**Invariant**: `len ≤ cap` always.

### Memory Layout




**On the stack**: 24 bytes (three `usize` values on 64-bit)
**On the heap**: `cap * size_of::<T>()` bytes

### Vec Guarantees

The documentation makes explicit promises:

1. **No small-buffer optimization** — Contents are always on the heap, never the stack
2. **Never auto-shrinks** — Capacity only grows unless explicitly shrunk
3. **Null-pointer optimized** — The pointer is never null (uses dangling for empty)
4. **No specific growth strategy guaranteed** — Only that push is amortized O(1)

### Empty Vec: No Allocation

```
let v: Vec<i32> = Vec::new();
// ptr: dangling (not null, but not allocated)
// len: 0
// cap: 0
```

A brand new Vec doesn't allocate until you push the first element. The pointer is "dangling" — a valid, non-null pointer that doesn't point to allocated memory. This enables null-pointer optimization for `Option<Vec<T>>`.

## 4.2 RawVec: The Allocation Engine

Vec delegates all allocation logic to `RawVec`. This separation of concerns keeps Vec focused on element management while RawVec handles bytes.

### The Inner/Outer Architecture

Modern Rust's RawVec uses a two-layer design to reduce code bloat:

```
pub(crate) struct RawVec<T, A: Allocator = Global> {
    inner: RawVecInner<A>,
    _marker: PhantomData<T>,
}

struct RawVecInner<A: Allocator = Global> {
    ptr: Unique<u8>,
    cap: Cap,
    alloc: A,
}
```

**Why this split?**

`RawVecInner` is only generic over the allocator, not the element type. Most operations (grow, shrink, deallocate) only need the element's Layout, not its actual type. This means: - `RawVecInner<Global>` is one type regardless of `T` - Less monomorphization = smaller binaries - Core allocation logic compiled once, not per-type

### The Cap Type

```
type Cap = core::num::niche_types::UsizeNoHighBit;
```

Instead of raw `usize`, capacity uses a special type with a safety invariant: **cap must be in the range `0..=isize::MAX`**. This: - Matches the `isize::MAX` allocation limit from Layout - Enables the compiler to assume this invariant - The high bit is guaranteed zero (potential niche optimization)

### Why Separate RawVec?

1. **Reusability**: Other types can use RawVec (`VecDeque`, `String`)
2. **Complexity isolation**: Allocation edge cases live in one place
3. **Unsafe encapsulation**: RawVec contains the unsafe code
4. **Monomorphization reduction**: Inner type is allocator-generic only

## 4.3 Growth Strategies: The Amortized Analysis

### The Problem with Naive Growth

If Vec grew by exactly one element each time:

```
let mut v = Vec::new();
for i in 0..1000 {
    v.push(i);  // Reallocates every single time!
}
```

Each push would: 1. Allocate new buffer (size n+1) 2. Copy n elements 3. Free old buffer

Total work: 1 + 2 + 3 + ... + 999 = **O(n²)** for n pushes!

## The Doubling Strategy

The core growth logic in `grow_amortized`:

```
fn grow_amortized(&mut self, len: usize, additional: usize, elem_layout: Layout)
    -> Result<(), TryReserveError>
{
    // Calculate required capacity
    let required_cap = len.checked_add(additional).ok_or(CapacityOverflow)?;

    // Exponential growth: at least double, but satisfy requirement
    let cap = cmp::max(self.cap.as_inner() * 2, required_cap);

    // Ensure we don't allocate tiny buffers
    let cap = cmp::max(min_non_zero_cap(elem_layout.size()), cap);

    // ... allocate with new capacity
}
```

## The min_non_zero_cap Optimization

Tiny allocations are inefficient. The standard library avoids them:

```
const fn min_non_zero_cap(size: usize) -> usize {
    if size == 1 {
        8   // Heap allocators round up to at least 8 bytes anyway
    } else if size <= 1024 {
        4   // Moderate-sized elements: start with 4
    } else {
        1   // Large elements: even 1 is significant
    }
}
```

This means: - `Vec<u8>` starts at capacity 8 (not 1) - `Vec<u64>` starts at capacity 4 - `Vec<[u8; 4096]>` starts at capacity 1

### Growth Timeline

```
Push 1: cap 0 → 8 (for u8) or 4 (for u64)
Push 2-8: no realloc
Push 9: cap 8 → 16 (doubled)
Push 17: cap 16 → 32
Push 33: cap 32 → 64
...
```

For 1000 elements in a `Vec<u64>`: ~8 reallocations total.

### The Math (Optional)

For n elements with doubling: - Number of reallocations: O(log n) - Total elements copied: 4 + 8 + 16 + ... + n/2 ≈ n

So n pushes cost O(n) work total, meaning **O(1) amortized per push**.

---

## 4.4 The Growth Machinery

### finish_grow: The Core Reallocation Function

All growth paths eventually call `finish_grow`:

```rust
fn finish_grow<A: Allocator>(
    new_layout: Layout,
    current_memory: Option<(NonNull<u8>, Layout)>,
    alloc: &mut A,
) -> Result<NonNull<[u8]>, TryReserveError> {
    let memory = if let Some((ptr, old_layout)) = current_memory {
        // Have existing allocation: try to grow it
        unsafe { alloc.grow(ptr, old_layout, new_layout) }
    } else {
        // No existing allocation: fresh allocate
        alloc.allocate(new_layout)
    };
    memory.map_err(|_| AllocError { layout: new_layout, non_exhaustive: () }
.into())
}
```

This is marked `#[cold]` — the compiler assumes it's rarely called, optimizing the hot path (no growth needed).

### The Two Growth Modes

**Amortized Growth** (`grow_amortized`): - Doubles capacity or grows to required, whichever is larger - Used by `push`, `reserve`, `extend` - Provides O(1) amortized operations

**Exact Growth** (`grow_exact`): - Grows to exactly the required capacity - Used by `reserve_exact` - No extra space beyond what's needed

---

## 4.5 Push and Pop

### Push: The Heart of Vec

```
pub fn push(&mut self, value: T) {
    // 1. Check if we need to grow
    if self.len == self.buf.capacity() {
        self.buf.grow_one();
    }

    // 2. Write the value
    unsafe {
        let end = self.as_mut_ptr().add(self.len);
        ptr::write(end, value);
    }

    // 3. Increment length
    self.len += 1;
}
```

Note the specialized `grow_one()` method — it's optimized for the common case of adding one element, avoiding some of the overhead of the general `reserve` calculation.

### Pop: Simple and Efficient

```
pub fn pop(&mut self) -> Option<T> {
    if self.len == 0 {
        None
    } else {
        self.len -= 1;
        unsafe {
            Some(ptr::read(self.as_ptr().add(self.len)))
        }
    }
}
```

**Pop never shrinks the buffer!** Capacity stays the same. This is intentional: - Shrinking would require reallocation - You might push again soon - Use `shrink_to_fit()` if you really want to reclaim memory

---

## 4.6 Reallocation: How grow() Works

When Vec needs more space, the allocator has options:

### The Allocator::grow Method

```
unsafe fn grow(
    &self,
    ptr: NonNull<u8>,
    old_layout: Layout,
    new_layout: Layout,
) -> Result<NonNull<[u8]>, AllocError>
```

The allocator can:

1. **Extend in place** (optimal) — If free space exists after the allocation
2. **Allocate-copy-free** (common) — New location, copy data, free old
3. **Fail** — Return `AllocError`

### What Happens on Failure

For infallible methods (`push`, `reserve`):

```
fn handle_error(e: TryReserveError) -> ! {
    match e.kind() {
        CapacityOverflow => capacity_overflow(), // panic!
        AllocError { layout, .. } => handle_alloc_error(layout), // abort
    }
}
```

For fallible methods (`try_reserve`): - Return `Err(TryReserveError)` — caller decides what to do

## 4.7 Drop: Cleaning Up Collections

```
unsafe impl<#[may_dangle] T, A: Allocator> Drop for Vec<T, A> {
    fn drop(&mut self) {
        unsafe {
            // Drop all elements
            ptr::drop_in_place(ptr::slice_from_raw_parts_mut(
                self.as_mut_ptr(),
                self.len
            ))
        }
        // RawVec::drop handles deallocation
    }
}
```

Two phases: 1. **Drop each element**: Calls destructor on `v[0]`, `v[1]`, ..., `v[len-1]` 2. **Deallocate buffer**: RawVec's Drop frees the memory

### RawVec's Drop

```
unsafe impl<#[may_dangle] T, A: Allocator> Drop for RawVec<T, A> {
    fn drop(&mut self) {
        unsafe { self.inner.deallocate(T::LAYOUT) }
    }
}
```

Note: RawVec **does not** drop contents — it only frees the allocation. This separation of concerns is crucial.

### Panic Safety

If an element's destructor panics: - Remaining elements might not be dropped (potential leak) - But memory safety is preserved — no double-free or use-after-free - This is a known limitation,

documented in the source

---

## 4.8 Zero-Sized Types in Vec

Vec handles ZSTs specially:

```
let mut v: Vec<()> = Vec::new();
for _ in 0..1_000_000 {
    v.push(());  // No allocation ever happens!
}
println!("len: {}, cap: {}", v.len(), v.capacity());
// len: 1000000, cap: 18446744073709551615 (usize::MAX)
```

### How It Works

```
const fn capacity(&self, elem_size: usize) -> usize {
    if elem_size == 0 { usize::MAX } else { self.cap.as_inner() }
}
```

For ZSTs: - `ptr` is dangling (never allocated) - `cap` returns `usize::MAX` (effectively infinite) - `len` tracks count normally - No memory is used except the Vec struct itself (24 bytes)

### Why usize::MAX?

This clever design means: - `len < cap` is always true for ZSTs (can always "push") - Overflow checking in len catches real issues - No special-casing in most Vec methods

---

## 4.9 Common Vec Operations

### Reserve: Ensure Capacity

```
let mut v = Vec::new();
v.reserve(100);  // Guarantee space for 100 more elements
// v.capacity() >= 100 + v.len()
```

Internally:

```rust
pub fn reserve(&mut self, additional: usize) {
    self.buf.reserve(self.len, additional);
}
```

### Shrink: Reclaim Memory

```rust
let mut v = vec![1, 2, 3, 4, 5];
v.pop();
v.pop();
v.shrink_to_fit();  // Reduce capacity to match length
// v.capacity() == 3
```

Note the check in `shrink_to_fit`:

```rust
pub fn shrink_to_fit(&mut self) {
    if self.capacity() > self.len {  // Avoid panic in RawVec
        self.buf.shrink_to_fit(self.len);
    }
}
```

### With Capacity: Avoid Reallocations

```rust
// If you know the size upfront
let mut v = Vec::with_capacity(1000);
for i in 0..1000 {
    v.push(i);  // No reallocations!
}
```

### Clear: Remove All Elements

```rust
let mut v = vec![1, 2, 3];
v.clear();
// len: 0, but capacity unchanged
```

## 4.10 Optimizer Hints

The source uses hints to help LLVM:

```rust
fn try_reserve(&mut self, len: usize, additional: usize, ...) -> Result<(),
TryReserveError> {
    if self.needs_to_grow(len, additional, elem_layout) {
        self.grow_amortized(len, additional, elem_layout)?;
    }
    unsafe {
        // Tell optimizer that growth succeeded
        hint::assert_unchecked(!self.needs_to_grow(len, additional,
elem_layout));
    }
    Ok(())
}
```

`hint::assert_unchecked` allows the optimizer to assume the condition is true, enabling better code generation for subsequent operations.

---

## 4.11 Key Takeaways

1. **Vec is a (ptr, len, cap) triplet** — 24 bytes on stack
2. **RawVec uses inner/outer split** — Reduces monomorphization
3. **Cap has a safety invariant** — Always ≤ `isize::MAX`
4. **Doubling growth gives O(1) amortized push** — Essential for performance
5. **min_non_zero_cap avoids tiny allocations** — 8 for bytes, 4 for moderate, 1 for large
6. **Pop doesn't shrink** — Use `shrink_to_fit` explicitly
7. **ZSTs are special-cased** — Infinite capacity, no allocation
8. **Drop is two-phase** — Elements first, then memory

---

## Source Files

| File | Purpose |
|---|---|
| `library/alloc/src/vec/mod.rs` | Main Vec implementation |
| `library/alloc/src/raw_vec/mod.rs` | RawVec and RawVecInner |

| | |
|---|---|
| `library/alloc/src/vec/into_iter.rs` | IntoIterator implementation |
| `library/alloc/src/vec/drain.rs` | Drain iterator |

---

## Exercises

1. Why does RawVec use `RawVecInner` instead of putting everything in one struct?
2. What's the minimum number of allocations for pushing 1000 `u64` values starting from an empty Vec?
3. Implement a `SimpleVec<T>` with just `new`, `push`, `pop`, and `Drop`.
4. Why is `Vec<()>` capacity `usize::MAX` instead of tracking a real count?
5. What happens if you `mem::forget` a Vec? Is this a memory leak? What about the elements?

---

## Hands-On Project: SimpleVec

Build a minimal Vec to solidify understanding:

```rust
use std::alloc::{alloc, dealloc, realloc, Layout};
use std::ptr::{self, NonNull};
use std::mem;

pub struct SimpleVec<T> {
    ptr: NonNull<T>,
    len: usize,
    cap: usize,
}

impl<T> SimpleVec<T> {
    pub fn new() -> Self {
        assert!(mem::size_of::<T>() != 0, "ZST not supported in SimpleVec");
        SimpleVec {
            ptr: NonNull::dangling(),
            len: 0,
            cap: 0,
        }
    }

    pub fn push(&mut self, value: T) {
        if self.len == self.cap {
            self.grow();
        }
        unsafe {
            ptr::write(self.ptr.as_ptr().add(self.len), value);
        }
        self.len += 1;
    }

    pub fn pop(&mut self) -> Option<T> {
        if self.len == 0 {
            None
        } else {
            self.len -= 1;
            unsafe { Some(ptr::read(self.ptr.as_ptr().add(self.len))) }
        }
    }

    fn grow(&mut self) {
        // Start with 4, then double (like std for moderate types)
        let new_cap = if self.cap == 0 { 4 } else { self.cap * 2 };
        let new_layout = Layout::array::<T>(new_cap).unwrap();

        assert!(new_layout.size() <= isize::MAX as usize, "Allocation too
large");

        let new_ptr = if self.cap == 0 {
            unsafe { alloc(new_layout) }
        } else {
            let old_layout = Layout::array::<T>(self.cap).unwrap();
            unsafe { realloc(self.ptr.as_ptr() as *mut u8, old_layout,
new_layout.size()) }
        };
```

```
        self.ptr = NonNull::new(new_ptr as *mut T).expect("allocation failed");
self.cap = new_cap;    }    pub fn len(&self) -> usize { self.len }    pub fn
capacity(&self) -> usize { self.cap }}impl<T> Drop for SimpleVec<T> {    fn
drop(&mut self) {        if self.cap > 0 {            // Drop all elements first
for i in 0..self.len {                unsafe {
ptr::drop_in_place(self.ptr.as_ptr().add(i)); }            }            // Then
deallocate            let layout = Layout::array::<T>(self.cap).unwrap();
unsafe { dealloc(self.ptr.as_ptr() as *mut u8, layout); }        }    }}
```

---

## What's Next?

With Vec mastered, you understand Rust's core collection machinery. Suggested further reading:

- **String**: Essentially `Vec<u8>` with UTF-8 guarantees
- **HashMap**: Uses RawTable, a more complex allocation strategy
- **VecDeque**: Ring buffer built on RawVec
- **BTreeMap**: Node-based allocation

You now have the foundation to understand any Rust collection at the allocation level!

box_exploration.rs

```rust
//! Exploring Box and Allocation Concepts
//!
//! This program demonstrates the allocation mechanics we've studied:
//! - Box creation and deallocation
//! - Zero-sized types (ZSTs)
//! - Layout calculations
//! - Custom allocators (using the unstable allocator_api)
//! - Memory layout inspection

use std::alloc::Layout;
use std::mem;

fn main() {
    println!("=== Rust Allocation Exploration ===\n");

    // Chapter 1: Basic Box mechanics
    explore_basic_box();

    // Chapter 2: Layout calculations
    explore_layouts();

    // Chapter 3: Zero-sized types
    explore_zst();

    // Chapter 4: Box memory layout
    explore_box_memory_layout();

    // Chapter 5: Drop behavior
    explore_drop_behavior();

    // Chapter 6: Raw pointer round-trip
    explore_raw_pointers();
}

/// Demonstrates basic Box creation and the heap allocation it triggers
fn explore_basic_box() {
    println!("--- 1. Basic Box Mechanics ---\n");

    // Simple heap allocation
    let boxed_int: Box<i32> = Box::new(42);
    println!("Boxed i32 value: {}", *boxed_int);
    println!("Box<i32> size: {} bytes", mem::size_of::<Box<i32>>());
    println!("i32 size: {} bytes", mem::size_of::<i32>());
    println!("→ Box is just a pointer ({} bytes on this platform)\n",
mem::size_of::<*const ()>());

    // Larger type to see the benefit of heap allocation
    let large_array: Box<[u8; 1024]> = Box::new([0u8; 1024]);
    println!("Box<[u8; 1024]> size on stack: {} bytes", mem::size_of::<Box<[u8;
1024]>>());
    println!("Actual [u8; 1024] size: {} bytes", mem::size_of::<[u8; 1024]>());
    println!("→ Only the pointer lives on stack, data is on heap\n");

    // The value is dropped when boxed_int and large_array go out of scope
```

```
    // Drop order: large_array first (LIFO), then boxed_int    drop(large_array);
drop(boxed_int);}/// Demonstrates Layout calculations - the foundation of
allocationfn explore_layouts() {    println!("--- 2. Layout Calculations ---\n");
// Layout for primitive types    let i32_layout = Layout::new::<i32>();
println!("Layout for i32:  size={}, align={}", i32_layout.size(),
i32_layout.align());    let u64_layout = Layout::new::<u64>();    println!
("Layout for u64:  size={}, align={}", u64_layout.size(), u64_layout.align());
// Layout for a struct - demonstrates padding    #[repr(C)]  // Use C layout for
predictable ordering    struct PaddedStruct {        a: u8,    // 1 byte        //
7 bytes padding here to align b        b: u64,   // 8 bytes, needs 8-byte
alignment        c: u8,    // 1 byte        // 7 bytes padding at end to make
struct size multiple of alignment    }    let padded_layout = Layout::new::
<PaddedStruct>();    println!("\nLayout for PaddedStruct (u8, u64, u8):");
println!("  size={}, align={}", padded_layout.size(), padded_layout.align());
println!("  → Expected: 1 + 7(pad) + 8 + 1 + 7(pad) = 24 bytes");    // Rust's
default repr can reorder fields!    struct RustStruct {        a: u8,        b:
u64,        c: u8,    }    let rust_layout = Layout::new::<RustStruct>();
println!("\nLayout for RustStruct (same fields, Rust repr):");    println!("
size={}, align={}", rust_layout.size(), rust_layout.align());    println!("  →
Rust may reorder to: u64, u8, u8 = 8 + 1 + 1 + 6(pad) = 16 bytes");    // Array
layout    let array_layout = Layout::array::<u32>(10).unwrap();    println!
("\nLayout for [u32; 10]: size={}, align={}",              array_layout.size(),
array_layout.align());    // The isize::MAX constraint    println!("\nMax
allocation size for align=8: {} bytes",              isize::MAX as usize - 7);
println!("That's approximately {} GB",              (isize::MAX as usize - 7) /
(1024 * 1024 * 1024));    println!();}/// Zero-sized types don't actually
allocate!fn explore_zst() {    println!("--- 3. Zero-Sized Types (ZSTs) ---\n");
// Unit type is zero-sized    let unit_layout = Layout::new::<()>();    println!
("Layout for (): size={}, align={}", unit_layout.size(), unit_layout.align());
// Empty struct is zero-sized    struct Empty;    let empty_layout =
Layout::new::<Empty>();    println!("Layout for Empty struct: size={}, align={}",
             empty_layout.size(), empty_layout.align());    // Box<()> is still
pointer-sized, but no heap allocation occurs!    let boxed_unit: Box<()> =
Box::new(());    println!("\nBox<()> stack size: {} bytes", mem::size_of::
<Box<()>>());    println!("→ Box still needs a pointer, but it's 'dangling' (no
heap alloc)");    // The pointer is non-null but doesn't point to allocated
memory    let ptr = Box::into_raw(boxed_unit);    println!("Box<()> raw pointer:
{:p}", ptr);    println!("→ This is a well-aligned dangling pointer, not from the
allocator");    // We must reconstruct the Box to avoid leaking (even though
there's nothing to leak)    let _ = unsafe { Box::from_raw(ptr) };    // Array of
ZSTs    let zst_array_layout = Layout::array::<()>(1_000_000).unwrap();
println!("\nLayout for [(); 1_000_000]: size={}, align={}",
zst_array_layout.size(), zst_array_layout.align());    println!("→ A million
units take zero bytes!\n");}/// Inspect the actual memory layout of Boxfn
explore_box_memory_layout() {    println!("--- 4. Box Memory Layout ---\n");
let boxed: Box<u64> = Box::new(0xDEADBEEF_CAFEBABE);    // Get the raw pointer
without consuming the Box    let ptr: *const u64 = &*boxed;    println!("Value:
0x{:016X}", *boxed);    println!("Pointer to heap data: {:p}", ptr);    println!
("Address of Box on stack: {:p}", &boxed);    // The Box itself (on stack)
contains just the pointer    // We can see this by examining the raw bytes    let
box_bytes: &[u8] = unsafe {        std::slice::from_raw_parts(            &boxed
as *const Box<u64> as *const u8,            mem::size_of::<Box<u64>>()        )
};    println!("\nBox bytes on stack (the pointer itself):");    print!("  ");
for byte in box_bytes {        print!("{:02X} ", byte);    }    println!("\n→
```

```
This is the heap address in little-endian\n");}/// Demonstrates Drop behavior with a custom typefn explore_drop_behavior() {    println!("--- 5. Drop Behavior ---\n");    struct Noisy {        id: u32,    }    impl Drop for Noisy {fn drop(&mut self) {            println!("  Dropping Noisy({})", self.id);}    }    println!("Creating Box<Noisy>...");    {        let boxed = Box::new(Noisy { id: 1 });        println!("  Noisy({}) is alive", boxed.id);println!("Leaving scope...");    }    println!("After scope ends.\n");    // Nested boxes - drop order is important    println!("Creating nested boxes...");{        let outer = Box::new(Noisy { id: 10 });        let inner = Box::new(Noisy { id: 20 });        println!("  Both alive: outer={}, inner={}", outer.id, inner.id);        println!("Leaving scope (inner dropped first - LIFO)...");    }    println!("After scope ends.\n");    // Box containing another Box    println!("Creating Box<Box<Noisy>>...");    {        let nested: Box<Box<Noisy>> = Box::new(Box::new(Noisy { id: 100 }));        println!("  Nested Noisy is alive: {}", nested.id);        println!("Leaving scope (drops cascade from outer to inner)...");    }    println!("After scope ends.\n");}/// Demonstrates raw pointer conversion (into_raw / from_raw)fn explore_raw_pointers() {    println!("--- 6. Raw Pointer Round-Trip ---\n");// Convert Box to raw pointer    let boxed = Box::new(String::from("Hello, heap!"));    println!("Original Box contains: \"{}\"", boxed);    let raw_ptr: *mut String = Box::into_raw(boxed);    println!("Converted to raw pointer: {:p}", raw_ptr);    println!("→ Box is consumed, we now own the raw pointer");println!("→ No destructor ran yet - memory is still allocated");    // We can use the raw pointer (carefully!)    unsafe {        println!("Dereferencing raw pointer: \"{}\"", *raw_ptr);        // Modify through raw pointer        (*raw_ptr).push_str(" Modified!");        println!("After modification: \"{}\"", *raw_ptr);    }    // Convert back to Box - this is how we ensure cleanup    let recovered_box = unsafe { Box::from_raw(raw_ptr) };    println!("Recovered Box contains: \"{}\"", recovered_box);    println!("→ Box will deallocate when dropped");    // Manual deallocation example (educational - don't do this normally!)    println!("\nManual allocation/deallocation:");    unsafe {use std::alloc::{alloc, dealloc};        let layout = Layout::new::<u64>();println!("  Allocating {} bytes with align {}", layout.size(), layout.align());let ptr = alloc(layout) as *mut u64;        if ptr.is_null() {            panic!("Allocation failed!");        }        println!("  Allocated at: {:p}", ptr);// Write to the memory        ptr.write(0x1234_5678_9ABC_DEF0);        println!("  Wrote value: 0x{:016X}", ptr.read());        // Deallocate - must use same layout!        dealloc(ptr as *mut u8, layout);        println!("  Deallocated successfully");    }    println!();}// Bonus: Recursive data structure that requires Box#[allow(dead_code)]enum List<T> {    Cons(T, Box<List<T>>),    Nil,}#[allow(dead_code)]impl<T> List<T> {    fn new() -> Self {        List::Nil    }fn prepend(self, elem: T) -> Self {        List::Cons(elem, Box::new(self))    }fn len(&self) -> usize {        match self {            List::Nil => 0,List::Cons(_, tail) => 1 + tail.len(),        }    }}
```

vec_exploration.rs

```rust
//! Exploring Vec and RawVec Concepts
//!
//! This program demonstrates the allocation mechanics of Vec:
//! - Growth strategies and capacity changes
//! - Zero-sized type handling
//! - Drop behavior for collections
//! - Memory layout inspection

use std::alloc::Layout;
use std::mem;

fn main() {
    println!("=== Rust Vec Exploration ===\n");

    // Section 1: Vec memory layout
    explore_vec_layout();

    // Section 2: Growth strategy observation
    explore_growth_strategy();

    // Section 3: Zero-sized types
    explore_zst_vec();

    // Section 4: Drop behavior
    explore_drop_behavior();

    // Section 5: Capacity vs Length
    explore_capacity_vs_length();

    // Section 6: Initial allocation sizes
    explore_min_non_zero_cap();
}

/// Demonstrates Vec's memory layout on stack and heap
fn explore_vec_layout() {
    println!("--- 1. Vec Memory Layout ---\n");

    // Vec is always 24 bytes on 64-bit (ptr + len + cap)
    println!("Vec<i32> stack size: {} bytes", mem::size_of::<Vec<i32>>());
    println!("Vec<u8> stack size: {} bytes", mem::size_of::<Vec<u8>>());
    println!("Vec<[u8; 1024]> stack size: {} bytes", mem::size_of::<Vec<[u8;
1024]>>());
    println!("→ Vec is always 24 bytes on stack regardless of T\n");

    // The heap allocation depends on capacity and element size
    let v: Vec<u64> = Vec::with_capacity(10);
    println!("Vec<u64> with capacity 10:");
    println!("  Stack: {} bytes", mem::size_of::<Vec<u64>>());
    println!("  Heap: {} bytes (cap × size_of::<u64>)", v.capacity() *
mem::size_of::<u64>());

    // Pointer inspection
    let v = vec![1u64, 2, 3, 4];
    println!("\nVec<u64> = [1, 2, 3, 4]:");
```

```
    println!("  Pointer to heap: {:p}", v.as_ptr());    println!("  Length: {}", v.len());    println!("  Capacity: {}", v.capacity());    println!();}/// Demonstrates the doubling growth strategyfn explore_growth_strategy() {    println!("--- 2. Growth Strategy Observation ---\n");    let mut v: Vec<u64> = Vec::new();    let mut last_cap = 0;    println!("Pushing elements and tracking capacity changes:\n");    println!("{:>6} {:>10} {:>10} {:>15}", "Push#", "Length", "Capacity", "Reallocated?");    println!("{}", "-".repeat(45));    for i in 1..=33 {        v.push(i);        let reallocated = if v.capacity() != last_cap {            last_cap = v.capacity();            "YES"        } else {            ""        };        // Only print on reallocation or first few        if !reallocated.is_empty() || i <= 4 {            println!("{:>6} {:>10} {:>10} {:>15}", i, v.len(), v.capacity(), reallocated);        }    }    println!("\n→ Capacity doubles: 4 → 8 → 16 → 32 → 64");    println!("→ Only ~5 reallocations for 33 elements (amortized O(1) push)\n");}/// Zero-sized types get special treatmentfn explore_zst_vec() {    println!("--- 3. Zero-Sized Type Vec ---\n");    // Unit type Vec    let mut v: Vec<()> = Vec::new();    println!("Vec<()>::new():");    println!("  Length: {}", v.len());    println!("  Capacity: {} (usize::MAX for ZST)", v.capacity());    // Push a million units - no allocation!    for _ in 0..1_000_000 {        v.push(());    }    println!("\nAfter pushing 1,000,000 units:");    println!("  Length: {}", v.len());    println!("  Capacity: {} (still usize::MAX)", v.capacity());    println!("  Heap memory used: 0 bytes");    // Empty struct is also ZST    struct Empty;    let layout = Layout::new::<Empty>();    println!("\nLayout for Empty struct: size={}, align={}", layout.size(), layout.align());    let v_empty: Vec<Empty> = vec![Empty, Empty, Empty];    println!("Vec<Empty> with 3 elements: cap = {}", v_empty.capacity());    println!();}/// Drop behavior for collectionsfn explore_drop_behavior() {    println!("--- 4. Collection Drop Behavior ---\n");    struct Noisy(u32);    impl Drop for Noisy {        fn drop(&mut self) {            println!("    Dropping Noisy({})", self.0);        }    }    println!("Creating Vec<Noisy> with 3 elements...");    {        let v = vec![Noisy(1), Noisy(2), Noisy(3)];        println!("  Vec created with len={}, cap={}", v.len(), v.capacity());        println!("  Dropping Vec...");    }    println!("  → Elements dropped in order, then memory freed\n");    // Pop doesn't drop - the value is moved out    println!("Pop behavior:");    {        let mut v = vec![Noisy(10), Noisy(20)];        println!("  Created Vec with Noisy(10), Noisy(20)");        let popped = v.pop();        println!("  After pop: len={}, popped value exists", v.len());        println!("  Dropping popped value...");        drop(popped);        println!("  Dropping remaining Vec...");    }    println!();}/// Capacity vs Length distinctionfn explore_capacity_vs_length() {    println!("--- 5. Capacity vs Length ---\n");    // with_capacity allocates but doesn't initialize    let mut v: Vec<i32> = Vec::with_capacity(10);    println!("Vec::with_capacity(10):");    println!("  Length: {} (no elements yet)", v.len());    println!("  Capacity: {} (space allocated)", v.capacity());    // Push some elements    v.extend([1, 2, 3]);    println!("\nAfter pushing 3 elements:");    println!("  Length: {}", v.len());    println!("  Capacity: {} (unchanged)", v.capacity());    // Pop doesn't reduce capacity    v.pop();    v.pop();    println!("\nAfter popping 2 elements:");    println!("  Length: {}", v.len());    println!("  Capacity: {} (still 10!)", v.capacity());    // shrink_to_fit releases unused capacity    v.shrink_to_fit();    println!("\nAfter shrink_to_fit:");    println!("  Length: {}", v.len());    println!("  Capacity: {} (shrunk to fit)", v.capacity());    // Clear doesn't deallocate    v.extend([1, 2, 3, 4, 5]);    let cap_before = v.capacity();    v.clear();    println!("\nAfter clear:");    println!("  Length: {} (all elements gone)", v.len());    println!("  Capacity: {} (memory still allocated)", v.capacity());
```

```rust
assert_eq!(v.capacity(), cap_before);    println!();}/// Demonstrates min_non_zero_cap behaviorfn explore_min_non_zero_cap() {    println!("--- 6. Initial Allocation Sizes ---\n");    // The stdlib uses min_non_zero_cap to avoid tiny allocations:    // - 8 for size 1 (bytes)    // - 4 for size <= 1024    // - 1 for larger elements    println!("First push allocation (min_non_zero_cap behavior):\n");    // u8: starts at capacity 8    let mut v_u8: Vec<u8> = Vec::new();    v_u8.push(1);    println!("Vec<u8>: first push → capacity {}", v_u8.capacity());    println!("  (min 8 because heap allocators round up small requests)");    // u64: starts at capacity 4    let mut v_u64: Vec<u64> = Vec::new();    v_u64.push(1);    println!("\nVec<u64>: first push → capacity {}", v_u64.capacity());    println!("  (min 4 for moderate-sized elements <= 1KB)");    // Large element: starts at capacity 1    let mut v_large: Vec<[u8; 4096]> = Vec::new();    v_large.push([0; 4096]);    println!("\nVec<[u8; 4096]>: first push → capacity {}", v_large.capacity());    println!("  (min 1 for large elements to avoid wasting space)");    println!("\n→ This optimization reduces wasted memory for different use cases\n");}// Bonus: A minimal Vec implementation for educational purposes#[allow(dead_code)]mod simple_vec {    use std::alloc::{alloc, dealloc, realloc, Layout};    use std::ptr::{self, NonNull};    use std::mem;    pub struct SimpleVec<T> {        ptr: NonNull<T>,        len: usize,        cap: usize,    }    impl<T> SimpleVec<T> {        pub fn new() -> Self {            assert!(mem::size_of::<T>() != 0, "ZST not supported");            SimpleVec {                ptr: NonNull::dangling(),                len: 0,                cap: 0,            }        }        pub fn push(&mut self, value: T) {            if self.len == self.cap {                self.grow();            }            unsafe {                ptr::write(self.ptr.as_ptr().add(self.len), value);            }            self.len += 1;        }        pub fn pop(&mut self) -> Option<T> {            if self.len == 0 {                None            } else {                self.len -= 1;                unsafe { Some(ptr::read(self.ptr.as_ptr().add(self.len))) }            }        }        fn grow(&mut self) {            let new_cap = if self.cap == 0 { 4 } else { self.cap * 2 };            let new_layout = Layout::array::<T>(new_cap).unwrap();            let new_ptr = if self.cap == 0 {                unsafe { alloc(new_layout) }            } else {                let old_layout = Layout::array::<T>(self.cap).unwrap();                unsafe {                    realloc(self.ptr.as_ptr() as *mut u8, old_layout, new_layout.size())                }            };            self.ptr = NonNull::new(new_ptr as *mut T).expect("allocation failed");            self.cap = new_cap;        }        pub fn len(&self) -> usize { self.len }        pub fn capacity(&self) -> usize { self.cap }    }    impl<T> Drop for SimpleVec<T> {        fn drop(&mut self) {            if self.cap > 0 {                // Drop elements first                for i in 0..self.len {                    unsafe {                        ptr::drop_in_place(self.ptr.as_ptr().add(i));                    }                }                // Then deallocate                let layout = Layout::array::<T>(self.cap).unwrap();                unsafe { dealloc(self.ptr.as_ptr() as *mut u8, layout); }            }        }    }}
```

# Rust Syntax, Keywords, and Patterns Reference

A comprehensive reference for Rust's syntactic elements, reserved keywords, and idiomatic patterns.

---

## Part 1: Keywords

Rust reserves keywords for current and future use. They cannot be used as identifiers (variable names, function names, etc.).

### Currently Used Keywords

| Keyword | Purpose |
|---|---|
| `as` | Type casting or renaming imports |
| `async` | Declare an asynchronous function or block |
| `await` | Suspend execution until an async result is ready |
| `break` | Exit a loop immediately |
| `const` | Define compile-time constants or constant pointers |
| `continue` | Skip to the next loop iteration |
| `crate` | Refer to the current crate root |
| `dyn` | Dynamic dispatch for trait objects |
| `else` | Fallback branch for `if` expressions |

| | |
|---|---|
| `enum` | Define an enumeration type |
| `extern` | Link external code or specify ABI |
| `false` | Boolean literal |
| `fn` | Define a function |
| `for` | Loop over an iterator |
| `if` | Conditional branching |
| `impl` | Implement inherent or trait methods |
| `in` | Part of `for` loop syntax |
| `let` | Bind a value to a variable |
| `loop` | Infinite loop construct |
| `match` | Pattern matching expression |
| `mod` | Define a module |
| `move` | Force a closure to take ownership |
| `mut` | Declare mutable binding or reference |
| `pub` | Make an item public |
| `ref` | Bind by reference in patterns |
| `return` | Return from a function |
| `Self` | Type alias for the implementing type |
| `self` | Current module or method receiver |

| | |
|---|---|
| `static` | Global variable or `'static` lifetime |
| `struct` | Define a structure type |
| `super` | Parent module |
| `trait` | Define a trait |
| `true` | Boolean literal |
| `type` | Define a type alias |
| `unsafe` | Mark code that bypasses safety checks |
| `use` | Bring paths into scope |
| `where` | Specify trait bounds on generics |
| `while` | Conditional loop |

### Reserved for Future Use

These keywords are reserved but have no current functionality:

```
abstract   become   box   do   final   macro   override
priv   try   typeof   unsized   virtual   yield
```

## Part 2: Syntax Elements

### Variable Bindings

```
let x = 5;              // Immutable binding
let mut y = 10;         // Mutable binding
let z: i32 = 15;        // Explicit type annotation
let (a, b) = (1, 2);    // Destructuring bind
let _ = unused_value(); // Discard with underscore
```

## Type Annotations

```rust
let integer: i32 = 42;
let float: f64 = 3.14;
let boolean: bool = true;
let character: char = 'R';
let string_slice: &str = "hello";
let owned_string: String = String::from("world");
let array: [i32; 3] = [1, 2, 3];
let tuple: (i32, f64, char) = (42, 3.14, 'x');
let unit: () = ();
```

## Functions

```rust
// Basic function
fn add(a: i32, b: i32) -> i32 {
    a + b  // Implicit return (no semicolon)
}

// No return value (returns unit type)
fn print_value(x: i32) {
    println!("{}", x);
}

// Early return
fn early_return(x: i32) -> i32 {
    if x < 0 {
        return 0;  // Explicit return
    }
    x * 2
}

// Generic function
fn largest<T: PartialOrd>(a: T, b: T) -> T {
    if a > b { a } else { b }
}

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

## Control Flow

```
// if-else expression (returns a value)
let result = if condition { value_a } else { value_b };

// if-else chain
if x < 0 {
    println!("negative");
} else if x == 0 {
    println!("zero");
} else {
    println!("positive");
}

// if-let for single pattern matching
if let Some(value) = optional {
    println!("Got: {}", value);
}

// while loop
while condition {
    // body
}

// while-let
while let Some(item) = iterator.next() {
    process(item);
}

// for loop (iterates over anything implementing IntoIterator)
for item in collection {
    process(item);
}

// for with range
for i in 0..10 {      // 0 to 9
    println!("{}", i);
}

for i in 0..=10 {     // 0 to 10 inclusive
    println!("{}", i);
}

// loop (infinite, must break)
loop {
    if done {
        break;
    }
}

// loop with return value
let result = loop {
    if condition {
        break 42;  // Returns 42
    }
};
```

```
// labeled loops'outer: for i in 0..10 {    'inner: for j in 0..10 {        if
condition {            break 'outer;  // Breaks outer loop        }    }}
```

## Match Expressions

```
let x = 5;

match x {
    1 => println!("one"),
    2 | 3 => println!("two or three"),  // Multiple patterns
    4..=9 => println!("four through nine"),  // Range pattern
    n if n > 100 => println!("big: {}", n),  // Guard
    _ => println!("something else"),  // Wildcard
}

// Match with destructuring
match point {
    (0, 0) => println!("origin"),
    (x, 0) => println!("on x-axis at {}", x),
    (0, y) => println!("on y-axis at {}", y),
    (x, y) => println!("at ({}, {})", x, y),
}

// Match returns a value
let description = match value {
    0 => "zero",
    1 => "one",
    _ => "many",
};
```

## Structs

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

// Tuple struct
struct Color(u8, u8, u8);

// Unit struct
struct Marker;

// Instantiation
let p = Point { x: 1.0, y: 2.0 };
let c = Color(255, 128, 0);
let m = Marker;

// Field shorthand (when variable name matches field name)
let x = 1.0;
let y = 2.0;
let p = Point { x, y };

// Struct update syntax
let p2 = Point { x: 3.0, ..p };  // y comes from p
```

## Enums

```rust
// Simple enum
enum Direction {
    North,
    South,
    East,
    West,
}

// Enum with data
enum Message {
    Quit,                       // No data
    Move { x: i32, y: i32 },    // Named fields
    Write(String),              // Tuple variant
    ChangeColor(u8, u8, u8),    // Multiple values
}

// Usage
let msg = Message::Move { x: 10, y: 20 };

match msg {
    Message::Quit => println!("quit"),
    Message::Move { x, y } => println!("move to ({}, {})", x, y),
    Message::Write(text) => println!("write: {}", text),
    Message::ChangeColor(r, g, b) => println!("color: {},{},{}", r, g, b),
}
```

## Impl Blocks

```rust
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    // Associated function (no self) - called with ::
    fn new(width: u32, height: u32) -> Self {
        Self { width, height }
    }

    // Method (takes &self) - called with .
    fn area(&self) -> u32 {
        self.width * self.height
    }

    // Mutable method
    fn scale(&mut self, factor: u32) {
        self.width *= factor;
        self.height *= factor;
    }

    // Consuming method (takes ownership)
    fn into_square(self) -> Rectangle {
        let side = self.width.max(self.height);
        Rectangle { width: side, height: side }
    }
}

// Usage
let r = Rectangle::new(10, 20);  // Associated function
let area = r.area();             // Method
```

## Traits

```rust
// Define a trait
trait Summary {
    // Required method
    fn summarize(&self) -> String;

    // Default implementation
    fn preview(&self) -> String {
        format!("Read more: {}", self.summarize())
    }
}

// Implement for a type
impl Summary for Article {
    fn summarize(&self) -> String {
        format!("{} by {}", self.title, self.author)
    }
}

// Trait bounds
fn notify<T: Summary>(item: &T) {
    println!("News: {}", item.summarize());
}

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

// Where clause (cleaner for complex bounds)
fn complex<T, U>(t: &T, u: &U) -> i32
where
    T: Summary + Clone,
    U: Debug + PartialOrd,
{
    // ...
}

// impl Trait syntax
fn make_summary() -> impl Summary {
    Article { /* ... */ }
}
```

## Generics

```rust
// Generic struct
struct Container<T> {
    value: T,
}

// Generic enum
enum Option<T> {
    Some(T),
    None,
}

// Multiple type parameters
struct Pair<T, U> {
    first: T,
    second: U,
}

// Generic impl
impl<T> Container<T> {
    fn new(value: T) -> Self {
        Self { value }
    }
}

// Impl for specific type
impl Container<i32> {
    fn is_positive(&self) -> bool {
        self.value > 0
    }
}
```

## Lifetimes

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

// Struct with lifetime
struct Excerpt<'a> {
    part: &'a str,
}

// Multiple lifetimes
fn complex<'a, 'b>(x: &'a str, y: &'b str) -> &'a str {
    x
}

// Static lifetime
let s: &'static str = "I live forever";
```

## Closures

```rust
// Basic closure
let add_one = |x| x + 1;

// With type annotations
let add_one: fn(i32) -> i32 = |x: i32| -> i32 { x + 1 };

// Multi-line closure
let process = |x| {
    let y = x * 2;
    y + 1
};

// Capturing environment
let factor = 2;
let multiply = |x| x * factor;  // Borrows factor

// move closure (takes ownership)
let data = vec![1, 2, 3];
let closure = move || {
    println!("{:?}", data);  // Owns data now
};
```

## Error Handling

```rust
// Result type
fn divide(a: f64, b: f64) -> Result<f64, String> {
    if b == 0.0 {
        Err(String::from("division by zero"))
    } else {
        Ok(a / b)
    }
}

// ? operator (early return on error)
fn process() -> Result<i32, Error> {
    let file = File::open("data.txt")?;  // Returns Err if fails
    let data = read_data(&file)?;
    Ok(data.len() as i32)
}

// unwrap and expect
let value = result.unwrap();           // Panics on Err
let value = result.expect("failed!");  // Panics with message

// Option handling
let maybe = Some(5);
let value = maybe.unwrap_or(0);        // Default on None
let value = maybe.unwrap_or_else(|| compute_default());
```

## References and Borrowing

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

let r1 = &s;          // Immutable borrow
let r2 = &s;          // Multiple immutable borrows OK

let r3 = &mut s;      // Mutable borrow (only one allowed)

// Dereferencing
let x = 5;
let r = &x;
assert_eq!(*r, 5);   // Dereference with *
```

## Smart Pointers

```rust
use std::rc::Rc;
use std::cell::RefCell;

// Box: heap allocation
let boxed: Box<i32> = Box::new(42);

// Rc: reference counting
let shared: Rc<String> = Rc::new(String::from("shared"));
let clone = Rc::clone(&shared);

// RefCell: interior mutability
let cell: RefCell<i32> = RefCell::new(5);
*cell.borrow_mut() += 1;
```

## Attributes

```rust
// Function attributes
#[inline]
#[must_use]
fn important() -> i32 { 42 }

// Derive macros
#[derive(Debug, Clone, PartialEq)]
struct Point { x: i32, y: i32 }

// Conditional compilation
#[cfg(target_os = "linux")]
fn linux_only() { /* ... */ }

// Test attribute
#[test]
fn test_something() {
    assert_eq!(2 + 2, 4);
}

// Allow/deny lints
#[allow(dead_code)]
#[deny(unused_variables)]
fn example() { /* ... */ }

// Documentation
/// This is a doc comment for the following item
///
/// # Examples
/// ```
/// let x = my_function();
/// ```
fn my_function() { }
```

### Modules

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

    fn private_helper() { /* ... */ }

    pub mod advanced {
        pub fn sqrt(x: f64) -> f64 { x.sqrt() }
    }
}

// Use module items
use math::add;
use math::advanced::sqrt;

// Re-exports
pub use math::add;

// Glob import (generally discouraged)
use std::collections::*;

// Renaming
use std::io::Result as IoResult;
```

## Part 3: Patterns

Patterns are used in `match`, `if let`, `while let`, `let`, and function parameters for destructuring and matching values.

### Literal Patterns

Match exact values.

```
let x = 1;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    _ => println!("other"),
}
```

## Variable Patterns

Bind matched value to a name.

```
let x = 5;

match x {
    n => println!("got {}", n),  // n binds to x's value
}
```

## Wildcard Pattern

Matches anything, discards value.

```
let pair = (1, 2);

match pair {
    (0, _) => println!("first is zero"),
    (_, 0) => println!("second is zero"),
    (_, _) => println!("neither is zero"),
}
```

## Rest Pattern

Match remaining elements.

```
let numbers = (1, 2, 3, 4, 5);

match numbers {
    (first, .., last) => println!("{} to {}", first, last),
}

let arr = [1, 2, 3, 4, 5];
match arr {
    [first, second, ..] => println!("starts with {}, {}", first, second),
}
```

## Range Patterns

Match a range of values.

```
let x = 5;

match x {
    1..=5 => println!("one through five"),
    6..=10 => println!("six through ten"),
    _ => println!("something else"),
}

// Also works with char
match c {
    'a'..='z' => println!("lowercase"),
    'A'..='Z' => println!("uppercase"),
    _ => println!("other"),
}
```

## Or Patterns

Match multiple alternatives.

```
let x = 2;

match x {
    1 | 2 | 3 => println!("one, two, or three"),
    4 | 5 => println!("four or five"),
    _ => println!("other"),
}
```

## Struct Patterns

Destructure structs.

```
struct Point { x: i32, y: i32 }

let p = Point { x: 0, y: 7 };

match p {
    Point { x: 0, y } => println!("on y-axis at {}", y),
    Point { x, y: 0 } => println!("on x-axis at {}", x),
    Point { x, y } => println!("at ({}, {})", x, y),
}

// Shorthand when variable name matches field
let Point { x, y } = p;

// Ignore fields with ..
match p {
    Point { x, .. } => println!("x is {}", x),
}
```

## Tuple Patterns

Destructure tuples.

```
let tuple = (1, "hello", 3.14);

let (a, b, c) = tuple;

match tuple {
    (1, _, _) => println!("starts with 1"),
    (_, "hello", _) => println!("says hello"),
    _ => println!("other"),
}
```

## Enum Patterns

Match and destructure enum variants.

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}

let msg = Message::Move { x: 10, y: 20 };

match msg {
    Message::Quit => println!("quit"),
    Message::Move { x, y } => println!("move to ({}, {})", x, y),
    Message::Write(text) => println!("write: {}", text),
    Message::ChangeColor(r, g, b) => println!("color: {}/{}/{}", r, g, b),
}
```

## Reference Patterns

Match references and bind by reference.

```
let reference = &4;

match reference {
    &val => println!("got value {}", val),  // Dereference in pattern
}

// Bind by reference (avoid moving)
let value = String::from("hello");

match value {
    ref r => println!("got reference: {}", r),  // r is &String
}

// Mutable reference binding
let mut value = 5;
match value {
    ref mut m => *m += 10,
}
```

## Guards

Add conditions to pattern arms.

```
let pair = (2, -2);

match pair {
    (x, y) if x == y => println!("equal"),
    (x, y) if x + y == 0 => println!("sum to zero"),
    (x, _) if x % 2 == 0 => println!("first is even"),
    _ => println!("no match"),
}
```

## @ Bindings

Bind a name while also testing a pattern.

```
let x = 5;

match x {
    n @ 1..=5 => println!("got {} in range 1-5", n),
    n @ 6..=10 => println!("got {} in range 6-10", n),
    n => println!("got {} outside ranges", n),
}

// With enums
enum Message {
    Hello { id: i32 },
}

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

## Slice Patterns

Match slices and arrays.

```
let arr = [1, 2, 3];

match arr {
    [1, _, _] => println!("starts with 1"),
    [_, 2, _] => println!("2 in middle"),
    [.., 3] => println!("ends with 3"),
    _ => println!("other"),
}

// With variable binding
match arr {
    [first, rest @ ..] => println!("first: {}, rest: {:?}", first, rest),
}

// Exact length matching
match slice {
    [] => println!("empty"),
    [single] => println!("one element: {}", single),
    [first, second] => println!("two elements"),
    [first, middle @ .., last] => println!("many elements"),
}
```

### Box Patterns (Nightly)

Match through Box pointers.

```
#![feature(box_patterns)]

let boxed = Box::new(5);

match boxed {
    box 0 => println!("zero"),
    box n => println!("got {}", n),
}
```

---

## Part 4: Common Idioms

### The Builder Pattern

Construct complex objects step by step.

```
struct ServerConfig {
    host: String,
    port: u16,
    max_connections: u32,
}

struct ServerConfigBuilder {
    host: String,
    port: u16,
    max_connections: u32,
}

impl ServerConfigBuilder {
    fn new() -> Self {
        Self {
            host: String::from("localhost"),
            port: 8080,
            max_connections: 100,
        }
    }

    fn host(mut self, host: &str) -> Self {
        self.host = host.to_string();
        self
    }

    fn port(mut self, port: u16) -> Self {
        self.port = port;
        self
    }

    fn build(self) -> ServerConfig {
        ServerConfig {
            host: self.host,
            port: self.port,
            max_connections: self.max_connections,
        }
    }
}

// Usage
let config = ServerConfigBuilder::new()
    .host("0.0.0.0")
    .port(3000)
    .build();
```

## The Newtype Pattern

Wrap a type to provide distinct semantics.

```
struct Meters(f64);
struct Feet(f64);

impl Meters {
    fn to_feet(&self) -> Feet {
        Feet(self.0 * 3.28084)
    }
}

// Prevents mixing up units at compile time
fn calculate_area(width: Meters, height: Meters) -> f64 {
    width.0 * height.0
}
```

## The Typestate Pattern

Use the type system to enforce valid state transitions.

```
struct Locked;
struct Unlocked;

struct Door<State> {
    _state: std::marker::PhantomData<State>,
}

impl Door<Locked> {
    fn unlock(self) -> Door<Unlocked> {
        println!("Unlocking...");
        Door { _state: std::marker::PhantomData }
    }
}

impl Door<Unlocked> {
    fn lock(self) -> Door<Locked> {
        println!("Locking...");
        Door { _state: std::marker::PhantomData }
    }

    fn open(&self) {
        println!("Opening door");
    }
}

// Can only open an unlocked door - enforced at compile time
let door: Door<Locked> = Door { _state: std::marker::PhantomData };
let door = door.unlock();
door.open();
```

## RAII (Resource Acquisition Is Initialization)

Resources are tied to object lifetime.

```
struct FileHandle {
    path: String,
}

impl FileHandle {
    fn new(path: &str) -> Self {
        println!("Opening file: {}", path);
        Self { path: path.to_string() }
    }
}

impl Drop for FileHandle {
    fn drop(&mut self) {
        println!("Closing file: {}", self.path);
    }
}

// File automatically closed when handle goes out of scope
{
    let handle = FileHandle::new("data.txt");
    // use handle...
}  // Automatically calls drop() here
```

## The Iterator Pattern

Process sequences lazily.

```rust
let numbers = vec![1, 2, 3, 4, 5];

// Chained iterator methods
let result: Vec<i32> = numbers
    .iter()
    .filter(|&x| x % 2 == 0)
    .map(|x| x * x)
    .collect();

// Implementing Iterator
struct Counter {
    count: u32,
    max: u32,
}

impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        if self.count < self.max {
            self.count += 1;
            Some(self.count)
        } else {
            None
        }
    }
}
```

## Option/Result Combinators

Chain operations without explicit matching.

```rust
// Option combinators
let maybe_number: Option<i32> = Some(5);

let result = maybe_number
    .map(|n| n * 2)           // Some(10)
    .filter(|&n| n > 5)       // Some(10)
    .and_then(|n| Some(n + 1)) // Some(11)
    .unwrap_or(0);            // 11

// Result combinators
let result: Result<i32, &str> = Ok(5);

let final_value = result
    .map(|n| n * 2)
    .map_err(|e| format!("Error: {}", e))
    .and_then(|n| if n > 0 { Ok(n) } else { Err("negative".into()) })
    .unwrap_or_default();
```

## Default Trait Pattern

Provide sensible defaults.

```
#[derive(Default)]
struct Config {
    debug: bool,
    timeout: u32,
    name: String,
}

// Custom Default implementation
impl Default for Config {
    fn default() -> Self {
        Self {
            debug: false,
            timeout: 30,
            name: String::from("default"),
        }
    }
}

// Usage
let config = Config::default();
let custom = Config { timeout: 60, ..Default::default() };
```

## From/Into Conversions

Type-safe conversions.

```rust
struct Celsius(f64);
struct Fahrenheit(f64);

impl From<Celsius> for Fahrenheit {
    fn from(c: Celsius) -> Self {
        Fahrenheit(c.0 * 9.0 / 5.0 + 32.0)
    }
}

// From automatically provides Into
let c = Celsius(100.0);
let f: Fahrenheit = c.into();  // Uses Into, which uses From

// Common pattern with strings
impl From<&str> for MyType {
    fn from(s: &str) -> Self {
        MyType { name: s.to_string() }
    }
}
```

### Deref Coercion Pattern

Smart pointers that act like references.

```rust
use std::ops::Deref;

struct MyBox<T>(T);

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

// Now MyBox<String> can be used where &str is expected
fn print_str(s: &str) {
    println!("{}", s);
}

let boxed = MyBox(String::from("hello"));
print_str(&boxed);  // Deref coercion: &MyBox<String> -> &String -> &str
```

## Quick Reference: Operators

| Operator | Meaning |
| --- | --- |
| `::` | Path separator (modules, types) |
| `->` | Function return type |
| `=>` | Match arm |
| `..` | Range (exclusive end) |
| `..=` | Range (inclusive end) |
| `..` | Struct update / rest pattern |
| `?` | Error propagation |
| `&` | Borrow / reference type |
| `*` | Dereference / raw pointer type |
| `'a` | Lifetime annotation |
| `<>` | Generics |
| `\|` | Closure parameters / or-pattern |
| `_` | Wildcard / ignored binding |
| `@` | Pattern binding |
| `#[]` | Outer attribute |
| `#![]` | Inner attribute |
| `!` | Never type / macro invocation |

*This reference covers Rust 1.70+ syntax and patterns.*

# Rust Terminal Development Cheatsheet

Quick reference for terminal-based Rust development on Ubuntu.

---

## Cargo Basics

| Command | Description |
|---|---|
| `cargo new myproj` | Create new binary project |
| `cargo new mylib --lib` | Create new library project |
| `cargo build` | Build debug |
| `cargo build --release` | Build release |
| `cargo run` | Build and run |
| `cargo run --release` | Build and run release |
| `cargo check` | Fast syntax/type check (no codegen) |
| `cargo test` | Run tests |
| `cargo test -- --nocapture` | Tests with stdout |
| `cargo test test_name` | Run specific test |
| `cargo doc --open` | Generate and open docs |
| `cargo clean` | Remove target directory |

| | |
|---|---|
| `cargo update` | Update dependencies |

## Code Quality

| Command | Description |
|---|---|
| `cargo fmt` | Format code |
| `cargo fmt --check` | Check formatting (CI) |
| `cargo clippy` | Run linter |
| `cargo clippy -- -D warnings` | Clippy with warnings as errors |
| `cargo clippy --fix` | Auto-fix lint issues |
| `cargo fix` | Auto-fix compiler warnings |

## Fast Feedback Loop

| Command | Description |
|---|---|
| `cargo watch -c -x check` | Auto-check on save |
| `cargo watch -c -x test` | Auto-test on save |
| `cargo watch -c -x 'run -- args'` | Auto-run on save |
| `bacon` | TUI for cargo watch |
| `bacon test` | Bacon in test mode |

| | |
|---|---|
| `bacon clippy` | Bacon in clippy mode |
| `cargo nextest run` | Fast parallel test runner |

## Toolchain Management

| Command | Description |
|---|---|
| `rustup update` | Update all toolchains |
| `rustup default stable` | Set default toolchain |
| `rustup toolchain list` | List installed toolchains |
| `rustup component add rust-src` | Add component |
| `rustup component list --installed` | List components |
| `cargo +nightly build` | Use nightly for one command |
| `rustup run nightly cargo build` | Alternative syntax |
| `rustc --print sysroot` | Print toolchain location |
| `rustup doc` | Open local Rust documentation |

## Compiler Inspection (Source → Binary Pipeline)

### Macro Expansion

```
cargo expand                    # Full expansion
cargo expand main               # Expand main only
cargo expand path::to::module   # Expand specific module
```

### HIR (High-level IR) [Nightly]

```
cargo +nightly rustc -- -Zunpretty=hir
cargo +nightly rustc -- -Zunpretty=hir,typed    # With type annotations
```

### MIR (Mid-level IR) [Nightly]

```
cargo +nightly rustc -- -Zunpretty=mir          # Print MIR
cargo +nightly rustc -- -Zdump-mir=all          # Dump all MIR passes
cargo +nightly rustc -- -Zdump-mir=all -Zdump-mir-dir=./mir_dump
cargo +nightly rustc -- -Zmir-opt-level=0       # Unoptimized MIR
```

### LLVM IR

```
# Via rustc
cargo rustc --release -- --emit=llvm-ir -C debuginfo=0 -C codegen-units=1
# Output: target/release/deps/*.ll

# Via cargo-show-asm
cargo asm --lib --llvm function_name
```

### Assembly

```
# Via rustc (Intel syntax)
cargo rustc --release -- --emit=asm -C llvm-args=-x86-asm-syntax=intel
# Output: target/release/deps/*.s

# Via cargo-show-asm (recommended)
cargo asm --lib                     # List available functions
cargo asm --lib function_name       # Show assembly for function
cargo asm --lib --rust func_name    # Interleaved with Rust source
cargo asm --lib --mir func_name     # MIR via cargo-show-asm
```

### Type and Memory Layout [Nightly]

```
RUSTFLAGS="-Zprint-type-sizes" cargo +nightly build 2>&1 | grep print-type-size
```

### Binary Size Analysis

```
cargo bloat --release              # Function sizes
cargo bloat --release -n 20        # Top 20 functions
cargo bloat --release --crates     # Size by crate
```

### All --emit Options

```
--emit=asm          # Assembly
--emit=llvm-ir      # LLVM IR
--emit=llvm-bc      # LLVM bitcode
--emit=mir          # MIR
--emit=obj          # Object file
--emit=link         # Final binary (default)
--emit=dep-info     # Dependency info
```

---

## Debugging with rust-gdb

### Starting a Session

```
# Build with debug symbols (default for dev builds)
cargo build

# Start debugger
rust-gdb target/debug/myprogram
rust-gdb --args target/debug/myprogram arg1 arg2
```

### Essential GDB Commands

| Command | Description |
|---|---|
| `r` / `run` | Run program |
| `c` / `continue` | Continue after breakpoint |
| `n` / `next` | Step over (next line) |
| `s` / `step` | Step into function |

| | |
|---|---|
| `finish` | Run until function returns |
| `q` / `quit` | Exit GDB |

## Breakpoints

| Command | Description |
|---|---|
| `b main` | Break at main |
| `b main.rs:42` | Break at file:line |
| `b module::function` | Break at function |
| `b 42` | Break at line in current file |
| `info breakpoints` | List breakpoints |
| `delete 1` | Delete breakpoint #1 |
| `disable 1` | Disable breakpoint |
| `enable 1` | Enable breakpoint |

## Inspecting State

| Command | Description |
|---|---|
| `bt` / `backtrace` | Show call stack |
| `bt full` | Backtrace with locals |
| `frame 3` | Select stack frame |
| `info locals` | Show local variables |

| | |
|---|---|
| `info args` | Show function arguments |
| `p variable` | Print variable |
| `p *pointer` | Dereference pointer |
| `p struct.field` | Print struct field |
| `ptype variable` | Print variable type |

## Memory Inspection

| Command | Description |
|---|---|
| `x/10x $sp` | 10 hex words from stack |
| `x/s ptr` | Print as string |
| `x/i $pc` | Disassemble at PC |
| `info registers` | Show registers |

## TUI Mode

| Command | Description |
|---|---|
| `layout src` | Show source window |
| `layout asm` | Show assembly window |
| `layout split` | Source + assembly |
| `layout regs` | Show registers |
| `tui disable` | Exit TUI mode |

| | |
|---|---|
| `Ctrl-x a` | Toggle TUI mode |

## Binary Inspection (Post-Compilation)

### objdump

```
objdump -d target/debug/myprogram | less         # Disassemble
objdump --demangle -d target/debug/myprogram     # With demangled names
objdump -t target/debug/myprogram                # Symbol table
objdump -h target/debug/myprogram                # Section headers
```

### readelf

```
readelf -h target/debug/myprogram    # ELF header
readelf -S target/debug/myprogram    # Section headers
readelf -s target/debug/myprogram    # Symbol table
readelf -d target/debug/myprogram    # Dynamic section
readelf -l target/debug/myprogram    # Program headers
```

### nm (Symbol Table)

```
nm target/debug/myprogram                    # All symbols
nm --demangle target/debug/myprogram         # Demangled
nm --demangle --size-sort target/debug/myprogram  # Sorted by size
nm -C target/debug/myprogram | grep my_func  # Find function
```

### size

```
size target/debug/myprogram          # Section sizes summary
size -A target/debug/myprogram       # Detailed breakdown
```

### cargo-binutils (Cross-Platform)

```
cargo nm --release                   # nm via cargo
cargo objdump --release -- -d        # objdump via cargo
cargo size --release                 # size via cargo
```

## Standard Library Source

### Location

```
# Print std source path
rustc --print sysroot
# Full path: $(rustc --print sysroot)/lib/rustlib/src/rust/library/

# Directory structure:
# library/
# ├── core/     # #![no_std] basics
# ├── alloc/    # Vec, Box, String
# └── std/      # Full standard library
```

### Searching

```
# Search with ripgrep
rg "fn drop" --type rust $(rustc --print sysroot)/lib/rustlib/src/rust/library/

# Search specific crate
rg "pattern" $(rustc --print sysroot)/lib/rustlib/src/rust/library/alloc/
```

## CI/CD Tools

| Command | Description |
| --- | --- |
| `cargo audit` | Check for vulnerabilities |
| `cargo deny check` | Check dependency policies |
| `cargo tarpaulin` | Generate test coverage |
| `cargo tarpaulin --out Html` | HTML coverage report |

## Environment Variables

| Variable | Description |
|---|---|
| `RUST_BACKTRACE=1` | Enable backtraces |
| `RUST_BACKTRACE=full` | Full backtraces |
| `RUST_LOG=debug` | Log level (with `env_logger`) |
| `RUSTFLAGS="-Z..."` | Pass flags to rustc |
| `CARGO_INCREMENTAL=0` | Disable incremental compilation |

## Vim Quick Reference

| Key | Action |
|---|---|
| `\b` | `:make build` |
| `\r` | `:make run` |
| `\t` | `:make test` |
| `\c` | `:make check` |
| `]q` | Next quickfix error |
| `[q` | Previous quickfix error |
| `:copen` | Open quickfix window |
| `:cclose` | Close quickfix window |

## Shell Aliases (from setup script)

```
# Basic
cb      # cargo build
cbr     # cargo build --release
cc      # cargo check
cr      # cargo run
ct      # cargo test
ctn     # cargo nextest run

# Watch
cw      # cargo watch -c -x check
cwt     # cargo watch -c -x test

# Inspection
mir     # View MIR
hir     # View HIR
casm    # cargo asm --lib
casmr   # cargo asm --lib --rust

# Functions
mir_dump [dir]     # Dump all MIR passes
llvm_ir            # Emit and view LLVM IR
emit_asm           # Emit and view assembly
std_search "query" # Search std library source
rust_stats         # Show environment info
```

## Project Structure Best Practices

```
myproject/
├── .cargo/
│   └── config.toml       # Cargo configuration
├── src/
│   ├── main.rs           # Thin entry point
│   ├── lib.rs            # Application logic
│   └── prelude.rs        # Common re-exports
├── tests/                # Integration tests
├── benches/              # Benchmarks
├── Cargo.toml
├── Cargo.lock
├── rust-toolchain.toml   # Pinned toolchain
├── rustfmt.toml          # Formatting rules
├── clippy.toml           # Linter config
└── README.md
```

## Quick Debugging Workflow

```
# 1. Check for errors fast
cargo check

# 2. Run tests
cargo nextest run

# 3. If crash/panic, get backtrace
RUST_BACKTRACE=1 cargo run

# 4. Debug specific issue
cargo build && rust-gdb target/debug/myprogram
(gdb) b main.rs:42
(gdb) r
(gdb) bt
(gdb) info locals

# 5. Inspect generated code
cargo asm --lib --rust problematic_function
```

## Quick Inspection Workflow

```
# See what macros generate
cargo expand my_module

# See the control flow (MIR)
cargo +nightly rustc -- -Zunpretty=mir 2>&1 | less

# See the assembly
cargo asm --lib --rust my_function

# Check type sizes
RUSTFLAGS="-Zprint-type-sizes" cargo +nightly build 2>&1 | grep MyType
```

# Ideas and Code for TTS for mkdocs

**Native Browser API (no library needed)**

The Web Speech API is built into modern browsers:

```
const utterance = new SpeechSynthesisUtterance("Hello world");
utterance.rate = 1.0;
utterance.pitch = 1.0;
utterance.voice = speechSynthesis.getVoices()[0];
speechSynthesis.speak(utterance);
```

This works in Chrome, Firefox, Safari, and Edge with no dependencies. Voice quality depends on the OS/browser's installed voices.

**Libraries**

- **meSpeak.js** — A pure JavaScript port of eSpeak. Works entirely offline, ~1MB, produces robotic but functional speech. Good for environments where you can't rely on browser voices.
- **speak.js** — Similar Emscripten-compiled eSpeak port.
- **ResponsiveVoice.js** — Commercial wrapper that falls back gracefully across browsers, but requires their service for some features.

**For your Lecture Forge context**, if you're generating audio versions of your lectures, the native `SpeechSynthesis` API would be the simplest starting point. Browser voices have improved significantly — macOS and Windows both include high-quality neural voices now. You'd just need to handle the async nature (voices load lazily) and potentially chunk long text to avoid browser limits.

Want me to show a more complete implementation pattern for longer-form content like lectures?

This is a great addition to your learning workflow! Here's a complete implementation for adding TTS to MkDocs:

## Project Structure

```
docs/
├── javascripts/
│   └── tts-reader.js
├── stylesheets/
│   └── tts-reader.css
└── ...your markdown files...
mkdocs.yml
```

## The TTS Reader Implementation

```
// docs/javascripts/tts-reader.js

class LectureReader {
  constructor() {
    this.synth = window.speechSynthesis;
    this.utterance = null;
    this.chunks = [];
    this.currentChunk = 0;
    this.isPlaying = false;
    this.isPaused = false;
    this.voices = [];
    this.selectedVoice = null;
    this.rate = 1.0;

    // Browser limit safety - chunk at ~3000 chars
    this.chunkSize = 3000;

    this.init();
  }

  init() {
    // Voices load asynchronously in some browsers
    if (this.synth.onvoiceschanged !== undefined) {
      this.synth.onvoiceschanged = () => this.loadVoices();
    }
    this.loadVoices();
    this.createUI();
    this.bindEvents();
  }

  loadVoices() {
    this.voices = this.synth.getVoices();

    // Prefer high-quality voices
    const preferredVoices = [
      'Google US English',
      'Microsoft Zira',
      'Samantha',
      'Daniel',
      'Karen',
      'Alex'
    ];

    for (const name of preferredVoices) {
      const voice = this.voices.find(v => v.name.includes(name));
      if (voice) {
        this.selectedVoice = voice;
        break;
      }
    }

    // Fallback to first English voice
    if (!this.selectedVoice) {
      this.selectedVoice = this.voices.find(v => v.lang.startsWith('en')) ||
```

```
this.voices[0];    }    this.populateVoiceSelector();  }  createUI() {    const
widget = document.createElement('div');    widget.id = 'tts-reader';
widget.innerHTML = `      <div class="tts-collapsed" id="tts-toggle">        <svg
viewBox="0 0 24 24" width="24" height="24">          <path fill="currentColor"
d="M3 9v6h4l5 5V4L7 9H3zm13.5 3c0-1.77-1.02-3.29-2.5-4.03v8.05c1.48-.73 2.5-2.25
2.5-4.02zM14 3.23v2.06c2.89.86 5 3.54 5 6.71s-2.11 5.85-5 6.71v2.06c4.01-.91 7-
4.49 7-8.77s-2.99-7.86-7-8.77z"/>        </svg>      </div>      <div class="tts-
expanded" id="tts-panel">        <div class="tts-header">          <span>Lecture
Reader</span>          <button id="tts-minimize" title="Minimize">-</button>
</div>        <div class="tts-progress">          <div class="tts-progress-bar"
id="tts-progress-bar"></div>        </div>        <div class="tts-controls">
<button id="tts-prev" title="Previous section">⏮</button>          <button
id="tts-play" title="Play">▶</button>          <button id="tts-next" title="Next
section">⏭</button>        </div>        <div class="tts-settings">
<label>            Speed: <span id="tts-rate-display">1.0x</span>
<input type="range" id="tts-rate" min="0.5" max="2.5" step="0.1" value="1.0">
</label>          <label>            Voice:            <select id="tts-voice">
</select>          </label>        </div>        <div class="tts-status" id="tts-
status">Ready</div>      </div>    `;    document.body.appendChild(widget);  }
populateVoiceSelector() {    const selector = document.getElementById('tts-
voice');    if (!selector) return;    selector.innerHTML = '';    // Group by
language    const englishVoices = this.voices.filter(v =>
v.lang.startsWith('en'));    englishVoices.forEach(voice => {      const option =
document.createElement('option');      option.value = voice.name;
option.textContent = `${voice.name} (${voice.lang})`;      if (voice ===
this.selectedVoice) {        option.selected = true;      }
selector.appendChild(option);    });  }  bindEvents() {
document.getElementById('tts-toggle').addEventListener('click', () =>
this.togglePanel());    document.getElementById('tts-
minimize').addEventListener('click', () => this.togglePanel());
document.getElementById('tts-play').addEventListener('click', () =>
this.togglePlayback());    document.getElementById('tts-
prev').addEventListener('click', () => this.prevChunk());
document.getElementById('tts-next').addEventListener('click', () =>
this.nextChunk());    document.getElementById('tts-
rate').addEventListener('input', (e) => {      this.rate =
parseFloat(e.target.value);      document.getElementById('tts-rate-
display').textContent = `${this.rate.toFixed(1)}x`;      if (this.isPlaying) {
// Restart current chunk with new rate        const wasPlaying = this.isPlaying;
this.stop();        if (wasPlaying) this.play();      }    });
document.getElementById('tts-voice').addEventListener('change', (e) => {
this.selectedVoice = this.voices.find(v => v.name === e.target.value);      if
(this.isPlaying) {        this.stop();        this.play();      }    });    //
Keyboard shortcuts    document.addEventListener('keydown', (e) => {      // Only
if not typing in an input      if (e.target.tagName === 'INPUT' ||
e.target.tagName === 'TEXTAREA') return;      if (e.key === ' ' && e.ctrlKey) {
e.preventDefault();        this.togglePlayback();      } else if (e.key ===
'ArrowRight' && e.ctrlKey) {        e.preventDefault();        this.nextChunk();
} else if (e.key === 'ArrowLeft' && e.ctrlKey) {        e.preventDefault();
this.prevChunk();      }    });  }  togglePanel() {    const panel =
document.getElementById('tts-panel');    const toggle =
document.getElementById('tts-toggle');    panel.classList.toggle('visible');
toggle.classList.toggle('hidden');    // Extract content on first open    if
(this.chunks.length === 0) {      this.extractContent();    }  }
```

```
extractContent() {    // Target MkDocs content area    const content =
document querySelector('.md-content__inner') ||
document querySelector('article') ||
document querySelector('main');    if (!content) {      this setStatus('No
content found');      return;    }    // Clone to avoid modifying the page
const clone = content cloneNode(true);    // Remove elements we don't want read
const removeSelectors = [      'script', 'style', 'nav', '.headerlink', '.toc',
'pre', 'code', '.highlight', // Skip code blocks      '.admonition.note', //
Optional: skip admonitions      '#tts-reader'    ];
removeSelectors forEach(sel => {      clone querySelectorAll(sel).forEach(el =>
el remove());    });    // Get text content    let text = clone.textContent ||
'';    // Clean up whitespace    text = text      .replace(/\s+/g, ' ')
.replace(/\n+/g, '\n')      .trim();    // Chunk by sentences, respecting the
size limit    this chunks = this chunkText(text);    this.currentChunk = 0;
this setStatus(`${this chunks length} sections ready`);    this.updateProgress();
}  chunkText(text) {    const chunks = [];    // Split on sentence boundaries
const sentences = text match(/[^.!?]+[.!?]+/g) || [text];    let currentChunk =
'';    for (const sentence of sentences) {      if ((currentChunk +
sentence) length > this chunkSize) {        if (currentChunk) {
chunks push(currentChunk trim());        }        currentChunk = sentence;      }
else {        currentChunk += sentence;      }    }    if (currentChunk.trim()) {
chunks push(currentChunk trim());    }    return chunks;  }  play() {    if
(this chunks length === 0) {      this extractContent();      if
(this chunks length === 0) return;    }    if (this isPaused) {
this synth resume();      this isPaused = false;      this.isPlaying = true;
this updatePlayButton();      return;    }    this speakChunk(this.currentChunk);
}  speakChunk(index) {    if (index >= this chunks length) {      this.stop();
this setStatus('Finished');      return;    }    this.currentChunk = index;
const text = this chunks[index];    this utterance = new
SpeechSynthesisUtterance(text);    this utterance voice = this selectedVoice;
this utterance rate = this rate;    this utterance pitch = 1.0;
this utterance onstart = () => {      this isPlaying = true;      this.isPaused =
false;      this updatePlayButton();      this setStatus(`Section ${index + 1} of
${this chunks length}`);      this updateProgress();    };
this utterance onend = () => {      if (this isPlaying && !this.isPaused) {
this speakChunk(this currentChunk + 1);      }    };    this.utterance.onerror =
(e) => {      if (e error !== 'interrupted') {        console.error('TTS error:',
e);        this setStatus(`Error: ${e error}`);      }    };
this synth speak(this utterance);  }  pause() {    this synth.pause();
this isPaused = true;    this isPlaying = false;    this.updatePlayButton();
this setStatus('Paused');  }  stop() {    this synth cancel();    this.isPlaying
= false;    this isPaused = false;    this updatePlayButton();  }
togglePlayback() {    if (this isPlaying) {      this.pause();    } else {
this play();    }  }  prevChunk() {    this.stop();    this.currentChunk =
Math max(0, this currentChunk - 1);    this.updateProgress();
this.setStatus(`Section ${this.currentChunk + 1} of ${this.chunks.length}`);
this.play();  }  nextChunk() {    this.stop();    this.currentChunk =
Math.min(this.chunks.length - 1, this.currentChunk + 1);
this.updateProgress();    this.setStatus(`Section ${this.currentChunk + 1} of
${this.chunks.length}`);    this.play();  }  updatePlayButton() {    const btn =
document.getElementById('tts-play');    btn.textContent = this.isPlaying ? '⏸' :
'▶';    btn.title = this.isPlaying ? 'Pause' : 'Play';  }  updateProgress() {
const bar = document.getElementById('tts-progress-bar');    const percent =
this.chunks.length > 0      ? ((this.currentChunk + 1) / this.chunks.length) *
```

```
100       : 0;    bar.style.width = `${percent}%`;  }  setStatus(msg) {
document.getElementById('tts-status').textContent = msg;  }}// Initialize when
DOM is readydocument.addEventListener('DOMContentLoaded', () => {  // Check for
Speech Synthesis support  if ('speechSynthesis' in window) {
window.lectureReader = new LectureReader();  } else {    console.warn('Speech
Synthesis not supported in this browser');  }});
```

## The Stylesheet

```css
/* docs/stylesheets/tts-reader.css */

#tts-reader {
  position: fixed;
  bottom: 20px;
  right: 20px;
  z-index: 9999;
  font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto, sans-serif;
}

.tts-collapsed {
  width: 50px;
  height: 50px;
  background: #2094f3;
  border-radius: 50%;
  display: flex;
  align-items: center;
  justify-content: center;
  cursor: pointer;
  color: white;
  box-shadow: 0 2px 10px rgba(0,0,0,0.2);
  transition: transform 0.2s, background 0.2s;
}

.tts-collapsed:hover {
  transform: scale(1.1);
  background: #1976d2;
}

.tts-collapsed.hidden {
  display: none;
}

.tts-expanded {
  display: none;
  width: 280px;
  background: white;
  border-radius: 12px;
  box-shadow: 0 4px 20px rgba(0,0,0,0.15);
  overflow: hidden;
}

.tts-expanded.visible {
  display: block;
}

.tts-header {
  background: #2094f3;
  color: white;
  padding: 12px 15px;
  display: flex;
  justify-content: space-between;
  align-items: center;
  font-weight: 500;
```

```
}.tts-header button {  background: none;  border: none;  color: white;  font-
size: 18px;  cursor: pointer;  padding: 0 5px;}.tts-progress {  height: 4px;
background: #e0e0e0;}.tts-progress-bar {  height: 100%;  background: #2094f3;
width: 0%;  transition: width 0.3s;}.tts-controls {  display: flex;  justify-
content: center;  gap: 15px;  padding: 15px;}.tts-controls button {  width: 44px;
height: 44px;  border-radius: 50%;  border: none;  background: #f5f5f5;  font-
size: 16px;  cursor: pointer;  transition: background 0.2s;}.tts-controls
button:hover {  background: #e0e0e0;}#tts-play {  width: 54px;  height: 54px;
background: #2094f3;  color: white;  font-size: 20px;}#tts-play:hover {
background: #1976d2;}.tts-settings {  padding: 0 15px 15px;}.tts-settings label {
display: block;  margin-bottom: 10px;  font-size: 13px;  color: #666;}.tts-
settings input[type="range"] {  width: 100%;  margin-top: 5px;}.tts-settings
select {  width: 100%;  padding: 6px;  border: 1px solid #ddd;  border-radius:
4px;  margin-top: 5px;  font-size: 12px;}.tts-status {  padding: 10px 15px;
background: #f9f9f9;  font-size: 12px;  color: #888;  text-align: center;
border-top: 1px solid #eee;}/* Dark mode support for Material theme */[data-md-
color-scheme="slate"] .tts-expanded {  background: #2d2d2d;}[data-md-color-
scheme="slate"] .tts-controls button {  background: #404040;  color: #fff;}[data-
md-color-scheme="slate"] .tts-controls button:hover {  background: #505050;}
[data-md-color-scheme="slate"] .tts-settings label {  color: #aaa;}[data-md-
color-scheme="slate"] .tts-settings select {  background: #404040;  color: #fff;
border-color: #555;}[data-md-color-scheme="slate"] .tts-status {  background:
#252525;  color: #888;  border-color: #404040;}
```

## MkDocs Configuration

```yaml
# mkdocs.yml

site_name: Rust Learning Lectures
theme:
  name: material
  palette:
    - scheme: default
      toggle:
        icon: material/brightness-7
        name: Switch to dark mode
    - scheme: slate
      toggle:
        icon: material/brightness-4
        name: Switch to light mode

extra_javascript:
  - javascripts/tts-reader.js

extra_css:
  - stylesheets/tts-reader.css
```

## Usage

Once integrated, a speaker icon appears in the bottom-right corner. Click it to expand the reader panel, then hit play. The reader will work through the lecture content section by section, skipping code blocks automatically.

**Keyboard shortcuts:**

- `Ctrl+Space` — Play/Pause
- `Ctrl+←` — Previous section
- `Ctrl+→` — Next section

**Tips for your lectures:**

The reader automatically skips `<pre>` and `<code>` elements, which is perfect for your two-document format. The prose lecture plays as audio while you can glance at the code companion separately.

If you want certain sections read differently, you could add data attributes:

```
<div data-tts-skip>
This won't be read aloud.
</div>
```

Then modify `extractContent()` to honor those attributes.

Want me to extend this with features like bookmarking position, localStorage persistence for settings, or integration with the code companion documents?