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

```rust
/// 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

```rust
/// 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

```
Command Line Arguments
        |
        ▼
┌──────────────────┐
│  flags/parse.rs  │    Tokenize and validate
└──────────────────┘
        |
        ▼
┌──────────────────┐
│    LowArgs       │    Direct flag mirror (this file)
└──────────────────┘
        |
        ▼
┌──────────────────┐
│  flags/hiargs.rs │    Transform, compute, build objects
└──────────────────┘
        |
        ▼
┌──────────────────┐
│     HiArgs       │    Ready for execution
└──────────────────┘
        |
        ▼
┌──────────────────┐
│     main.rs      │    Dispatch to search/files/types
└──────────────────┘
```