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
```