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
}
```