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
```