# 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

```rust
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

```rust
/// 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

```rust
// 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