

# Build Your Own Text Editor: Design Document

## Overview

This system implements a terminal-based text editor that manages file content in memory while providing real-time visual feedback through ANSI escape sequences. The key architectural challenge is coordinating between raw terminal I/O, in-memory text buffer operations, and efficient screen rendering without flicker.

This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

## Context and Problem Statement

**Milestone(s):** Milestone 1 (Raw Mode and Input), Milestone 2 (Screen Refresh)

Building a terminal-based text editor presents a unique set of architectural challenges that distinguish it fundamentally from both GUI applications and simple command-line tools. Understanding these challenges and the design approaches that address them is crucial for creating a responsive, reliable editor that feels natural to use.

### Terminal I/O Complexity: Why terminal programming is fundamentally different from GUI programming

Think of terminal programming as **conducting an orchestra through handwritten notes passed back and forth**, while GUI programming is like **having a dedicated stage manager who handles all the logistics**. In GUI programming, you have a framework that manages windows, handles input events, redraws dirty regions, and provides high-level abstractions like buttons and text boxes. In terminal programming, you're working at the level of individual character streams, escape sequences, and raw keyboard scancodes—essentially controlling a 1970s-era display terminal that happens to be emulated in modern software.

The **fundamental paradigm difference** lies in the interaction model. GUI applications operate in a **retained mode** where the framework maintains a scene graph or widget tree, automatically handles redraws when something changes, and provides high-level input events like "button clicked" or "text changed." Terminal applications must operate in **immediate mode** where every frame is explicitly constructed from scratch, sent as a stream of characters and control sequences, and input arrives as raw byte sequences that must be manually parsed into meaningful commands.

## Decision: Immediate Mode Terminal Rendering vs. Retained Mode Abstractions

- **Context:** Terminal applications can either work directly with character streams and escape sequences (immediate mode) or build abstractions that hide terminal complexity behind higher-level interfaces (retained mode)
- **Options Considered:** Direct ANSI sequence manipulation, terminal UI libraries (ncurses, termbox), custom retained-mode abstraction layer
- **Decision:** Use direct ANSI sequence manipulation with minimal abstractions
- **Rationale:** Educational value in understanding terminal fundamentals, better performance control, simpler debugging when things go wrong, matches the kilo editor's approach
- **Consequences:** More manual work required, deeper understanding of terminal protocols needed, but complete control over rendering behavior

| Approach              | Complexity                | Performance | Learning Value | Debugging Ease                   |
|-----------------------|---------------------------|-------------|----------------|----------------------------------|
| Direct ANSI sequences | High setup, low ongoing   | Optimal     | Maximum        | Transparent                      |
| Terminal UI library   | Low setup, medium ongoing | Good        | Medium         | Library abstractions hide issues |
| Custom abstractions   | High setup, low ongoing   | Variable    | High           | Depends on abstraction quality   |

**Input handling complexity** represents another major challenge. In GUI applications, the windowing system delivers discrete, pre-processed events: "user pressed the 'a' key" or "user clicked at coordinates (100, 50)." Terminal input arrives as a **raw byte stream** where a single logical keypress might generate multiple bytes. For example, pressing the up arrow key sends the three-byte sequence `\x1b[A` (escape, left bracket, A). The application must implement a **state machine** that accumulates bytes until it can recognize complete key sequences, handle timing issues where bytes arrive separately, and deal with the fact that different terminal emulators might send different sequences for the same logical key.

The **screen management problem** adds another layer of complexity. GUI frameworks provide automatic dirty region tracking and efficient redraw mechanisms. Terminal applications must manually track what has changed, decide what portions of the screen need updating, and coordinate the cursor position with the content being displayed. A naive approach that redraws everything on every keypress creates visible flicker and poor performance. The solution requires building a **frame buffer** that accumulates all screen changes before sending them as a single atomic update to the terminal.

**Asynchronous I/O coordination** creates additional challenges. The terminal operates with separate input and output streams that are fundamentally asynchronous. While the application is writing a complex screen update, the user might press keys that generate input events. The application must be prepared to handle

input at any time while ensuring that partial screen updates don't corrupt the display. This is particularly tricky during operations like syntax highlighting, where computing the correct colors for visible text might take significant time.

**State management and error recovery** become critical concerns because terminal applications directly manipulate the terminal's operating mode. The terminal must be switched from its normal **canonical mode** (where input is line-buffered and echoed automatically) to **raw mode** (where every keypress is delivered immediately without echo). If the application crashes while in raw mode, the user's terminal remains in an unusable state where typing doesn't echo and the terminal doesn't respond normally. Robust terminal applications must install signal handlers to catch crashes and restore terminal state, but this introduces complexity around async-signal-safe programming and cleanup order.

| Terminal Mode      | Input Processing       | Output Echo | Line Editing                 | Use Case                 |
|--------------------|------------------------|-------------|------------------------------|--------------------------|
| Canonical (cooked) | Line-buffered          | Automatic   | Built-in (backspace, Ctrl+U) | Normal shell interaction |
| Raw                | Character-by-character | None        | Application-controlled       | Text editors, games      |
| Cbreak             | Character-by-character | Automatic   | Limited                      | Simple interactive tools |

**Cross-platform compatibility** adds yet another dimension of complexity. While ANSI escape sequences are standardized, different terminal emulators interpret them with subtle variations. Windows Command Prompt has historically had limited ANSI support (though this has improved). Terminal size detection, color capability negotiation, and special key sequences all vary between platforms. A robust terminal application must either restrict itself to the lowest common denominator or implement platform-specific adaptations.

The fundamental insight here is that terminal programming forces you to work at the level of the hardware interface. You're not building on top of a rich UI framework—you're essentially writing a device driver for a text display, with all the low-level concerns that entails.

## Existing Text Editor Approaches: Comparison of vim, nano, and modern editors' architectural patterns

Understanding how different text editors approach these terminal programming challenges reveals several successful architectural patterns, each with distinct trade-offs in complexity, functionality, and user experience.

**Vim's modal architecture** represents one of the most sophisticated approaches to terminal text editing. Think of vim as a **Swiss Army knife with multiple specialized tools** rather than a single general-purpose instrument. The editor operates in distinct modes—normal mode for navigation and commands, insert mode

for text entry, visual mode for selection, and command mode for complex operations. This modal approach elegantly solves the input interpretation problem by changing the meaning of keystrokes based on context.

Vim's internal architecture centers around a **command language** where almost every user action translates to an internal command that can be recorded, repeated, and combined. The `d` (delete) command can be combined with motion commands like `w` (word) or `{` (paragraph) to create composite operations like `dw` (delete word) or `d{` (delete paragraph). This compositional approach allows complex editing operations to be built from simple primitives, reducing the amount of special-case code needed.

The **buffer management system** in vim demonstrates a sophisticated approach to text representation. Vim maintains multiple buffers simultaneously, each containing the text of a file along with metadata like cursor position, undo history, and local settings. The screen rendering system can display different portions of different buffers simultaneously through windows and tabs. This separation between logical text containers (buffers) and display regions (windows) enables powerful workflows but requires complex coordination between buffer state and screen state.

| Vim Component   | Responsibility                                | Key Design Pattern                                 |
|-----------------|---|--|
| Buffer Manager  | Text storage, undo history, file I/O          | Multiple named buffers with independent state      |
| Command Parser  | Keystroke interpretation, command composition | Context-sensitive parsing based on current mode    |
| Motion System   | Cursor movement, text object definition       | Composable movement primitives                     |
| Display Manager | Window layout, syntax highlighting            | Separation of logical content from display regions |

**Nano's simplicity-first approach** takes the opposite philosophy. Think of nano as a **pencil and paper**—straightforward, predictable, and immediately usable without training. Nano operates in a single mode where most keystrokes insert text directly, and special operations are accessed through Ctrl key combinations that are displayed at the bottom of the screen as hints.

Nano's architecture reflects this simplicity bias through a **monolithic design** where most functionality is concentrated in a single main loop. The editor reads a keypress, determines its meaning through a simple lookup table, performs the corresponding action (which might involve buffer modification, cursor movement, or screen refresh), and immediately updates the display. This approach minimizes state management complexity at the cost of limiting functionality.

The **screen rendering approach** in nano favors correctness over performance optimization. Rather than maintaining complex dirty region tracking, nano often redraws the entire visible portion of the text on each change. While this creates more terminal I/O than necessary, it eliminates entire classes of bugs related to inconsistent screen state and makes the rendering code much simpler to understand and maintain.

**Modern terminal editors** like micro, kakoune, and helix represent attempts to combine vim's power with nano's discoverability while leveraging modern programming practices. These editors typically employ **event-driven architectures** with clear separation between input handling, buffer management, and display rendering.

Helix, for example, implements a **selection-first editing model** where text objects are first selected (visualized on screen) and then operated upon. This approach makes the result of commands more predictable than vim's motion-then-operator model while still enabling powerful text manipulation. The architecture reflects this by maintaining selection state as a first-class concept alongside cursor position.

### Decision: Single-Buffer vs. Multi-Buffer Architecture

- **Context:** Text editors can support editing one file at a time or maintain multiple files simultaneously in memory
- **Options Considered:** Single buffer with file switching, multiple buffers with window management, multiple buffers with tab interface
- **Decision:** Single buffer architecture for initial implementation
- **Rationale:** Simpler state management, clearer data flow, matches kilo's approach and learning objectives, can be extended later
- **Consequences:** Editing multiple files requires closing and reopening, but implementation complexity remains manageable for learning purposes

**Plugin and extensibility approaches** reveal another key architectural decision. Vim's extensibility through Vimscript (and now Lua) requires embedding an entire scripting language interpreter within the editor. This enables tremendous customization but significantly increases code complexity and potential security issues. Nano deliberately avoids plugins entirely, keeping the codebase minimal. Modern editors often use **Language Server Protocol (LSP)** to push complex functionality like syntax highlighting and code completion into separate processes, reducing the editor's core complexity while still providing advanced features.

| Editor     | Input Model                         | Buffer Management                     | Extensibility                      | Screen Rendering                      |
|------------|-------------------------------------|---------------------------------------|------------------------------------|---------------------------------------|
| Vim        | Modal with command composition      | Multiple buffers, windows, tabs       | Embedded scripting (Vimscript/Lua) | Optimized with dirty regions          |
| Nano       | Direct insertion with Ctrl commands | Single buffer                         | None                               | Full screen refresh                   |
| Helix      | Selection-first with operators      | Multiple buffers with LSP integration | External LSP servers               | Event-driven with incremental updates |
| Our Editor | Direct insertion with special keys  | Single buffer with basic operations   | None initially                     | Frame-buffered with minimal flicker   |

**Performance optimization strategies** differ significantly between editors. Vim employs sophisticated caching and lazy evaluation—syntax highlighting is computed incrementally, and screen updates are batched. Nano prioritizes simplicity and accepts some performance overhead. Modern editors often use **background processing** for expensive operations like syntax highlighting, updating the display asynchronously as results become available.

The key insight is that there's no universally "correct" approach to terminal text editor architecture. Each design represents a different balance between functionality, complexity, and usability. Understanding these trade-offs helps inform architectural decisions for new implementations.

## Implementation Guidance

### A. Technology Recommendations Table:

| Component        | Simple Option   | Advanced Option                      |
|------------------|---|--------------------------------------|
| Terminal Control | Direct ANSI escape sequences                              | ncurses library                      |
| Input Processing | Raw character reading with manual escape sequence parsing | termbox or similar abstraction       |
| Text Storage     | Array of strings (char**)                                 | Gap buffer or rope data structure    |
| Screen Buffering | Single write() with accumulated output                    | Double buffering with diff detection |
| File I/O         | Standard C file operations (fopen, fread, fwrite)         | Memory-mapped files                  |

### B. Recommended File Structure:

For a C implementation following the kilo editor approach, organize the code as follows:

```
text-editor/
    kilo.c           ← main implementation file
    kilo.h           ← data structures and function declarations
    Makefile         ← build configuration
    test/
        test_input.txt   ← sample files for testing
        test_long_lines.txt ← test horizontal scrolling
    docs/
        terminal_sequences.md ← ANSI escape sequence reference
```

The monolithic file approach matches kilo's educational philosophy—all code is visible in one place, making it easier to understand the relationships between components. As the implementation grows, consider splitting into modules:

```
text-editor/
src/
    main.c           ← entry point and main loop
    terminal.c       ← raw mode, ANSI sequences, cleanup
    buffer.c         ← text storage and manipulation
    input.c          ← keypress parsing and command dispatch
    render.c         ← screen composition and output
    file.c           ← load and save operations
include/
    editor.h         ← shared data structures
    terminal.h       ← terminal control interface
    buffer.h         ← text buffer interface
```

### C. Infrastructure Starter Code:

Here's a complete terminal management module that handles the low-level details:

```
// terminal.h - Terminal control interface

#ifndef TERMINAL_H

#define TERMINAL_H


#include <termios.h>

// Terminal state management

int enable_raw_mode(void);

void disable_raw_mode(void);

int get_window_size(int *rows, int *cols);

// ANSI escape sequence generation

void clear_screen(void);

void position_cursor(int row, int col);

void hide_cursor(void);

void show_cursor(void);

void set_status_message(const char *fmt, ...);

// Cleanup and error handling

void die(const char *s);

void editor_exit_cleanup(void);

#endif

// terminal.c - Complete terminal control implementation

#include "terminal.h"

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>
```

```
#include <sys/ioctl.h>

#include <signal.h>

static struct termios orig_termios;

void disable_raw_mode(void) {

    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &orig_termios) == -1) {

        die("tcsetattr");

    }

}

int enable_raw_mode(void) {

    if (tcgetattr(STDIN_FILENO, &orig_termios) == -1) die("tcgetattr");

    atexit(disable_raw_mode);

    struct termios raw = orig_termios;

    raw.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);

    raw.c_oflag &= ~(OPOST);

    raw.c_cflag |= (CS8);

    raw.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);

    raw.c_cc[VMIN] = 0;

    raw.c_cc[VTIME] = 1;

    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw) == -1) die("tcsetattr");

    return 0;

}

int get_window_size(int *rows, int *cols) {

    struct winsize ws;
```

```
if (ioctl(STDOUT_FILENO, TIOCGWINSZ, &ws) == -1 || ws.ws_col == 0) {

    return -1;

} else {

    *cols = ws.ws_col;

    *rows = ws.ws_row;

    return 0;

}

}

void clear_screen(void) {

    write(STDOUT_FILENO, "\x1b[2J", 4);

}

void position_cursor(int row, int col) {

    char buf[32];

    snprintf(buf, sizeof(buf), "\x1b[%d;%dH", row, col);

    write(STDOUT_FILENO, buf, strlen(buf));

}

void hide_cursor(void) {

    write(STDOUT_FILENO, "\x1b[?25l", 6);

}

void show_cursor(void) {

    write(STDOUT_FILENO, "\x1b[?25h", 6);

}

void die(const char *s) {
```

```
clear_screen();

position_cursor(1, 1);

perror(s);

exit(1);

}

// Signal handler for graceful cleanup

void sigint_handler(int sig) {

    editor_exit_cleanup();

    exit(1);

}

void editor_exit_cleanup(void) {

    clear_screen();

    position_cursor(1, 1);

    show_cursor();

    disable_raw_mode();

}
```

#### D. Core Logic Skeleton Code:

Here are the key data structures and function signatures that learners should implement:

```
// editor.h - Core data structures

#ifndef EDITOR_H

#define EDITOR_H


#include <time.h>

typedef struct erow {

    int size;

    char *chars;

    int rsize;

    char *render;

} erow;

struct editor_config {

    int cx, cy;           // Cursor position

    int rx;               // Render cursor position (for tabs)

    int rowoff;            // Row offset (vertical scrolling)

    int coloff;            // Column offset (horizontal scrolling)

    int screenrows;        // Terminal height

    int screencols;        // Terminal width

    int numrows;           // Number of rows in file

    erow *row;             // Array of text rows

    int dirty;              // File modified flag

    char *filename;         // Current filename

    char statusmsg[80];     // Status bar message

    time_t statusmsg_time; // Status message timestamp

};

extern struct editor_config E;
```

C

```
// Function signatures for learners to implement

void editor_open(char *filename);

void editor_save(void);

int editor_read_key(void);

void editor_process_keypress(void);

void editor_refresh_screen(void);

void editor_insert_char(int c);

void editor_del_char(void);

void editor_insert_newline(void);

#endif

// Key implementation skeleton with detailed TODOs

int editor_read_key(void) {

    int nread;

    char c;

    // TODO 1: Read one character from stdin using read()

    // TODO 2: If read returns 0, continue reading (timeout occurred)

    // TODO 3: If read returns -1 and errno != EAGAIN, call die()

    // TODO 4: If character is ESC (27), read additional characters for escape sequences

    // TODO 5: Handle arrow keys: ESC[A (up), ESC[B (down), ESC[C (right), ESC[D (left)

    // TODO 6: Handle page up/down: ESC[5~ and ESC[6~

    // TODO 7: Handle home/end keys: ESC[H, ESC[F, ESC[1~, ESC[4~

    // TODO 8: Return special key codes or the character itself

    // Hint: Use a while loop with timeout for robust escape sequence parsing

}
```

```

void editor_insert_char(int c) {

    // TODO 1: If cursor is at end of file, append a new empty row

    // TODO 2: Call editor_row_insert_char() to insert character at cursor position

    // TODO 3: Advance cursor x position

    // TODO 4: Set dirty flag to indicate file has been modified

    // Hint: Remember to handle the case where cursor is past the end of the current row

}

void editor_refresh_screen(void) {

    // TODO 1: Create a screen buffer to accumulate output

    // TODO 2: Hide cursor to prevent flicker during redraw

    // TODO 3: Position cursor at top-left (1,1)

    // TODO 4: Draw each visible row, handling vertical scrolling

    // TODO 5: Draw status bar at bottom with filename and cursor position

    // TODO 6: Draw message bar below status bar

    // TODO 7: Position cursor at correct location based on cx, cy, and scrolling offsets

    // TODO 8: Show cursor again

    // TODO 9: Write entire buffer to stdout in one operation

    // Hint: Use snprintf to build escape sequences, realloc to grow buffer as needed

}

```

## E. Language-Specific Hints:

For C implementation:

- Use `tcgetattr()` and `tcsetattr()` for terminal mode control
- Use `read()` with `VMIN=0, VTIME=1` for non-blocking input with timeout
- Use `ioctl(STDOUT_FILENO, TIOCGWINSZ, &ws)` for terminal size detection
- Use `atexit()` to register cleanup functions for normal exit
- Use `signal()` to handle SIGINT (Ctrl+C) for abnormal exit
- Use `realloc()` carefully—always check return value and handle NULL

- Use `snprintf()` instead of `sprintf()` to prevent buffer overflows

Key C pitfalls to avoid:

- Always null-terminate strings when building them manually
- Check return values from system calls (`read`, `write`, `tcsetattr`)
- Free allocated memory in reverse order of allocation
- Handle the case where `malloc()` or `realloc()` returns NULL

## F. Milestone Checkpoints:

After implementing Milestone 1 (Raw Mode and Input):

- Run the program and verify that typed characters don't echo to screen
- Press arrow keys and verify they're recognized (print key codes for debugging)
- Press Ctrl+C and verify terminal is restored to normal mode
- Force-quit with Ctrl+Z then `kill %1` and verify terminal still works

Expected behavior:

```
$ ./editor                                BASH

# Screen should clear, cursor should be visible

# Typing 'hello' should not display anything yet (no echo)

# Arrow keys should move cursor or print debug messages

# Ctrl+Q should exit cleanly and restore terminal
```

After implementing Milestone 2 (Screen Refresh):

- Open editor and verify screen clears and cursor positions correctly
- Type characters and verify they appear on screen at cursor position
- Verify status bar shows at bottom with editor information
- Verify cursor tracks correctly as characters are typed

Expected screen layout:

```
Hello world█
~
~
[filename.txt] 1 lines          1,12
Welcome to the editor!
```

Common issues and debugging:

- **Screen flickers:** You're writing multiple small chunks instead of one large buffer
- **Cursor in wrong position:** Check math for converting logical position to screen coordinates
- **Status bar overwritten:** Ensure cursor positioning accounts for status bar rows
- **Garbled display after exit:** Terminal cleanup isn't running—check signal handlers

## Goals and Non-Goals

**Milestone(s):** All milestones (defines scope for entire project)

Building a text editor involves countless design decisions and potential features. To create a focused learning experience that teaches fundamental concepts without overwhelming complexity, we must carefully define what our editor will and will not do. Think of this as drawing the boundaries of a city — we need clear borders to know what infrastructure to build inside and what lies beyond our responsibility.

The primary educational goal is to understand the core architectural patterns that make terminal-based text editors work: how to coordinate between raw terminal I/O, in-memory text manipulation, and real-time visual feedback. By constraining our scope, we can dive deep into these essential concepts rather than getting lost in feature creep.

### Primary Goals

Our text editor will provide a **complete but minimal editing experience** that demonstrates all the fundamental building blocks of terminal text editors. The editor should feel familiar to users of nano or basic vim, offering intuitive navigation and editing without requiring extensive documentation to use.

#### Core Editing Functionality

The editor must support the fundamental text manipulation operations that users expect from any text editor. This includes character-by-character insertion and deletion, line operations like splitting and joining, and basic cursor movement. Every keystroke should provide immediate visual feedback, making the editing experience feel responsive and predictable.

| Operation           | Behavior  | Implementation Requirement                        |
|---------------------|---|---|
| Character Insertion | Insert at cursor, shift right, advance cursor     | Real-time buffer modification with screen update  |
| Character Deletion  | Backspace removes left, Delete removes right      | Handle boundary conditions at line start/end      |
| Line Breaking       | Enter splits line at cursor position              | Create new buffer line, manage cursor positioning |
| Line Joining        | Backspace at line start joins with previous       | Merge buffer lines, handle cursor repositioning   |
| Cursor Movement     | Arrow keys navigate character and line boundaries | Coordinate logical position with visual display   |

## File Operations

The editor must handle the complete lifecycle of text files, from loading existing content to saving modifications. Users need confidence that their work won't be lost and clear feedback about the file's modification state.

| Operation             | Behavior                               | User Feedback                         |
|-----------------------|--|---------------------------------------|
| File Loading          | Read file into buffer on startup       | Display filename in status bar        |
| File Saving           | Write buffer to disk, clear dirty flag | Show save confirmation message        |
| Modification Tracking | Set dirty flag on any edit             | Display asterisk (*) next to filename |
| New File Creation     | Start with empty buffer                | Show "New File" until first save      |

## Terminal Integration

The editor must seamlessly integrate with the terminal environment, handling window resizing, proper cleanup, and graceful error recovery. Users should never find their terminal in an unusable state after running the editor.

## Visual Feedback Systems

Real-time visual feedback is crucial for user confidence and productivity. The editor must provide clear status information, responsive cursor movement, and smooth scrolling for files larger than the terminal window.

**Design Principle:** Every user action should produce immediate and predictable visual feedback. The delay between keypress and screen update should be imperceptible under normal conditions.

## Advanced Features (Included)

While maintaining simplicity, our editor will include several advanced features that demonstrate important text editor concepts and significantly improve usability.

### Undo and Redo System

A command pattern-based undo system teaches important architectural concepts about reversible operations and state management. This feature transforms the editor from a basic character manipulator into a professional tool that users can trust with important documents.

The undo system will track every modification operation, allowing users to step backward through their editing history and re-apply undone changes. This requires careful design of the edit operation representation and efficient storage of the undo stack.

### Incremental Search

Search functionality demonstrates real-time user interaction patterns and text processing algorithms. The incremental approach, where results update as the user types, provides immediate feedback and helps users refine their queries dynamically.

| Search Feature              | Behavior                                  | Educational Value                   |
|-----------------------------|---|-------------------------------------|
| Incremental Updates         | Results update with each character typed  | Real-time UI state management       |
| Forward/Backward Navigation | Navigate between matches with F3/Shift+F3 | Search state machine implementation |
| Match Highlighting          | Visual indication of all matches          | Screen rendering with overlays      |
| Cancel and Restore          | Escape returns to original position       | State preservation and restoration  |

### Syntax Highlighting

Basic syntax highlighting for common programming languages (C, JavaScript, Python) demonstrates tokenization, state machines, and the application of visual styling to text content. This feature shows how editors can provide context-aware assistance to users.

The implementation will use simple pattern matching and finite state machines to identify language elements like keywords, strings, and comments. While not as sophisticated as modern editors, it illustrates the fundamental concepts behind syntax-aware text processing.

### Viewport Management and Scrolling

For files larger than the terminal window, the editor must implement smooth scrolling that keeps the cursor visible while providing efficient navigation. This includes both vertical scrolling for long files and horizontal scrolling for wide lines.

## Decision: Immediate Mode Rendering

- **Context:** Terminal editors can use immediate mode (redraw everything) or retained mode (track changes) rendering
- **Options Considered:** Immediate mode vs. retained mode vs. hybrid approach
- **Decision:** Pure immediate mode rendering for simplicity
- **Rationale:** Immediate mode eliminates complex state synchronization between model and view, making the codebase much easier to understand and debug. Terminal rendering is fast enough that full redraws are not perceptually slow.
- **Consequences:** Simpler architecture and debugging at the cost of some CPU cycles on large files

## Non-Goals (Explicitly Excluded)

To maintain focus on core concepts, we explicitly exclude several categories of features that would add significant complexity without proportional educational value.

### Advanced Text Editor Features

These features, while useful in production editors, would obscure the fundamental concepts we're trying to teach:

| Excluded Feature      | Rationale  | Alternative Learning Path                               |
|-----------------------|--|---|
| Multiple Buffers/Tabs | Complex state management obscures core concepts  | Build as separate project after mastering single buffer |
| Split Windows/Panes   | Screen real estate management complexity         | Focus on single view mastery first                      |
| Advanced Navigation   | Jump-to-line, bracket matching add UI complexity | Implement as extensions after core completion           |
| Folding/Outlining     | Document structure analysis beyond scope         | Separate project focused on parsing                     |
| Macros/Recording      | Complex command recording and playback           | Advanced feature for later learning                     |

## Plugin and Extension Systems

Plugin architectures require sophisticated component isolation, API design, and dynamic loading mechanisms. While valuable in production systems, they would triple the codebase size and shift focus from text editing fundamentals to software architecture patterns.

## Configuration Management

Configuration files, user preferences, and customizable keybindings introduce file parsing, validation, and storage concerns that distract from the core terminal programming and text manipulation concepts.

## Advanced File Operations

| Excluded Feature        | Complexity Added                       | Focus Maintained On         |
|-------------------------|--|-----------------------------|
| Multiple File Formats   | Encoding detection, format conversion  | Pure text manipulation      |
| Binary File Support     | Hex editing, non-text rendering        | Character-based editing     |
| Large File Optimization | Memory mapping, lazy loading           | In-memory buffer operations |
| Backup/Recovery         | File system management, crash recovery | Core editing operations     |
| Remote Files            | Network protocols, async I/O           | Local terminal interaction  |

## Modern Editor Features

Contemporary editors like VS Code include language servers, intelligent completion, real-time collaboration, and integrated development tools. These features require complex client-server architectures, protocol implementations, and sophisticated UI frameworks that would overshadow the terminal programming concepts we're teaching.

## Performance Optimization

While important in production systems, advanced optimizations like incremental parsing, background syntax highlighting, and memory pooling would complicate the implementation without teaching fundamental concepts. Our editor prioritizes code clarity over maximum performance.



### Pitfall: Feature Creep

Learners often want to add "just one more feature" during implementation. This leads to incomplete core functionality and architectural debt. Resist the temptation to add features not listed in the goals. Complete the core editor first, then build extensions as separate projects.

## Success Criteria

The completed editor will be considered successful when it meets these concrete criteria:

### Functional Requirements

| Requirement          | Verification Method                       | Acceptance Criteria                 |
|----------------------|---|-------------------------------------|
| File Editing         | Load, modify, save a C source file        | No data loss, modifications persist |
| Terminal Integration | Resize terminal window during editing     | Editor adapts layout, no corruption |
| Error Recovery       | Kill editor with Ctrl+C, restart terminal | Terminal remains fully functional   |
| Large File Handling  | Open 1000+ line file                      | Smooth scrolling, responsive cursor |
| Search Functionality | Find text in large document               | Fast results, accurate highlighting |

## Educational Requirements

Students completing this project should understand:

- How terminal raw mode differs from canonical mode and why it matters
- The coordination between input processing, buffer management, and screen rendering
- Why immediate-mode rendering simplifies terminal applications
- How ANSI escape sequences control terminal behavior
- The architecture patterns that scale from simple to complex text editors

## Code Quality Requirements

The implementation should demonstrate good systems programming practices:

- Proper resource management with cleanup on all exit paths
- Clear separation of concerns between components
- Error handling that preserves system state
- Memory management appropriate for the chosen language
- Code organization that supports future extension

## Scope Boundaries

Understanding what lies just outside our scope helps clarify the architectural decisions within it. These boundaries define where our implementation ends and where future learning projects might begin.

### Input Processing Boundary

Our editor handles standard keyboard input and basic terminal escape sequences. We don't implement mouse support, clipboard integration, or exotic terminal capabilities. This boundary keeps the input processing logic focused on keyboard-driven interaction patterns.

### File System Boundary

The editor works with single text files in the local filesystem. We don't handle directories, file browsing, or filesystem watching. This boundary maintains focus on text manipulation rather than file management.

## Language Support Boundary

Syntax highlighting covers basic language elements (keywords, strings, comments) for a few common languages. We don't implement semantic analysis, error detection, or integration with external language tools. This boundary demonstrates tokenization concepts without the complexity of full language support.

## User Interface Boundary

The interface uses character-cell rendering with ANSI escape sequences. We don't implement graphical elements, proportional fonts, or advanced terminal features like 256-color support. This boundary keeps the rendering model simple and portable.

**The Learning Journey:** These boundaries aren't limitations — they're stepping stones. Master the concepts within these bounds, then use that foundation to tackle the excluded features as separate learning projects. Each boundary crossed teaches new architectural patterns and implementation techniques.

## Implementation Guidance

The scope definition directly impacts how we structure the learning experience and implementation approach. Here are practical guidelines for maintaining focus during development.

### A. Technology Recommendations Table:

| Component        | Simple Option                              | Advanced Option                          |
|------------------|--|--|
| Terminal Control | Raw termios + ANSI escape sequences        | ncurses library with window management   |
| Text Storage     | Array of strings (char**)                  | Gap buffer or rope data structure        |
| Input Processing | Character-by-character with manual parsing | Event-driven input with key binding maps |
| File I/O         | Standard fopen/fread/fwrite                | Memory-mapped files with lazy loading    |
| Error Handling   | Simple error codes with cleanup            | Exception handling with stack unwinding  |

### B. Recommended File/Module Structure:

Organize the codebase to reflect the clear separation between core functionality and excluded features:

```
kilo-editor/
  kilo.c           ← main program and editor_config management
  terminal.c       ← raw mode setup and ANSI escape sequences
  input.c          ← keypress parsing and command dispatch
  buffer.c         ← text storage and manipulation operations
  display.c        ← screen rendering and viewport management
  search.c         ← incremental search functionality
  syntax.c         ← basic syntax highlighting rules
  kilo.h           ← shared data structures and function declarations

  tests/
    test_basic.c   ← milestone verification tests
    test_files/     ← sample files for testing

  examples/
    sample.c       ← C file for syntax highlighting demo
    large.txt       ← 1000+ line file for scrolling tests
```

This structure makes it easy to see what's included and provides clear extension points for future features.

### C. Infrastructure Starter Code:

**Terminal Control Foundation** (complete implementation):

```
// terminal.c - Complete terminal management implementation
```

C

```
#include <termios.h>

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <sys/ioctl.h>

struct termios orig_termios;

void disable_raw_mode() {

    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &orig_termios) == -1)
        perror("tcsetattr");

}

int enable_raw_mode() {

    if (tcgetattr(STDIN_FILENO, &orig_termios) == -1) return -1;
    atexit(disable_raw_mode);

    struct termios raw = orig_termios;

    raw.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);

    raw.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);

    raw.c_oflag &= ~(OPOST);

    raw.c_cflag |= (CS8);

    raw.c_cc[VMIN] = 0;

    raw.c_cc[VTIME] = 1;

    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw) == -1) return -1;

    return 0;
}
```

```
}

int get_window_size(int *rows, int *cols) {

    struct winsize ws;

    if (ioctl(STDOUT_FILENO, TIOCGWINSZ, &ws) == -1 || ws.ws_col == 0) {

        return -1;

    } else {

        *cols = ws.ws_col;

        *rows = ws.ws_row;

        return 0;

    }

}

void clear_screen() {

    write(STDOUT_FILENO, "\x1b[2J", 4);

    write(STDOUT_FILENO, "\x1b[H", 3);

}

void hide_cursor() {

    write(STDOUT_FILENO, "\x1b[?25l", 6);

}

void show_cursor() {

    write(STDOUT_FILENO, "\x1b[?25h", 6);

}
```

**Error Handling Utilities** (complete implementation):

```
// error.c - Complete error handling and cleanup

#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

#include <string.h>

void die(const char *s) {

    clear_screen();

    perror(s);

    exit(1);

}

// Wrapper for file operations with error handling

FILE* safe_fopen(const char *filename, const char *mode) {

    FILE *fp = fopen(filename, mode);

    if (!fp) {

        die("fopen");

    }

    return fp;

}
```

#### D. Core Logic Skeleton Code:

```
// buffer.c - Core text buffer operations (learner implements) C

// Insert character at current cursor position in the editor buffer

void editor_insert_char(int c) {

    // TODO 1: Check if cursor is at end of file, may need to append new row

    // TODO 2: Get current row from E.row[E.cy]

    // TODO 3: Reallocate row->chars to make room for new character

    // TODO 4: Use memmove to shift existing characters right

    // TODO 5: Insert the new character at cursor position

    // TODO 6: Update row->size and increment cursor column (E.cx)

    // TODO 7: Set dirty flag to indicate unsaved changes

    // Hint: Handle special case when inserting at end of line vs middle

}

// Delete character at current cursor position

void editor_del_char() {

    // TODO 1: Check boundary conditions - can't delete at start of empty file

    // TODO 2: Handle backspace at beginning of line (join with previous line)

    // TODO 3: Handle normal character deletion within line

    // TODO 4: Use memmove to shift remaining characters left

    // TODO 5: Update row size and cursor position

    // TODO 6: Set dirty flag

    // Hint: Backspace moves cursor left then deletes, Delete deletes at cursor

}

// Break current line at cursor position (Enter key)

void editor_insert_newline() {

    // TODO 1: If at beginning of line, insert blank line above
```

```
// TODO 2: If at end of line, insert blank line below

// TODO 3: If in middle, split line at cursor position

// TODO 4: Relocate E.row array to make room for new row

// TODO 5: Create new row with characters after cursor position

// TODO 6: Truncate current row at cursor position

// TODO 7: Update numrows, cursor position, and dirty flag

}
```

## E. Language-Specific Hints:

For C implementation:

- Use `realloc()` carefully — always check return value and update pointers
- `memmove()` is safer than `memcpy()` for overlapping memory regions
- Terminal I/O should use `write()` and `read()` system calls for better control
- Use `atexit()` to register cleanup function that restores terminal mode
- Global `struct editor_config E` simplifies passing state between functions
- String operations need explicit null termination with `malloc(size + 1)`

## F. Milestone Checkpoint:

After implementing the scope-defined features:

### Verification Commands:

```
# Test basic editing

./kilo sample.c

# Type some characters, verify they appear

# Use arrow keys, verify cursor movement

# Press Enter, verify line splitting

# Press Backspace, verify character deletion

# Save with Ctrl+S, verify file updated

# Test boundaries

./kilo empty_file.txt

# Verify behavior with empty file

./kilo nonexistent.txt

# Verify creating new file
```

BASH

#### **Expected Behavior:**

- Editor starts without corrupting terminal
- All keyboard input produces expected results
- File operations work correctly
- Editor exits cleanly with Ctrl+Q
- Terminal remains usable after editor exits

#### **Signs of Scope Creep:**

- Adding features not in the goals list
- Implementing complex configuration systems
- Building plugin architectures
- Adding multiple buffer support
- Implementing advanced navigation commands

#### **G. Debugging Tips:**

| Symptom                      | Likely Cause         | How to Diagnose                   | Fix                                    |
|------------------------------|----------------------|-----------------------------------|--|
| Feature requests from users  | Scope creep          | Review goals vs requests          | Politely defer to future versions      |
| Complex conditional logic    | Feature interactions | Count feature flags in code       | Simplify by removing non-goal features |
| Difficulty testing           | Too many features    | Measure test coverage per feature | Focus tests on goal features only      |
| Long development time        | Scope expansion      | Track time per feature            | Cut features not in core goals         |
| Hard to explain architecture | Over-engineering     | Explain system in 5 minutes       | Simplify to essential components only  |

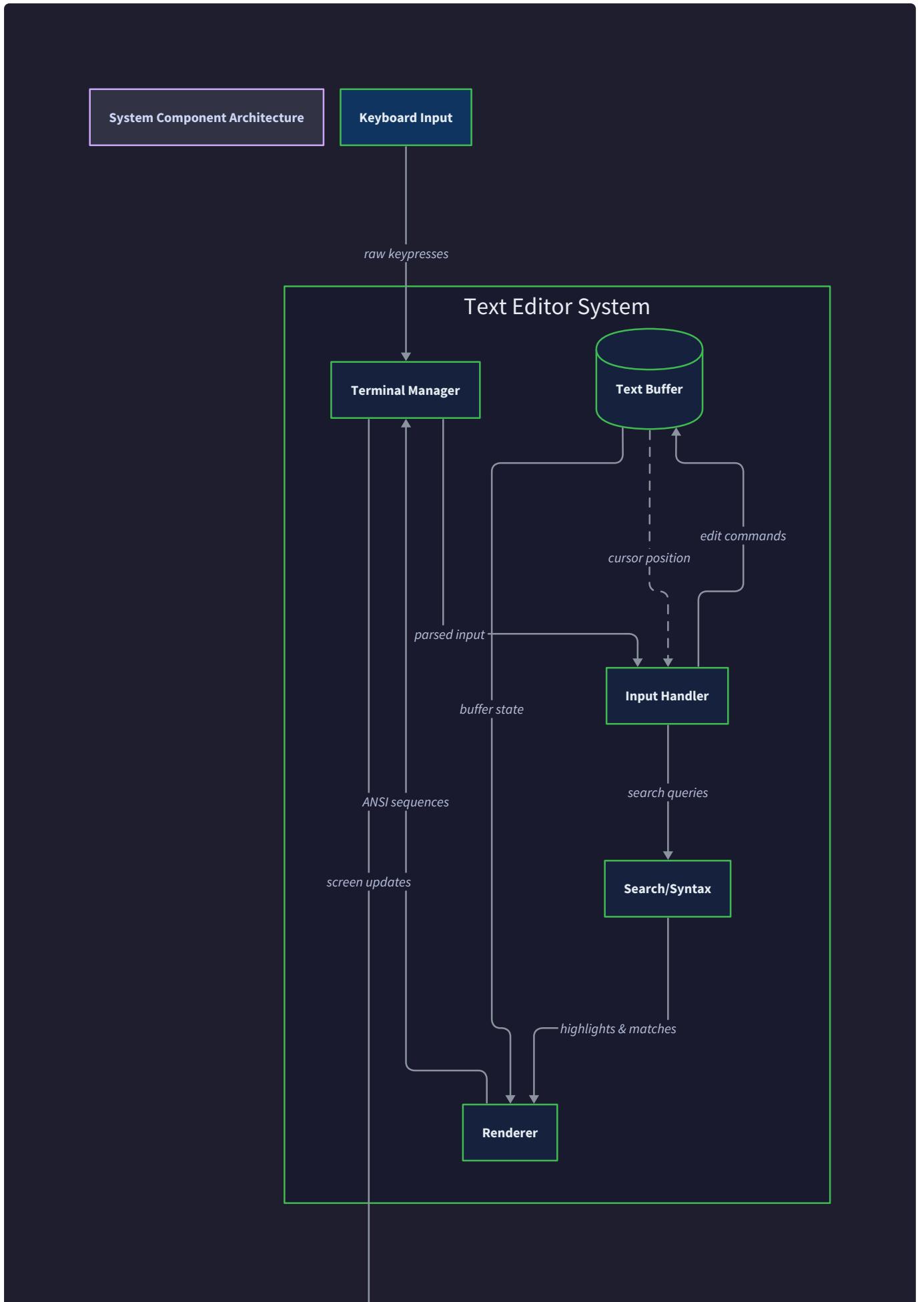
The key to successful scope management is remembering that this is a learning project, not a production editor. Every feature beyond the core goals dilutes the educational value and makes the project harder to complete successfully.

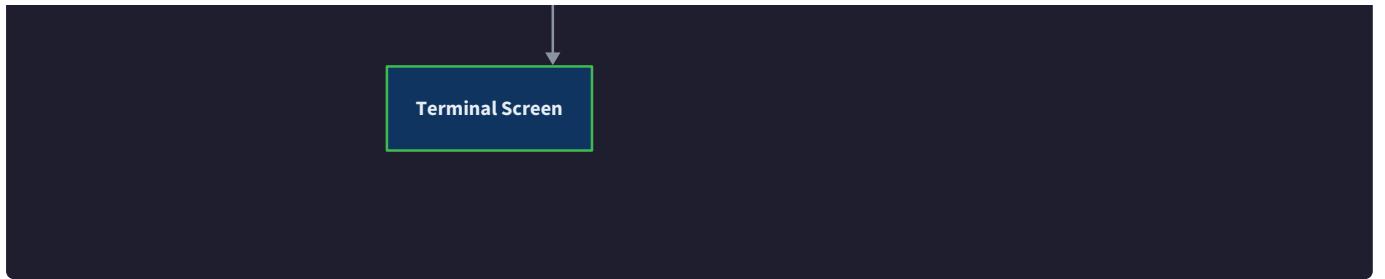
## High-Level Architecture

**Milestone(s):** Milestone 1 (Raw Mode and Input), Milestone 2 (Screen Refresh), Milestone 3 (File Viewing), Milestone 4 (Text Editing)

The architecture of our terminal-based text editor follows an immediate mode rendering pattern where each frame is explicitly constructed and sent to the terminal. Think of it like a video game engine: every frame, we read input, update our internal state, and completely redraw the screen. This is fundamentally different from GUI applications that use retained mode rendering where the framework maintains display state and only updates what changed.

The editor operates in a tight event loop: read keypress → parse command → update buffer → render screen → repeat. This cycle must be fast enough to feel responsive (ideally under 16ms per frame for 60fps) while being robust enough to handle terminal quirks, file I/O errors, and graceful cleanup. The challenge lies in coordinating between raw terminal I/O, in-memory text buffer operations, and efficient screen rendering without flicker.





## Component Overview

Our text editor architecture consists of five major components that work together to transform user keypresses into visual text editing operations. Each component has a clear responsibility and well-defined interfaces, making the system modular and testable.

### Decision: Component Separation by Responsibility

- **Context:** Terminal text editors involve multiple concerns: terminal control, input parsing, text manipulation, screen rendering, and feature logic (search, syntax highlighting)
- **Options Considered:**
  1. Monolithic approach with all logic in main()
  2. Model-View-Controller (MVC) separation
  3. Component-based architecture with single responsibilities
- **Decision:** Component-based architecture with clear interfaces
- **Rationale:** Each component can be developed, tested, and debugged independently. Terminal programming is complex enough that separation of concerns is essential for maintainability.
- **Consequences:** More files and interfaces to manage, but much easier to understand, test, and extend individual components.

| Component          | Primary Responsibility                                       | Key Data Structures                                     | Main Interfaces  |
|--------------------|--|---|--|
| Terminal Manager   | Raw mode setup, ANSI sequences, cleanup                      | <code>termios</code> original settings                  | <code>enable_raw_mode()</code> ,<br><code>disable_raw_mode()</code> ,<br><code>clear_screen()</code> ,<br><code>position_cursor()</code> |
| Input Handler      | Keypress parsing, escape sequences, command dispatch         | Key mapping tables, parser state                        | <code>editor_read_key()</code> , command dispatch functions  |
| Text Buffer        | Text content storage, editing operations, file I/O           | <code>erow_array</code> ,<br><code>editor_config</code> | <code>editor_insert_char()</code> ,<br><code>editor_delete_char()</code> , load/save functions   |
| Screen Renderer    | Viewport calculation, screen composition, flicker prevention | Frame buffer, viewport state                            | <code>editor_refresh_screen()</code> ,<br><code>hide_cursor()</code> , <code>show_cursor()</code>  |
| Feature Components | Search, syntax highlighting, undo system                     | Search state, syntax rules, undo stack                  | Feature-specific handlers integrated into main loop  |

The **Terminal Manager** acts as the foundation layer, abstracting away the complexity of terminal control. Think of it as a hardware abstraction layer that provides a clean interface for screen manipulation while handling the messy details of different terminal types and ANSI escape sequences. It owns the terminal's raw mode state and is responsible for graceful cleanup even during crashes.

The **Input Handler** serves as the translator between raw terminal input and editor commands. It must parse multi-byte escape sequences (like arrow keys), handle special key combinations (Ctrl+C, Ctrl+S), and dispatch appropriate actions. This component shields the rest of the system from the complexity of terminal input protocols.

The **Text Buffer** is the heart of the editor, maintaining the actual text content and cursor position. It provides atomic operations for text manipulation while ensuring the cursor and buffer state remain consistent. Think of it as a database for text content with ACID properties - operations either complete fully or not at all, and the buffer is never in an inconsistent state.

The **Screen Renderer** transforms the abstract text buffer into concrete terminal output. It calculates which portion of the buffer should be visible (the viewport), formats lines with syntax highlighting and line numbers, and assembles everything into a complete frame. The renderer must be efficient enough to redraw the entire screen 60 times per second without flicker.

**Feature Components** like search and syntax highlighting integrate into this architecture by hooking into specific points in the event loop. They maintain their own state and provide handlers that the main

components call at appropriate times.

## Component Interaction Patterns

The components communicate through a carefully orchestrated data flow that maintains separation of concerns while enabling rich text editing functionality. The interaction follows a strict layering where higher-level components depend on lower-level ones, but not vice versa.

**Initialization Flow:** Terminal Manager sets up raw mode → Input Handler initializes key mappings → Text Buffer loads file content → Renderer calculates initial screen layout → Main loop begins.

**Edit Operation Flow:** Input Handler receives keypress → parses into editor command → Text Buffer executes command and updates state → Renderer notices buffer changes → redraws affected screen regions → cycle repeats.

**Error Recovery Flow:** Any component detects error → calls `die()` function → Terminal Manager restores original terminal settings → process exits cleanly.

The critical architectural insight is that terminal applications require explicit cleanup in ways that GUI applications do not. The terminal state must be restored even if the program crashes, which means the Terminal Manager's cleanup functions must be registered with signal handlers and `atexit()`.

## Interface Design Principles

Each component exposes a minimal interface that hides implementation complexity while providing necessary functionality to other components. The interfaces are designed to be:

**Stateless where possible:** Functions take explicit parameters rather than relying on global state, making them easier to test and reason about.

**Atomic:** Each public function performs a complete operation that leaves the system in a consistent state. For example, `editor_insert_char()` updates both the text buffer and cursor position.

**Error-transparent:** Functions that can fail return error codes and leave the system unchanged on failure, rather than partially applying changes.

**Composable:** Simple operations can be combined to create complex behaviors. For example, the search system composes buffer scanning with renderer highlighting.

| Interface Type       | Error Handling   | State Management                           | Return Values                         |
|----------------------|--|--|---------------------------------------|
| Terminal Operations  | Fatal errors call <code>die()</code> , recoverable errors return codes | Module maintains terminal state internally | Success/failure codes                 |
| Text Operations      | Return codes, buffer unchanged on failure                              | Cursor and buffer always consistent        | Modified character/line counts        |
| Rendering Operations | Best-effort rendering, never fails                                     | Stateless viewport calculations            | Void (rendering always succeeds)      |
| Input Operations     | Invalid sequences ignored  | Parser state reset on errors               | Parsed key codes or special constants |

## Recommended File Structure

The codebase organization reflects the component architecture, with each major component in its own module and clear separation between interface definitions and implementations. This structure supports incremental development where each milestone builds upon previous components without requiring major reorganization.

```
kilo/  
|   └── src/  
|       |   └── main.c           ← Entry point and main event loop  
|       |   └── editor.h          ← Global definitions and editor_config structure  
|       |   └── terminal.c        ← Terminal Manager implementation  
|       |   └── terminal.h        ← Terminal Manager interface  
|       |   └── input.c           ← Input Handler implementation  
|       |   └── input.h           ← Key definitions and input interface  
|       |   └── buffer.c          ← Text Buffer implementation  
|       |   └── buffer.h          ← Text Buffer interface and erow structure  
|       |   └── render.c          ← Screen Renderer implementation  
|       |   └── render.h          ← Rendering interface  
|       |   └── search.c          ← Search functionality (Milestone 6)  
|       |   └── search.h          ← Search interface  
|       |   └── syntax.c          ← Syntax highlighting (Milestone 7)  
|       |   └── syntax.h          ← Syntax rules and highlighting interface  
|       └── undo.c              ← Undo system (Milestone 5)  
|       └── undo.h              ← Undo interface  
└── tests/  
    |   └── test_terminal.c     ← Terminal Manager tests  
    |   └── test_buffer.c        ← Text Buffer tests  
    |   └── test_input.c         ← Input parsing tests  
└── Makefile                  ← Build configuration  
└── README.md                 ← Project documentation
```

**Header File Organization:** Each component has a corresponding header file that defines its public interface. The `editor.h` file serves as the central header that includes all necessary system headers and defines the global `editor_config` structure that coordinates between components.

| File                    | Purpose                                  | Key Exports   | Dependencies          |
|-------------------------|--|---|-----------------------|
| <code>editor.h</code>   | Global definitions, main data structures | <code>editor_config</code> , system includes, constants | System headers only   |
| <code>terminal.h</code> | Terminal control interface               | Raw mode functions, ANSI sequence functions             | <code>editor.h</code> |
| <code>input.h</code>    | Key parsing and command dispatch         | <code>editor_read_key()</code> , key constants          | <code>editor.h</code> |
| <code>buffer.h</code>   | Text manipulation interface              | <code>erow</code> structure, buffer operations          | <code>editor.h</code> |
| <code>render.h</code>   | Screen rendering interface               | <code>editor_refresh_screen()</code> , cursor functions | All other headers     |

**Implementation File Responsibilities:** Each `.c` file implements the functions declared in its corresponding header, plus any internal helper functions needed for that component. Implementation files should never directly access data structures owned by other components - all interaction goes through the defined interfaces.

**Build Dependencies:** The Makefile compiles each source file independently and links them together, allowing incremental compilation during development. This organization supports Test-Driven Development where individual components can be tested in isolation.

### Decision: C Module Organization

- **Context:** C lacks built-in module system, so organization must be imposed through file structure and naming conventions
- **Options Considered:**
  1. Single file with all code (like original kilo)
  2. Separate files by feature (`editing.c`, `display.c`)
  3. Separate files by architectural component
- **Decision:** Component-based file organization with clear header interfaces
- **Rationale:** Matches the logical architecture, enables independent testing, and supports incremental development across milestones
- **Consequences:** More files to manage, but much cleaner separation of concerns and easier debugging

**Development Workflow:** The file structure supports building the editor incrementally across milestones:

1. **Milestone 1-2:** Implement `terminal.c`, `input.c`, and basic `render.c` for raw mode and screen refresh

2. **Milestone 3:** Add file I/O functions to `buffer.c` for viewing files
3. **Milestone 4:** Extend `buffer.c` with editing operations for text modification
4. **Milestone 5:** Add `undo.c` for undo/redo functionality and save operations
5. **Milestone 6:** Implement `search.c` for incremental search features
6. **Milestone 7:** Add `syntax.c` for syntax highlighting

Each milestone can be developed and tested independently because the interfaces between components are well-defined. This prevents the common problem in C projects where everything becomes tightly coupled and difficult to modify.

## Common File Organization Pitfalls

⚠ **Pitfall: Circular Dependencies in Headers** Many C beginners create circular dependencies where `buffer.h` includes `render.h`, which includes `buffer.h`. This causes compilation failures and indicates poor separation of concerns. Fix by ensuring dependencies flow in one direction: lower-level components (terminal, input) are included by higher-level ones (render, search) but never the reverse.

⚠ **Pitfall: Exposing Internal Data Structures** Putting implementation details like static helper functions or internal state variables in header files breaks encapsulation. Only put public interfaces in headers. Internal functions should be `static` and defined only in the `.c` file.

⚠ **Pitfall: Missing Include Guards** Every header file must have include guards (`#ifndef HEADER_H #define HEADER_H ... #endif`) to prevent multiple inclusion errors during compilation. Modern compilers support `#pragma once` as an alternative.

⚠ **Pitfall: Global State Scattered Across Files** Having global variables defined in multiple files makes the system hard to understand and debug. Centralize global state in the `editor_config` structure defined in `editor.h` and pass it explicitly to functions that need it.

## Implementation Guidance

This section provides practical guidance for organizing and implementing the component architecture in C. The focus is on creating a maintainable codebase that can grow incrementally across milestones while maintaining clean interfaces between components.

## A. Technology Recommendations

| Component        | Simple Approach                                  | Advanced Approach                       |
|------------------|--|---|
| Terminal Control | Raw <code>tcgetattr / tcsetattr</code> calls     | termios wrapper library                 |
| Input Parsing    | Simple state machine with switch statements      | Table-driven finite automaton           |
| Text Storage     | Dynamic array of strings ( <code>char**</code> ) | Gap buffer or rope data structure       |
| Screen Rendering | String concatenation with <code>snprintf</code>  | Pre-allocated frame buffer              |
| Build System     | Basic Makefile with explicit rules               | Auto-dependency generation              |
| Testing          | Manual testing with sample files                 | Automated unit tests with mock terminal |

For this project, we recommend the simple approaches. They're easier to understand, debug, and implement correctly. The advanced approaches can be explored as future extensions once the basic editor is working.

## B. Recommended File Structure Implementation

Start by creating the basic file structure with minimal implementations that compile and link together. This establishes the architecture early and prevents integration problems later.

**Core Header File ( `editor.h` ):**

```
#ifndef KILO_EDITOR_H
```

C

```
#define KILO_EDITOR_H
```

```
#define _DEFAULT_SOURCE
```

```
#define _BSD_SOURCE
```

```
#define _GNU_SOURCE
```

```
#include <termios.h>
```

```
#include <stdlib.h>
```

```
#include <stdio.h>
```

```
#include <errno.h>
```

```
#include <string.h>
```

```
#include <ctype.h>
```

```
#include <sys/ioctl.h>
```

```
#include <sys/types.h>
```

```
#include <time.h>
```

```
#include <unistd.h>
```

```
#include <stdarg.h>
```

```
#include <fcntl.h>
```

```
#define KILO_VERSION "0.0.1"
```

```
#define KILO_TAB_STOP 8
```

```
#define KILO_QUIT_TIMES 3
```

```
// Forward declarations
```

```
struct erow;
```

```
struct editor_config;
```

```
// Key definitions
```

```
enum editor_key {  
  
    BACKSPACE = 127,  
  
    ARROW_LEFT = 1000,  
  
    ARROW_RIGHT,  
  
    ARROW_UP,  
  
    ARROW_DOWN,  
  
    DEL_KEY,  
  
    HOME_KEY,  
  
    END_KEY,  
  
    PAGE_UP,  
  
    PAGE_DOWN  
  
};  
  
// Core data structures  
  
typedef struct erow {  
  
    int size;  
  
    int rsize;  
  
    char *chars;  
  
    char *render;  
  
} erow;  
  
struct editor_config {  
  
    int cx, cy;  
  
    int rx;  
  
    int rowoff;  
  
    int coloff;  
  
    int screenrows;  
  
    int screencols;
```

```
int numrows;

erow *row;

int dirty;

char *filename;

char statusmsg[80];

time_t statusmsg_time;

struct termios orig_termios;

};

extern struct editor_config E;

// Global utility function

void die(const char *s);

#endif
```

#### Main Entry Point ( `main.c` ):

```
#include "editor.h"
#include "terminal.h"
#include "input.h"
#include "buffer.h"
#include "render.h"

struct editor_config E;

void init_editor() {
    E.cx = 0;
    E.cy = 0;
    E.rx = 0;
    E.rowoff = 0;
    E.coloff = 0;
    E.numrows = 0;
    E.row = NULL;
    E.dirty = 0;
    E.filename = NULL;
    E.statusmsg[0] = '\0';
    E.statusmsg_time = 0;

    if (get_window_size(&E.screenrows, &E.screencols) == -1)
        die("get_window_size");

    E.screenrows -= 2; // Leave room for status bar and message bar
}

int main(int argc, char *argv[]) {
    enable_raw_mode();
```

C

```

init_editor();

if (argc >= 2) {
    editor_open(argv[1]);
}

editor_set_status_message("HELP: Ctrl-S = save | Ctrl-Q = quit | Ctrl-F = find");

while (1) {
    editor_refresh_screen();
    editor_process_keypress();
}

return 0;
}

void die(const char *s) {
    clear_screen();
    position_cursor(1, 1);

    perror(s);
    exit(1);
}

```

### Component Interface Skeletons:

Create header files for each component with function signatures and TODO comments:

```
// terminal.h
```

C

```
#ifndef KILO_TERMINAL_H
```

```
#define KILO_TERMINAL_H
```

```
#include "editor.h"
```

```
// Terminal setup and cleanup
```

```
int enable_raw_mode();
```

```
void disable_raw_mode();
```

```
int get_window_size(int *rows, int *cols);
```

```
// Screen manipulation
```

```
void clear_screen();
```

```
void position_cursor(int row, int col);
```

```
void hide_cursor();
```

```
void show_cursor();
```

```
#endif
```

```
// input.h
```

```
#ifndef KILO_INPUT_H
```

```
#define KILO_INPUT_H
```

```
#include "editor.h"
```

```
int editor_read_key();
```

```
void editor_process_keypress();
```

```
void editor_move_cursor(int key);
```

```
#endif
```

```
// buffer.h

#ifndef KILO_BUFFER_H

#define KILO_BUFFER_H


#include "editor.h"


// File operations

void editor_open(char *filename);

char *editor_rows_to_string(int *buflen);

void editor_save();


// Text manipulation

void editor_insert_char(int c);

void editor_insert_newline();

void editor_del_char();


// Row operations

int editor_row_cx_to_rx(erow *row, int cx);

int editor_row_rx_to_cx(erow *row, int rx);

void editor_update_row(erow *row);

void editor_insert_row(int at, char *s, size_t len);

void editor_free_row(erow *row);

void editor_del_row(int at);

void editor_row_insert_char(erow *row, int at, int c);

void editor_row_append_string(erow *row, char *s, size_t len);

void editor_row_del_char(erow *row, int at);


#endif

// render.h
```

```
#ifndef KILO_RENDER_H

#define KILO_RENDER_H

#include "editor.h"

void editor_scroll();
void editor_refresh_screen();
void editor_set_status_message(const char *fmt, ...);

#endif
```

## C. Infrastructure Starter Code

Terminal Manager ( `terminal.c` ):

```
#include "terminal.h"

int enable_raw_mode() {

    if (tcgetattr(STDIN_FILENO, &E.orig_termios) == -1)

        return -1;

    atexit(disable_raw_mode);

    struct termios raw = E.orig_termios;

    raw.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);

    raw.c_oflag &= ~(OPOST);

    raw.c_cflag |= (CS8);

    raw.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);

    raw.c_cc[VMIN] = 0;

    raw.c_cc[VTIME] = 1;

    return tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw);

}

void disable_raw_mode() {

    tcsetattr(STDIN_FILENO, TCSAFLUSH, &E.orig_termios);

}

int get_window_size(int *rows, int *cols) {

    struct winsize ws;

    if (ioctl(STDOUT_FILENO, TIOCGWINSZ, &ws) == -1 || ws.ws_col == 0) {

        return -1;
```

C

```

} else {

    *cols = ws.ws_col;

    *rows = ws.ws_row;

    return 0;

}

}

void clear_screen() {

    write(STDOUT_FILENO, "\x1b[2J", 4);

    write(STDOUT_FILENO, "\x1b[H", 3);

}

void position_cursor(int row, int col) {

    char buf[32];

    snprintf(buf, sizeof(buf), "\x1b[%d;%dH", row, col);

    write(STDOUT_FILENO, buf, strlen(buf));

}

void hide_cursor() {

    write(STDOUT_FILENO, "\x1b[?25l", 6);

}

void show_cursor() {

    write(STDOUT_FILENO, "\x1b[?25h", 6);

}

```

## D. Core Logic Skeleton Code

**Input Handler ( `input.c` ) - for learners to implement:**

```
#include "input.h"                                     C

#include "buffer.h"

#include "render.h"

int editor_read_key() {

    int nread;

    char c;

    // TODO 1: Read one character from stdin

    // TODO 2: If character is escape (\x1b), read the escape sequence

    // TODO 3: Parse escape sequence to determine special key (arrow, page up/down, etc.)

    // TODO 4: Return either the character or the special key constant

    // Hint: Arrow keys send sequences like \x1b[A, \x1b[B, \x1b[C, \x1b[D

}

void editor_process_keypress() {

    int c = editor_read_key();

    switch (c) {

        case '\r':

            // TODO: Handle Enter key - insert new line

            break;

        case CTRL_KEY('q'):

            // TODO: Handle quit - clear screen and exit

            break;

        case CTRL_KEY('s'):

    }
}
```

```
// TODO: Handle save - write buffer to file

break;

case BACKSPACE:

case CTRL_KEY('h'):

case DEL_KEY:

    // TODO: Handle delete operations

    break;

case ARROW_UP:

case ARROW_DOWN:

case ARROW_LEFT:

case ARROW_RIGHT:

    // TODO: Handle cursor movement

    editor_move_cursor(c);

    break;

default:

    // TODO: Handle regular character insertion

    break;

}

}

void editor_move_cursor(int key) {

    // TODO 1: Get current cursor position (E.cx, E.cy)

    // TODO 2: Determine new position based on key

    // TODO 3: Validate new position is within buffer bounds
```

```

    // TODO 4: Update cursor position (E.cx, E.cy)

    // TODO 5: Update render cursor position (E.rx) if moving horizontally

    // Hint: Moving past end of line should wrap to next/previous line

}

```

## E. Language-Specific Hints

### C Programming Specifics:

- Use `tcgetattr()` and `tcsetattr()` for terminal control - these are POSIX standard
- Always check return values - terminal operations can fail in many ways
- Use `atexit()` to register cleanup functions that run even on abnormal termination
- Use `write()` instead of `printf()` for terminal output - it's faster and doesn't buffer
- Allocate strings with `malloc()` and always pair with `free()` - memory leaks are common
- Use `snprintf()` instead of `sprintf()` to prevent buffer overflows
- Set `c_cc[VMIN] = 0` and `c_cc[VTIME] = 1` for non-blocking reads with timeout

### Build Configuration (Makefile):

```

CC = gcc
CFLAGS = -Wall -Wextra -Wpedantic -std=c99
TARGET = kilo
OBJS = main.o terminal.o input.o buffer.o render.o

$(TARGET): $(OBJS)
    $(CC) -o $(TARGET) $(OBJS)

%.o: %.c
    $(CC) $(CFLAGS) -c $<

clean:
    rm -f $(OBJS) $(TARGET)

.PHONY: clean

```

## Memory Management:

- Every `malloc()` must have a corresponding `free()`
- Use `realloc()` to grow the row array as lines are added
- Set pointers to NULL after freeing to catch use-after-free bugs
- Use `valgrind` to detect memory leaks during development

## F. Milestone Checkpoints

### After Milestone 1 (Raw Mode and Input):

- Run `./kilo` and verify typed characters don't appear on screen (raw mode working)
- Press arrow keys and verify they don't insert characters or move cursor yet
- Press Ctrl+C and verify it doesn't kill the program immediately
- Press Ctrl+Q and verify it exits cleanly and restores terminal

### After Milestone 2 (Screen Refresh):

- Run `./kilo` and verify screen clears and shows empty editor
- Verify status bar appears at bottom with placeholder message
- Press arrow keys and verify cursor moves (even though no content is loaded)
- Resize terminal and verify editor adapts to new dimensions

### After Milestone 3 (File Viewing):

- Run `./kilo filename.txt` and verify file contents appear
- Use arrow keys to navigate through file
- Verify long lines scroll horizontally
- Verify large files scroll vertically with line numbers

## G. Debugging Tips

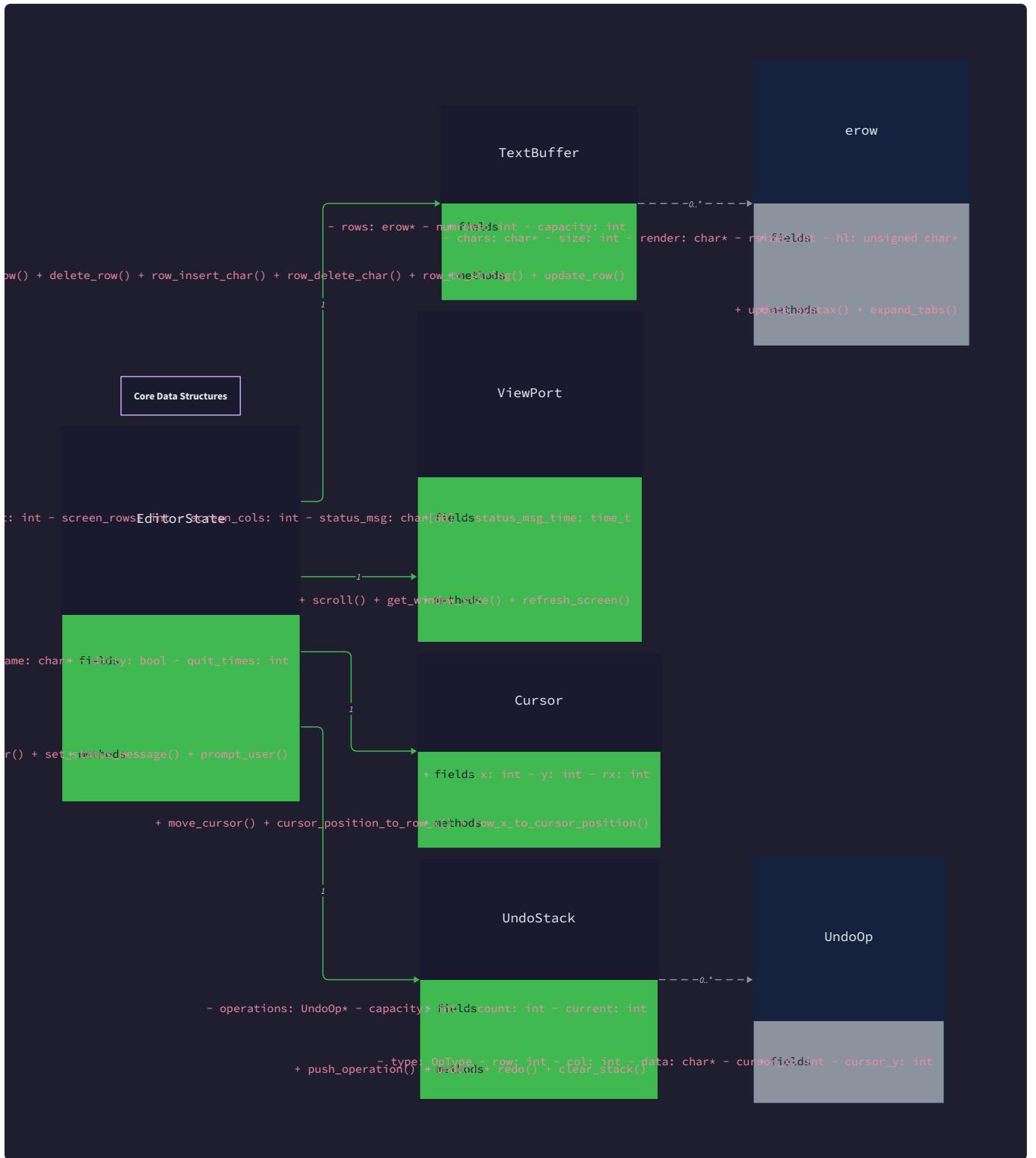
| Symptom                            | Likely Cause                | How to Diagnose                                 | Fix   |
|------------------------------------|-----------------------------|---|---|
| Terminal broken after crash        | Raw mode not disabled       | Check if <code>disable_raw_mode()</code> called | Register with <code>atexit()</code> and signal handlers |
| Characters not appearing           | Raw mode disables echo      | Expected behavior                               | Implement screen rendering to show characters           |
| Arrow keys insert weird characters | Escape sequences not parsed | Print key codes to debug                        | Implement escape sequence parser                        |
| Screen flickers during updates     | Multiple write() calls      | Use terminal profiler                           | Buffer entire frame before writing                      |
| Cursor position incorrect          | Off-by-one error            | Print cursor coordinates                        | Remember terminals use 1-based indexing                 |
| Memory crashes on large files      | Buffer overrun or leak      | Run with <code>valgrind</code>                  | Check all malloc/realloc/free pairs                     |

## Data Model

---

**Milestone(s):** Milestone 3 (File Viewing), Milestone 4 (Text Editing), Milestone 5 (Save and Undo)

The data model forms the heart of our text editor, representing all the information needed to maintain the editor's state in memory. Think of it as the editor's "working memory" — just like how your brain keeps track of what you're reading, where you are on the page, and what changes you've made while writing, our editor needs data structures to represent the text content, cursor position, viewport, and modification state.



Unlike GUI text editors that can rely on framework-managed widgets and layouts, terminal text editors must explicitly model every aspect of the editing state. The terminal provides only a grid of character cells — we must build our own abstractions for text lines, cursor positioning, scrolling regions, and content modification tracking.

## Text Buffer Representation

The **text buffer** is the core data structure that holds the actual file content in memory. Think of it like a dynamic notebook where each page represents a line of text, and you can insert new pages, remove pages, or modify the content on any page. Unlike a simple string that stores all text as one continuous sequence, our line-based approach mirrors how users conceptually think about text files — as a collection of lines.

### Decision: Line-Based Buffer Storage

- **Context:** Text content can be stored as a single large string with newline characters, or as an array of separate line strings. Terminal editors need to frequently access individual lines for rendering and navigation.
- **Options Considered:**
  1. Single string with embedded newlines
  2. Array of line strings
  3. Gap buffer with line indexing
- **Decision:** Array of line strings (`erow` structures)
- **Rationale:** Line-based access patterns dominate terminal editor operations. Rendering requires iterating through visible lines, cursor movement operates on line boundaries, and most editing operations affect individual lines. The overhead of storing line boundaries explicitly is offset by simplified line-based algorithms.
- **Consequences:** Enables O(1) access to any line for rendering, simplifies vertical scrolling calculations, but requires line splitting/joining operations for Enter/Backspace across line boundaries.

| Storage Approach | Pros   | Cons   | Chosen? |
|------------------|--|--|---------|
| Single string    | Memory efficient, simple append                  | Requires scanning for line breaks, complex line-based operations | No      |
| Array of lines   | O(1) line access, natural for terminal rendering | Extra memory overhead per line, line boundary operations         | Yes     |
| Gap buffer       | Efficient local edits, used by Emacs             | Complex implementation, less intuitive                           | No      |

Each line in our buffer is represented by an `erow` structure that contains both the original file content and a processed "render" version. The render version handles special characters like tabs, which need to be expanded to multiple spaces for display purposes.

| Field               | Type               | Description  |
|---------------------|--------------------|--|
| <code>size</code>   | <code>int</code>   | Number of characters in the original line content                      |
| <code>chars</code>  | <code>char*</code> | Dynamic array containing the actual line characters                    |
| <code>rsize</code>  | <code>int</code>   | Number of characters in the rendered version (after tab expansion)     |
| <code>render</code> | <code>char*</code> | Processed version of the line for display with tabs expanded to spaces |

The separation between `chars` and `render` addresses a fundamental challenge in text editing: the difference between logical content and visual presentation. A tab character logically represents one character in the file, but visually occupies multiple column positions on screen. By maintaining both representations, we can preserve the original file content while providing accurate cursor positioning and display formatting.

The key insight is that terminal cursor positioning operates in visual columns, not logical characters. A tab at the beginning of a line might occupy 8 visual columns, so moving the cursor "right" from before the tab to after it requires understanding both the logical position (advance 1 character) and the visual position (advance 8 columns).

The `erow` structures are stored in a dynamic array within the main editor configuration, allowing the buffer to grow and shrink as lines are added or removed. This design supports efficient insertion and deletion at any position, though operations in the middle of large files require shifting array elements.

### Memory Management Strategy:

1. Each `erow` allocates its `chars` and `render` arrays independently using dynamic allocation
2. Line insertion creates a new `erow` and shifts subsequent array elements
3. Line deletion frees the `erow` memory and compacts the array
4. Character insertion within a line reallocates that line's `chars` array
5. The main `row` array in `editor_config` grows by doubling when capacity is exceeded

## Cursor and Viewport Management

Terminal text editors face a unique challenge: the text document can be arbitrarily large, but the terminal screen shows only a small rectangular window of that content. The **cursor** represents the user's current editing position within the logical document, while the **viewport** defines which portion of the document is currently visible on screen.

Think of this relationship like looking at a large map through a small window. The cursor is a pin showing your current location on the map, while the viewport is the window frame that determines what portion of the map you can see. As you move the cursor, sometimes the window needs to shift to keep the cursor visible.

## Decision: Cursor-Follows-Viewport vs Viewport-Follows-Cursor

- **Context:** When the cursor moves beyond the visible area, the editor must decide whether to move the cursor to stay within the viewport, or move the viewport to keep the cursor visible.
- **Options Considered:**
  1. Cursor constrained to viewport (cursor-follows-viewport)
  2. Viewport tracks cursor position (viewport-follows-cursor)
  3. Hybrid approach with viewport lag
- **Decision:** Viewport-follows-cursor with immediate tracking
- **Rationale:** Users expect to navigate through arbitrarily large files without artificial movement restrictions. The cursor represents their logical position in the document, which should not be limited by screen size. The viewport is a presentation detail that should adapt to the user's navigation.
- **Consequences:** Enables seamless navigation through large files, requires viewport adjustment logic on every cursor movement, adds complexity to ensure cursor remains visible.

The cursor position is tracked using two coordinate systems simultaneously: **logical coordinates** within the document, and **visual coordinates** for screen rendering.

| Coordinate System | Field           | Type             | Description   |
|-------------------|-----------------|------------------|---|
| Logical Document  | <code>cx</code> | <code>int</code> | Cursor column position within the current line (character index)      |
| Logical Document  | <code>cy</code> | <code>int</code> | Cursor row position within the document (line number)                 |
| Visual Screen     | <code>rx</code> | <code>int</code> | Cursor column position in rendered/visual coordinates (screen column) |

The distinction between `cx` and `rx` becomes critical when dealing with tab characters. If a line contains "Hello\tWorld" and the cursor is positioned after the tab character, `cx` might be 6 (after 5 letters + 1 tab), but `rx` could be 12 (if the tab expanded to 8 spaces starting from column 5).

The viewport is defined by offset values that specify which portion of the document appears at the top-left corner of the screen:

| Field                   | Type             | Description   |
|-------------------------|------------------|---|
| <code>rowoff</code>     | <code>int</code> | Number of lines scrolled down from the document beginning                             |
| <code>coloff</code>     | <code>int</code> | Number of columns scrolled right from the document left edge                          |
| <code>screenrows</code> | <code>int</code> | Number of text rows available for document display (terminal height minus status bar) |
| <code>screencols</code> | <code>int</code> | Number of columns available for document display (terminal width minus line numbers)  |

## Viewport Adjustment Algorithm:

The viewport automatically adjusts whenever cursor movement would place the cursor outside the visible region. This ensures the cursor remains visible at all times while minimizing jarring viewport jumps.

1. **Vertical Scrolling:** If `cy` is less than `rowoff`, set `rowoff = cy` (cursor moved above visible area). If `cy` is greater than or equal to `rowoff + screenrows`, set `rowoff = cy - screenrows + 1` (cursor moved below visible area).
2. **Horizontal Scrolling:** If `rx` is less than `coloff`, set `coloff = rx` (cursor moved left of visible area). If `rx` is greater than or equal to `coloff + screencols`, set `coloff = rx - screencols + 1` (cursor moved right of visible area).
3. **Boundary Clamping:** Ensure `rowoff` never becomes negative, and `coloff` never becomes negative. For documents shorter than the screen, `rowoff` remains 0.
4. **Cursor Positioning:** After viewport adjustment, the cursor's screen position is calculated as `screen_row = cy - rowoff` and `screen_col = rx - coloff`.

The viewport adjustment algorithm implements a "cursor tracking" policy where the viewport moves the minimum distance necessary to keep the cursor visible. Alternative policies like "page-at-a-time" scrolling (moving the viewport by full screen heights) are possible but less common in modern editors.

## Cursor Movement Constraints:

Cursor movement must respect document boundaries and handle edge cases gracefully:

- **Horizontal Movement:** Cursor cannot move left of column 0 or right beyond the end of the current line
- **Vertical Movement:** Cursor cannot move above line 0 or below the last line in the document
- **Line Length Adjustment:** When moving vertically to a shorter line, the cursor column is clamped to the line's length
- **Empty Document Handling:** In an empty document, cursor remains at position (0,0)

The interaction between logical cursor position, visual rendering, and viewport management forms the foundation for all editor operations. Editing commands modify the logical document structure, cursor movement updates both coordinate systems, and viewport adjustment ensures the user can always see their current working position.

## Common Pitfalls

**⚠ Pitfall: Tab Character Cursor Positioning** When the cursor moves through a line containing tab characters, developers often forget to maintain the distinction between logical character position (`cx`) and visual column position (`rx`). For example, if a line contains "a\tb" and the cursor is after the 'b', `cx` equals 3 (position after 3 logical characters), but `rx` might equal 10 (if the tab expanded to 8 spaces). Arrow key

movement must update both coordinates correctly, and mouse clicks on the terminal must convert visual column positions back to logical positions.

**⚠ Pitfall: Viewport Scrolling Off-by-One Errors** Terminal screens use 1-indexed coordinates (row 1, column 1 is the top-left), but internal arrays use 0-indexed coordinates. When calculating viewport offsets and cursor screen positions, mixing these coordinate systems leads to cursor positioning errors where the cursor appears one row or column away from the intended position. Always use consistent 0-indexed coordinates internally and convert to 1-indexed only when generating terminal escape sequences.

**⚠ Pitfall: Memory Leaks in Dynamic Line Arrays** Each `erow` structure contains dynamically allocated `chars` and `render` arrays that must be explicitly freed when lines are deleted. Additionally, when reallocating line arrays during insertion, the old memory must be freed. Forgetting to free memory during line deletion, or during array reallocation when the buffer grows, leads to memory leaks that accumulate as the user edits large files.

**⚠ Pitfall: Inconsistent State After Failed Operations** When text editing operations fail (e.g., memory allocation failure during line insertion), the data structures can be left in an inconsistent state where cursor position, line count, and viewport offsets don't match. Always implement transactional updates where either the entire operation succeeds, or all changes are rolled back to maintain consistent state.

## Implementation Guidance

### A. Technology Recommendations

| Component         | Simple Option   | Advanced Option  |
|-------------------|---|--|
| Memory Management | <code>malloc/realloc/free</code> with manual tracking                     | Custom memory pool allocator                               |
| String Operations | Standard C string functions ( <code>strlen</code> , <code>memcpy</code> ) | Optimized string library (e.g., <code>stb_sprintf</code> ) |
| Dynamic Arrays    | Manual reallocation with doubling strategy                                | Generic dynamic array library                              |
| Tab Handling      | Fixed 8-space tab stops   | Configurable tab width with alignment                      |

### B. Recommended File Structure

```
text-editor/
src/
  main.c           ← entry point and main event loop
  editor.h         ← all structure definitions and function declarations
  buffer.c         ← text buffer operations (this component)
  terminal.c       ← raw mode and ANSI sequences
  input.c          ← keypress parsing and command dispatch
  render.c         ← screen composition and output
  file.c           ← file I/O operations
tests/
  buffer_test.c    ← unit tests for buffer operations
Makefile
```

## C. Infrastructure Starter Code

Complete buffer management utilities that handle the low-level memory management:

```
// buffer_utils.c - Memory management utilities for text buffers

#include <stdlib.h>

#include <string.h>

#include <stdio.h>

// Initialize an empty row structure

void init_erow(struct erow *row) {

    row->size = 0;

    row->chars = NULL;

    row->rsize = 0;

    row->render = NULL;

}

// Free all memory associated with a row

void free_erow(struct erow *row) {

    if (row->chars) {

        free(row->chars);

        row->chars = NULL;

    }

    if (row->render) {

        free(row->render);

        row->render = NULL;

    }

    row->size = 0;

    row->rsize = 0;

}

// Reallocate row array when buffer grows
```

C

```
struct erow *realloc_rows(struct erow *rows, int old_count, int new_count) {

    struct erow *new_rows = realloc(rows, sizeof(struct erow) * new_count);

    if (!new_rows) {

        return NULL; // Allocation failed

    }

    // Initialize new rows if expanding

    for (int i = old_count; i < new_count; i++) {

        init_erow(&new_rows[i]);

    }

    return new_rows;

}

// Calculate visual width of string with tab expansion

int calculate_render_width(const char *chars, int len, int tab_size) {

    int width = 0;

    for (int i = 0; i < len; i++) {

        if (chars[i] == '\t') {

            width += tab_size - (width % tab_size);

        } else {

            width++;

        }

    }

    return width;

}

// Convert logical cursor position to visual position
```

```

int cx_to_rx(const struct erow *row, int cx, int tab_size) {

    int rx = 0;

    for (int i = 0; i < cx && i < row->size; i++) {

        if (row->chars[i] == '\t') {

            rx += tab_size - (rx % tab_size);

        } else {

            rx++;

        }

    }

    return rx;
}

// Convert visual cursor position to logical position

int rx_to_cx(const struct erow *row, int rx, int tab_size) {

    int current_rx = 0;

    int cx;

    for (cx = 0; cx < row->size; cx++) {

        if (row->chars[cx] == '\t') {

            current_rx += tab_size - (current_rx % tab_size);

        } else {

            current_rx++;

        }

        if (current_rx > rx) break;

    }

    return cx;
}

```

## D. Core Logic Skeleton Code

Core buffer operations that learners should implement:

```
// Update the render representation of a row after chars change

void editor_update_row(struct erow *row) {

    // TODO 1: Count tabs in row->chars to calculate render size

    // TODO 2: Allocate row->render array with calculated size

    // TODO 3: Copy chars to render, expanding tabs to spaces

    // TODO 4: Update row->rsize with final render length

    // Hint: Use calculate_render_width() from utilities above

    // Hint: Tab stops are typically every 8 characters

}

// Insert a character at the specified position in a row

void editor_row_insert_char(struct erow *row, int at, int c) {

    // TODO 1: Validate 'at' is between 0 and row->size (inclusive)

    // TODO 2: Reallocate row->chars to accommodate new character

    // TODO 3: Shift characters from position 'at' one position right

    // TODO 4: Insert character 'c' at position 'at'

    // TODO 5: Update row->size and call editor_update_row()

    // Hint: Use memmove() for shifting to handle overlapping memory

}

// Delete a character at the specified position in a row

void editor_row_del_char(struct erow *row, int at) {

    // TODO 1: Validate 'at' is between 0 and row->size-1

    // TODO 2: Shift characters left to overwrite deleted character

    // TODO 3: Update row->size and call editor_update_row()

    // TODO 4: Optionally shrink row->chars allocation if significantly smaller

}
```

```

// Insert a new row at the specified position in the editor

void editor_insert_row(int at, const char *s, size_t len) {

    // TODO 1: Validate 'at' is between 0 and E.numrows (inclusive)

    // TODO 2: Reallocate E.row array to accommodate new row

    // TODO 3: Shift rows from position 'at' one position down

    // TODO 4: Initialize new row at position 'at' with string 's'

    // TODO 5: Update E.numrows and set E.dirty flag

    // TODO 6: Call editor_update_row() on the new row

    // Hint: Use reallocate_rows() utility function

}

// Delete a row at the specified position

void editor_del_row(int at) {

    // TODO 1: Validate 'at' is between 0 and E.numrows-1

    // TODO 2: Free memory for the row being deleted

    // TODO 3: Shift remaining rows up to fill the gap

    // TODO 4: Update E.numrows and set E.dirty flag

    // TODO 5: Optionally shrink E.row allocation

}

// Adjust viewport to ensure cursor is visible

void editor_scroll(void) {

    // TODO 1: Update E.rx using cx_to_rx() for current row

    // TODO 2: Check if E.cy is above viewport (cy < rowoff)

    // TODO 3: Check if E.cy is below viewport (cy >= rowoff + screenrows)

    // TODO 4: Check if E.rx is left of viewport (rx < coloff)

    // TODO 5: Check if E.rx is right of viewport (rx >= coloff + screencols)

    // TODO 6: Adjust E.rowoff and E.coloff as needed

```

```
// Hint: Viewport should move minimum distance to show cursor  
}
```

## E. Language-Specific Hints

- Use `realloc(ptr, new_size)` to grow dynamic arrays, but always check for NULL return indicating allocation failure
- `memmove(dest, src, count)` handles overlapping memory regions safely for shifting array elements
- Tab width is typically 8, but make it configurable: `#define TAB_SIZE 8`
- Use `malloc(size + 1)` for strings to ensure space for null terminator
- Set `E.dirty = 1` after any modification to track unsaved changes
- Initialize pointers to NULL and sizes to 0 to avoid accessing uninitialized memory

## F. Milestone Checkpoints

### After implementing text buffer (Milestone 3):

- Load a text file and verify each line is stored in a separate `erow` structure
- Print debug output showing `size`, `chars`, `rsize`, and `render` for a few rows
- Test with files containing tab characters - verify render version shows expanded spaces
- Navigate through the file with arrow keys - cursor should stay within line boundaries

### After implementing editing operations (Milestone 4):

- Type characters and verify they insert at cursor position
- Test backspace at various positions including start/middle/end of lines
- Test Enter key to split lines and backspace at line start to join lines
- Verify `E.dirty` flag is set after any modification

### Expected behavior:

- Character insertion shifts remaining text right and advances cursor
- Backspace removes character to the left and moves cursor left
- Enter splits current line at cursor position creating new line below
- Arrow keys move cursor without modifying text, adjusting viewport as needed

## G. Debugging Tips

| Symptom                        | Likely Cause                              | How to Diagnose                                  | Fix   |
|--------------------------------|---|--|---|
| Cursor appears in wrong column | <code>cx / rx</code> coordinate confusion | Print both values during movement                | Use <code>cx_to_rx()</code> consistently  |
| Memory corruption/crashes      | Double-free or use-after-free             | Run with valgrind or AddressSanitizer            | Check all <code>free()</code> calls have matching <code>malloc()</code>           |
| Text appears garbled           | Render array not updated after edit       | Check if <code>editor_update_row()</code> called | Call <code>editor_update_row()</code> after every <code>chars</code> modification |
| Cursor moves beyond line end   | Boundary checking missing                 | Add debug prints in movement code                | Clamp <code>cx</code> to <code>0 &lt;= cx &lt;= row-&gt;size</code>               |
| Viewport doesn't follow cursor | <code>editor_scroll()</code> not called   | Check if scroll called after movement            | Call <code>editor_scroll()</code> after every cursor position change              |

## Terminal Manager Component

**Milestone(s):** Milestone 1 (Raw Mode and Input), Milestone 2 (Screen Refresh)

The Terminal Manager Component serves as the bridge between our text editor and the underlying terminal hardware. Think of it as a translator that speaks two languages fluently: the high-level world of text editing operations and the low-level world of terminal control sequences. Just as a diplomat must understand both the formal protocols of international relations and the local customs of each country, the Terminal Manager must understand both the editor's internal state and the specific control sequences that make terminals respond correctly.

The fundamental challenge in terminal programming lies in the fact that terminals operate in two fundamentally different modes. In **canonical mode** (also called cooked mode), the terminal acts like a traditional typewriter with line buffering - characters accumulate in a buffer until the user presses Enter, and the terminal handles basic editing operations like backspace automatically. This mode works well for command-line interfaces but is completely inadequate for a text editor where we need immediate response to every keypress and complete control over cursor positioning.

In **raw mode**, the terminal becomes a bare-metal interface where every single keypress is immediately available to our program, echo is disabled (preventing double-display of characters), and special key combinations are passed through as escape sequences rather than being interpreted by the terminal. This gives us the control we need but also places the burden of handling all terminal operations squarely on our shoulders.

## Raw Mode Configuration

The transition from canonical to raw mode involves manipulating the `termios` structure, which is the POSIX standard for terminal I/O settings. Think of `termios` as the control panel for a complex piece of audio equipment - it has dozens of flags and settings that control how input and output flow through the terminal interface. Our goal is to reconfigure this control panel to give us maximum control while preserving the ability to restore the original settings when our editor exits.

The `termios` structure contains several categories of flags that control different aspects of terminal behavior. The **input flags** (`c_iflag`) control how incoming characters are processed, including automatic conversion of carriage returns to newlines and software flow control. The **output flags** (`c_oflag`) control how outgoing characters are processed, including whether carriage returns are automatically converted to carriage return/linefeed pairs. The **control flags** (`c_cflag`) manage hardware-level settings like baud rate and character size. Most importantly for our purposes, the **local flags** (`c_lflag`) control whether the terminal operates in canonical mode and whether echo is enabled.

| Flag Category                            | Key Flags for Raw Mode                | Purpose   |
|--|---------------------------------------|---|
| Input Flags<br>( <code>c_iflag</code> )  | BRKINT, ICRNL, INPCK,<br>ISTRIP, IXON | Disable break interrupts, CR-to-NL conversion, parity checking,<br>8th bit stripping, software flow control |
| Output Flags<br>( <code>c_oflag</code> ) | OPOST                                 | Disable output processing that would interfere with escape sequences  |
| Local Flags<br>( <code>c_lflag</code> )  | ECHO, ICANON, IEXTEN,<br>ISIG         | Disable echo, canonical mode, extended input processing, and signal generation                              |
| Control Characters                       | VMIN, VTIME                           | Set minimum characters for read (0) and timeout (1 decisecond)  |

The `enable_raw_mode` function must carefully preserve the original terminal attributes before making any changes. This is critical because if our program crashes without restoring the terminal state, the user's terminal session can become unusable - characters won't echo, the shell prompt may not work correctly, and the user may need to reset their terminal or start a new session.

## Decision: Store Original Terminal State Globally

- **Context:** The terminal state must be restored when the editor exits normally or crashes unexpectedly
- **Options Considered:**
  1. Pass original state through function parameters
  2. Store in global variable accessible to cleanup functions
  3. Store in editor configuration structure
- **Decision:** Use a global variable to store original `termios` state
- **Rationale:** Signal handlers and cleanup functions need access to restore terminal state, and they cannot easily receive parameters. Global access ensures cleanup can occur from any context.
- **Consequences:** Introduces global state but enables reliable cleanup from signal handlers and exit paths

The process of entering raw mode follows a specific sequence that must be executed atomically to avoid leaving the terminal in an inconsistent state:

1. Read the current terminal attributes using `tcgetattr` and store them in a global variable for later restoration
2. Create a copy of the current attributes to modify for raw mode
3. Clear the ECHO flag to prevent automatic echoing of typed characters
4. Clear the ICANON flag to disable line buffering and make characters available immediately
5. Clear the ISIG flag to prevent Ctrl+C and Ctrl+Z from generating signals that would bypass our cleanup
6. Clear input processing flags that would interfere with reading special keys
7. Clear output processing flags that would interfere with our ANSI escape sequences
8. Set VMIN to 0 and VTIME to 1 to enable non-blocking reads with a short timeout
9. Apply the modified attributes using `tcsetattr` with the TCSAFLUSH flag to ensure all pending I/O is completed

The `disable_raw_mode` function simply restores the original attributes that were saved during initialization. This function must be called from multiple contexts: normal program termination, signal handlers for crash recovery, and error handling paths.

## ANSI Escape Sequence Management

ANSI escape sequences are the command language for terminal control, analogous to the markup tags in HTML but designed for real-time terminal manipulation. These sequences all begin with the ESC character (ASCII 27) followed by specific character patterns that terminals recognize as commands rather than text to display. Think of them as remote control commands for your terminal - just as pressing a button on your TV

remote sends an infrared signal that tells the TV to change channels, sending an ANSI escape sequence tells the terminal to move the cursor, clear the screen, or change text colors.

The VT100 terminal, introduced by Digital Equipment Corporation in 1978, established many of the escape sequence standards that modern terminals still support. Our text editor relies on this backward compatibility to control cursor positioning and screen clearing across different terminal emulators. The most commonly used sequences follow the pattern `ESC[` followed by parameters and a command character.

| Escape Sequence               | Purpose                            | Parameters           | Example                  |
|-------------------------------|------------------------------------|----------------------|--------------------------|
| <code>ESC[H</code>            | Move cursor to home position (1,1) | None                 | <code>\x1b[H</code>      |
| <code>ESC[{row};{col}H</code> | Move cursor to specific position   | row, col (1-indexed) | <code>\x1b[10;25H</code> |
| <code>ESC[2J</code>           | Clear entire screen                | None                 | <code>\x1b[2J</code>     |
| <code>ESC[K</code>            | Clear from cursor to end of line   | None                 | <code>\x1b[K</code>      |
| <code>ESC[?25l</code>         | Hide cursor                        | None                 | <code>\x1b[?25l</code>   |
| <code>ESC[?25h</code>         | Show cursor                        | None                 | <code>\x1b[?25h</code>   |
| <code>ESC[{n}A</code>         | Move cursor up n lines             | n (lines)            | <code>\x1b[3A</code>     |
| <code>ESC[{n}B</code>         | Move cursor down n lines           | n (lines)            | <code>\x1b[5B</code>     |

The `clear_screen` function generates the sequence `ESC[2J` followed by `ESC[H` to both clear the entire screen buffer and move the cursor to the home position. Some terminal emulators require both operations to achieve the expected result - clearing alone might leave the cursor in an arbitrary position.

The `position_cursor` function must convert from our internal coordinate system (typically 0-indexed) to the terminal's coordinate system (1-indexed). This off-by-one conversion is a common source of bugs, particularly when positioning the cursor at the edges of the screen. The function constructs the escape sequence dynamically using string formatting to embed the row and column values.

Cursor visibility control serves an important performance and visual quality purpose. During screen updates, the cursor can flicker or appear in intermediate positions as we redraw the screen content. The `hide_cursor` and `show_cursor` functions bracket screen updates to ensure the cursor only appears in its final position, creating a smoother visual experience.

## Decision: Buffer All Escape Sequences Before Output

- **Context:** Multiple small writes to the terminal can cause visible flicker and poor performance
- **Options Considered:**
  1. Write each escape sequence immediately as generated
  2. Buffer all sequences and write in one operation
  3. Use terminal-specific double buffering features
- **Decision:** Buffer all screen content and escape sequences, then write in single operation
- **Rationale:** Reduces system calls, prevents partial screen updates from being visible, and works across all terminal types without requiring specific features
- **Consequences:** Requires managing a screen buffer but significantly improves rendering performance and eliminates flicker

The escape sequence generation functions must handle edge cases carefully. Terminal window dimensions can change while the editor is running (due to window resizing), and positioning the cursor outside the valid screen area can cause undefined behavior in some terminal emulators. The `get_window_size` function uses the `TIOCGWINSZ` ioctl to query the current terminal dimensions, providing fallback logic for terminals that don't support this query.

## Graceful Cleanup

Terminal state cleanup represents one of the most critical aspects of terminal application development. Unlike GUI applications where the windowing system can clean up after crashed applications, terminal programs that fail to restore the terminal state can leave the user's session in an unusable condition. Think of this like borrowing someone's car and changing all the radio presets and seat positions - if you don't restore everything before returning the keys, the owner faces significant inconvenience.

The cleanup system must handle three distinct scenarios: normal program termination, where the editor exits through its standard shutdown process; signal-based termination, where the operating system kills the process due to user action (Ctrl+C) or system conditions; and abnormal termination, where the program encounters a fatal error that prevents normal shutdown.

Signal handling provides the mechanism for intercepting termination requests and performing cleanup before the process actually exits. The most important signals to handle are SIGINT (typically generated by Ctrl+C), SIGTERM (sent by process managers and kill commands), and SIGQUIT (generated by Ctrl+). When any of these signals is received, our signal handler must restore the terminal state before allowing the process to terminate.

| Signal  | Source                        | Default Action           | Our Handler Action           |
|---------|-------------------------------|--------------------------|------------------------------|
| SIGINT  | Ctrl+C pressed                | Terminate process        | Restore terminal, then exit  |
| SIGTERM | kill command, process manager | Terminate process        | Restore terminal, then exit  |
| SIGQUIT | Ctrl+\ pressed                | Terminate with core dump | Restore terminal, then exit  |
| SIGSEGV | Segmentation fault            | Terminate with core dump | Restore terminal, then crash |
| SIGABRT | abort() called                | Terminate with core dump | Restore terminal, then crash |

The signal handler implementation must be extremely careful about which functions it calls. Signal handlers execute in an asynchronous context and can interrupt the main program at any point, including in the middle of library function calls. Only **async-signal-safe** functions can be safely called from signal handlers.

Fortunately, `tcsetattr` is async-signal-safe, so we can safely restore terminal attributes from a signal handler.

```
// Example of what NOT to do in a signal handler:
// - Don't call malloc, free, printf, or other non-async-signal-safe functions
// - Don't access complex data structures that might be in inconsistent state
// - Don't perform lengthy operations that could delay signal handling

// Safe signal handler pattern:

static void sigint_handler(int sig) {
    disable_raw_mode(); // only calls tcsetattr, which is async-signal-safe
    _exit(1);           // _exit is async-signal-safe, exit() is not
}
```

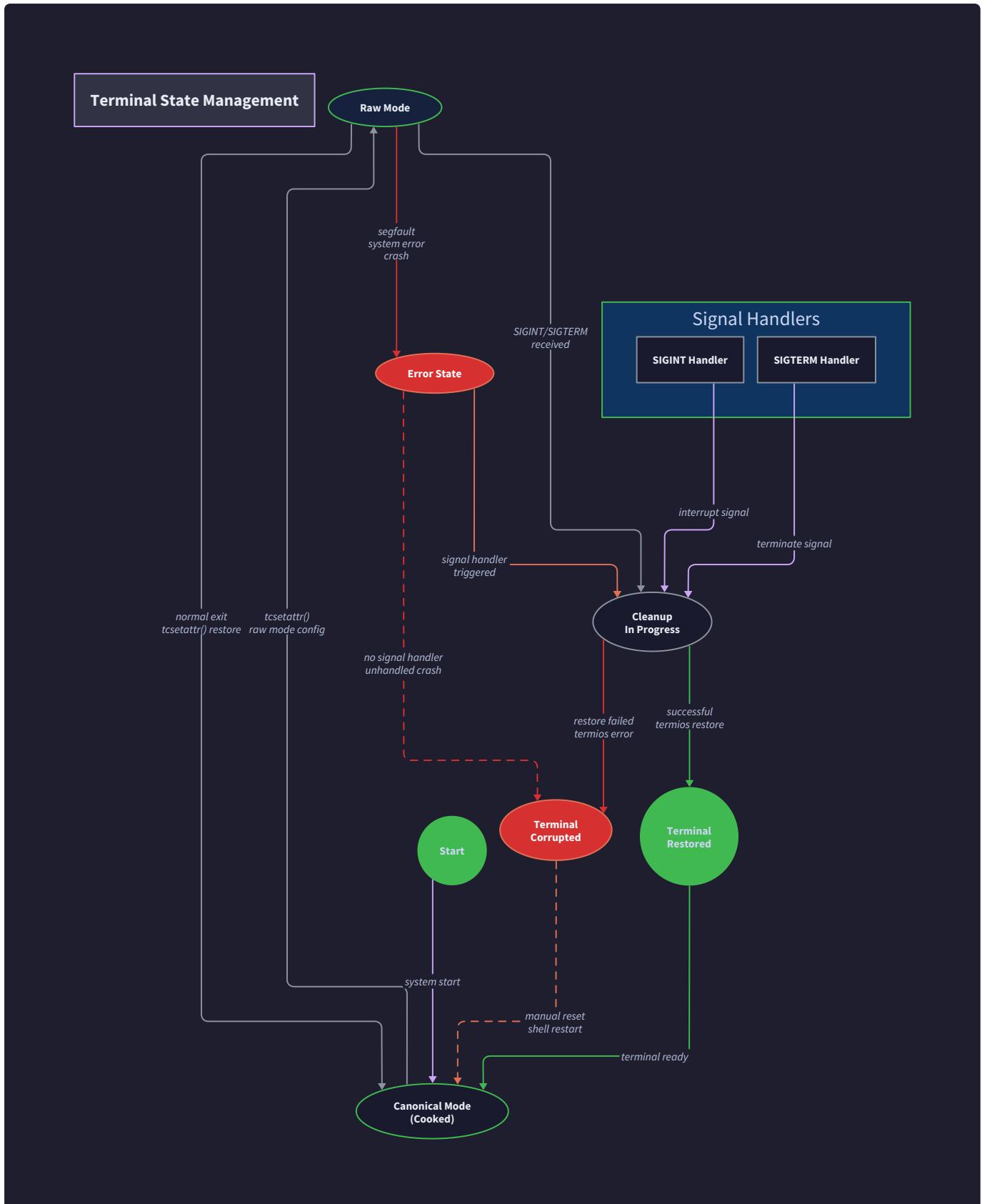
The `die` function serves as the central error handling mechanism for fatal errors. When the editor encounters an unrecoverable condition - such as inability to read from the terminal, failure to allocate memory for essential data structures, or corruption of the text buffer - it calls `die` to perform cleanup and terminate gracefully. This function must restore terminal state, optionally display an error message, and exit with a non-zero status code to indicate failure to the shell.

A robust implementation includes cleanup verification to detect cases where terminal restoration fails. Some terminal emulators or remote terminal sessions may not respond correctly to restoration attempts, and the cleanup process itself can fail due to I/O errors or system resource constraints. While we cannot always guarantee successful cleanup, we can at least detect and report these conditions.

**⚠ Pitfall: Forgetting to Handle All Exit Paths** Many implementations correctly restore terminal state for normal exit and SIGINT but forget about other termination signals or error conditions. This leaves the terminal corrupted when the editor crashes due to segmentation faults, assertion failures, or other unexpected conditions. The solution is to install signal handlers for all termination signals and ensure every error path calls the cleanup function before exiting.

**⚠ Pitfall: Race Conditions During Cleanup** If a signal arrives while the program is already in the middle of cleanup, it can cause double-cleanup attempts or corrupt the restoration process. Use atomic flags or signal masking to ensure cleanup runs exactly once, even if multiple signals arrive in rapid succession.

The cleanup system should also handle the case where terminal restoration fails. While this is rare, it can occur if the terminal connection is broken, the process runs out of file descriptors, or system resources are exhausted. In these cases, we should attempt to write a warning message to stderr (if possible) and exit with an error code that indicates the cleanup failure.



## Common Pitfalls

**⚠ Pitfall: Not Testing Cleanup Under All Conditions** Many developers test their editor under normal conditions but never verify that cleanup works when the program is killed with different signals or crashes due

to programming errors. Test cleanup by running your editor and then killing it with `kill -TERM`, `kill -INT`, and `kill -QUIT` to ensure the terminal is properly restored in all cases.

**⚠ Pitfall: Assuming Terminal Capabilities** Not all terminals support every ANSI escape sequence, and some terminal emulators have bugs in their sequence handling. Always test your editor in multiple terminal environments (xterm, gnome-terminal, tmux, screen, actual VT100-compatible terminals) and provide fallback behavior for unsupported features.

**⚠ Pitfall: Incorrect Coordinate System Conversion** Terminals use 1-indexed coordinates (top-left is 1,1) while most programming languages use 0-indexed arrays. This mismatch leads to off-by-one errors in cursor positioning, especially at screen edges. Always add 1 when converting from internal coordinates to terminal coordinates, and subtract 1 when converting from terminal to internal coordinates.

**⚠ Pitfall: Ignoring Terminal Window Resize** Terminal windows can be resized while the editor is running, invalidating cached screen dimensions and potentially causing cursor positioning outside the valid screen area. Install a SIGWINCH handler to detect window size changes and update your screen dimensions accordingly. Also bounds-check all cursor positioning operations to prevent attempting to position the cursor outside the valid screen area.

## Implementation Guidance

### Technology Recommendations

| Component             | Simple Option  | Advanced Option   |
|-----------------------|--|---|
| Terminal Attributes   | Direct termios manipulation with tcgetattr/tcsetattr | Terminfo library (ncurses) for portable terminal handling |
| Signal Handling       | Basic signal() calls for common signals              | sigaction() with fine-grained signal control and blocking |
| Escape Sequences      | Hard-coded VT100 sequences                           | Terminal capability database lookup (termcap/terminfo)    |
| Window Size Detection | TIOCGWINSZ ioctl with fallback defaults              | Full terminal capability detection and adaptation         |

## Recommended File Structure

```
text-editor/
├── src/
│   ├── terminal.c          ← Terminal Manager implementation
│   ├── terminal.h          ← Terminal Manager interface
│   ├── editor.c             ← Main editor logic
│   ├── editor.h             ← Editor state and configuration
│   └── main.c               ← Entry point and signal setup
└── tests/
    └── terminal_test.c     ← Terminal Manager unit tests
Makefile
```

The Terminal Manager should be completely isolated from the rest of the editor logic, communicating only through well-defined function interfaces. This separation allows the terminal handling code to be tested independently and potentially reused in other terminal applications.

## Infrastructure Starter Code

Here's a complete, working foundation for the Terminal Manager that handles all the low-level details:

```
// terminal.h
```

C

```
#ifndef TERMINAL_H
```

```
#define TERMINAL_H
```

```
#include <termios.h>
```

```
#include <sys/ioctl.h>
```

```
#include <unistd.h>
```

```
#include <stdlib.h>
```

```
#include <stdio.h>
```

```
#include <errno.h>
```

```
#include <string.h>
```

```
#include <signal.h>
```

```
// Terminal control constants
```

```
#define STDIN_FILENO 0
```

```
#define STDOUT_FILENO 1
```

```
#define ESCAPE_CHAR '\x1b'
```

```
// Error handling
```

```
void die(const char *s);
```

```
// Terminal mode management
```

```
int enable_raw_mode(void);
```

```
void disable_raw_mode(void);
```

```
void setup_signal_handlers(void);
```

```
// Screen control
```

```
void clear_screen(void);
```

```
void position_cursor(int row, int col);
```

```
void hide_cursor(void);

void show_cursor(void);

int get_window_size(int *rows, int *cols);

// Input handling

int editor_read_key(void);

#endif

// terminal.c - Complete implementation

#include "terminal.h"

static struct termios original_termios;

static int raw_mode_enabled = 0;

void die(const char *s) {

    // Clear screen and position cursor at top before showing error

    write(STDOUT_FILENO, "\x1b[2J", 4);

    write(STDOUT_FILENO, "\x1b[H", 3);

    // Try to restore terminal state

    disable_raw_mode();

    // Print error message with system error if available

    perror(s);

    exit(1);

}

static void signal_handler(int sig) {

    // Only call async-signal-safe functions
```

```
disable_raw_mode();

// Write error message using low-level write

const char *msg = "\r\nTerminal state restored\r\n";

write(STDERR_FILENO, msg, strlen(msg));

_exits(1); // Use _exit, not exit, in signal handlers

}

void setup_signal_handlers(void) {

    // Install handlers for all termination signals

    signal(SIGINT, signal_handler); // Ctrl+C

    signal(SIGTERM, signal_handler); // kill command

    signal(SIGQUIT, signal_handler); // Ctrl+backslash

    signal(SIGABRT, signal_handler); // abort()

    signal(SIGSEGV, signal_handler); // segmentation fault

}

int enable_raw_mode(void) {

    // Save original terminal attributes

    if (tcgetattr(STDIN_FILENO, &original_termios) == -1) {

        return -1;

    }

    // Make a copy to modify

    struct termios raw = original_termios;

    // Disable echo, canonical mode, signals, and input/output processing
```

```
raw.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);

raw.c_oflag &= ~(OPOST);

raw.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);

raw.c_cflag |= (CS8);

// Set read timeout: return after 1 character or 100ms

raw.c_cc[VMIN] = 0;

raw.c_cc[VTIME] = 1;

// Apply new attributes

if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw) == -1) {

    return -1;

}

raw_mode_enabled = 1;

return 0;

}

void disable_raw_mode(void) {

    if (raw_mode_enabled) {

        tcsetattr(STDIN_FILENO, TCSAFLUSH, &original_termios);

        raw_mode_enabled = 0;

    }

}

void clear_screen(void) {

    // Clear entire screen and move cursor to home position

    write(STDOUT_FILENO, "\x1b[2J\x1b[H", 7);
```

```
}

void position_cursor(int row, int col) {

    char buf[32];

    // Convert from 0-indexed to 1-indexed coordinates

    snprintf(buf, sizeof(buf), "\x1b[%d;%dH", row + 1, col + 1);

    write(STDOUT_FILENO, buf, strlen(buf));

}

void hide_cursor(void) {

    write(STDOUT_FILENO, "\x1b[?25l", 6);

}

void show_cursor(void) {

    write(STDOUT_FILENO, "\x1b[?25h", 6);

}

int get_window_size(int *rows, int *cols) {

    struct winsize ws;

    if (ioctl(STDOUT_FILENO, TIOCGWINSZ, &ws) == -1 || ws.ws_col == 0) {

        // Fallback: try to get size by positioning cursor at bottom-right

        if (write(STDOUT_FILENO, "\x1b[999C\x1b[999B", 12) != 12) return -1;

        return get_cursor_position(rows, cols);

    } else {

        *cols = ws.ws_col;

        *rows = ws.ws_row;

        return 0;

    }

}
```

```

}

// Helper function for window size fallback

static int get_cursor_position(int *rows, int *cols) {

    char buf[32];

    unsigned int i = 0;

    // Query cursor position

    if (write(STDOUT_FILENO, "\x1b[6n", 4) != 4) return -1;

    // Read response: ESC[row;colR

    while (i < sizeof(buf) - 1) {

        if (read(STDIN_FILENO, &buf[i], 1) != 1) break;

        if (buf[i] == 'R') break;

        i++;
    }

    buf[i] = '\0';

    // Parse response

    if (buf[0] != '\x1b' || buf[1] != '[') return -1;

    if (sscanf(&buf[2], "%d;%d", rows, cols) != 2) return -1;

    return 0;
}

```

## Core Logic Skeleton

For the main input reading function that integrates with the editor's event loop:

```

// editor_read_key - Handles both regular characters and escape sequences

// Returns: character code for regular keys, special codes for arrow keys, etc.

int editor_read_key(void) {

    // TODO 1: Read one character from stdin

    // TODO 2: If it's not an escape character, return it directly

    // TODO 3: If it's an escape character, read the next character

    // TODO 4: Handle escape sequences like ESC[A (up arrow), ESC[B (down), etc.

    // TODO 5: For unrecognized escape sequences, return the escape character

    // Hint: Use a switch statement to map escape sequences to special key codes

    // Hint: Define constants like ARROW_UP = 1000, ARROW_DOWN = 1001, etc.

}

// editor_refresh_screen - Coordinates terminal output for smooth rendering

void editor_refresh_screen(void) {

    // TODO 1: Hide cursor to prevent flicker during update

    // TODO 2: Position cursor at top-left (0,0)

    // TODO 3: Draw all visible lines from the text buffer

    // TODO 4: Draw status bar at bottom of screen

    // TODO 5: Position cursor at its logical location

    // TODO 6: Show cursor to complete the update

    // Hint: Build entire screen content in a buffer, then write once

}

```

## Language-Specific Hints

### C-Specific Considerations:

- Use `tcgetattr()` and `tcsetattr()` from `<termios.h>` for terminal control
- Always check return values from system calls - terminal operations can fail
- Use `write()` instead of `printf()` for escape sequences to avoid buffering
- The `termios` structure fields vary between systems - use feature test macros

- Signal handlers can only call async-signal-safe functions - avoid malloc, printf
- Use `snprintf()` for building escape sequences to prevent buffer overflows

### **Memory Management:**

- The `original_termios` structure should be a static global variable
- No dynamic allocation is needed for basic terminal operations
- Screen buffers for flicker prevention should be allocated once and reused
- Always null-terminate strings used with `strlen()` in escape sequence generation

### **Milestone Checkpoint**

After implementing the Terminal Manager component:

### **Verification Steps:**

1. Compile and run your editor - it should start in raw mode without echoing keystrokes
2. Type characters - they should NOT appear automatically (echo is disabled)
3. Press arrow keys - your input handler should receive escape sequences, not individual characters
4. Press Ctrl+C - the terminal should be restored to normal mode before the program exits
5. Kill the program with `kill -TERM <pid>` - terminal should still be restored properly

### **Expected Behavior:**

- Terminal prompt works normally after editor exits
- No double-echoing of characters when typing in shell after editor use
- Cursor positioning works correctly without off-by-one errors
- Screen clearing produces a blank terminal ready for editor content

### **Common Issues and Fixes:**

| Symptom  | Likely Cause                              | Fix  |
|--|---|--|
| Characters echo twice                              | Raw mode not enabled or echo not disabled | Check that ECHO flag is cleared in termios               |
| Arrow keys print characters like <code>^[[A</code> | Escape sequences not being parsed         | Implement multi-character escape sequence reading        |
| Terminal corrupted after Ctrl+C                    | Signal handler not installed              | Call <code>setup_signal_handlers()</code> in main()      |
| Cursor positioning off by one row/column           | Coordinate system mismatch                | Add 1 when converting internal coords to terminal coords |

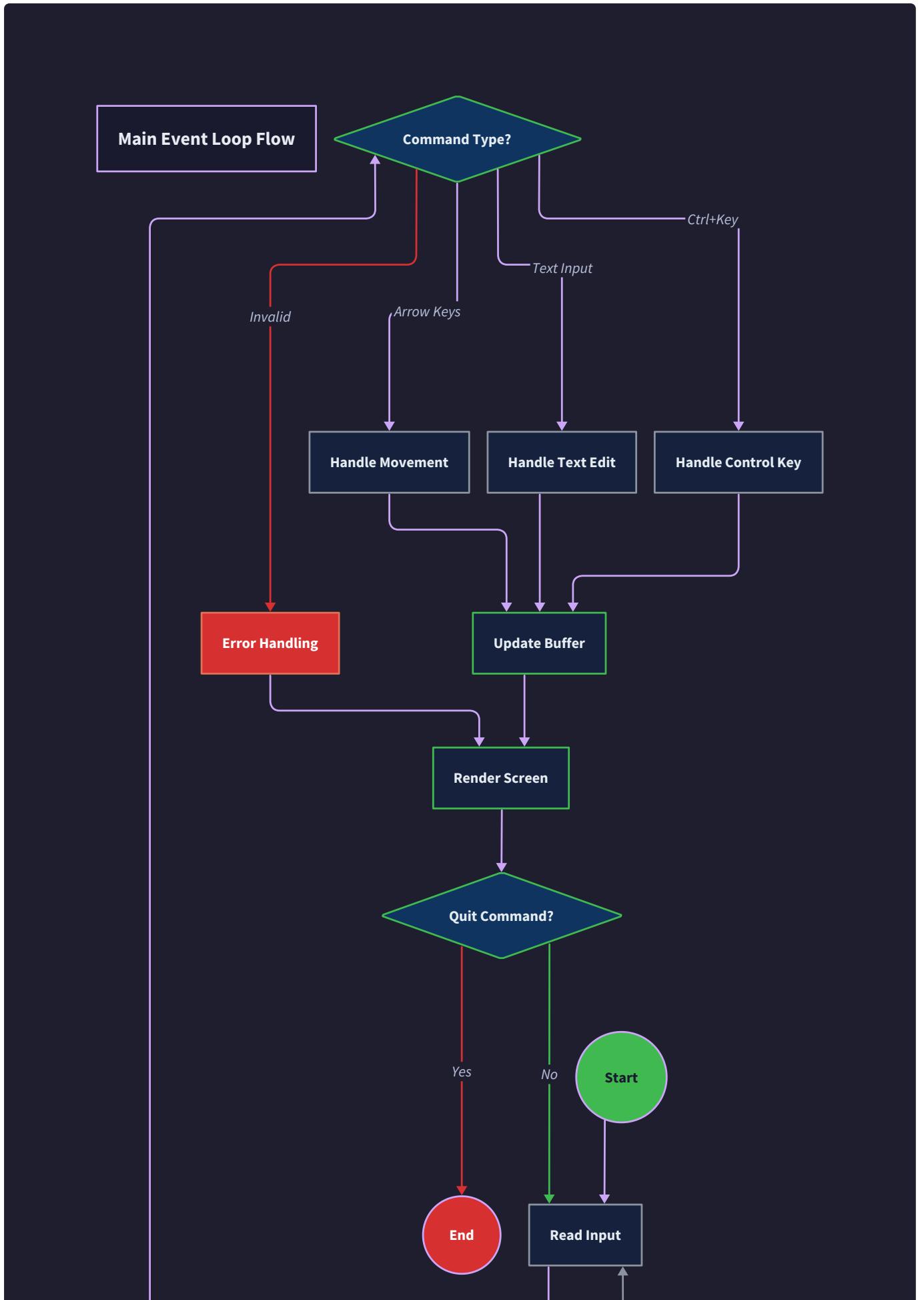
# Input Handler Component

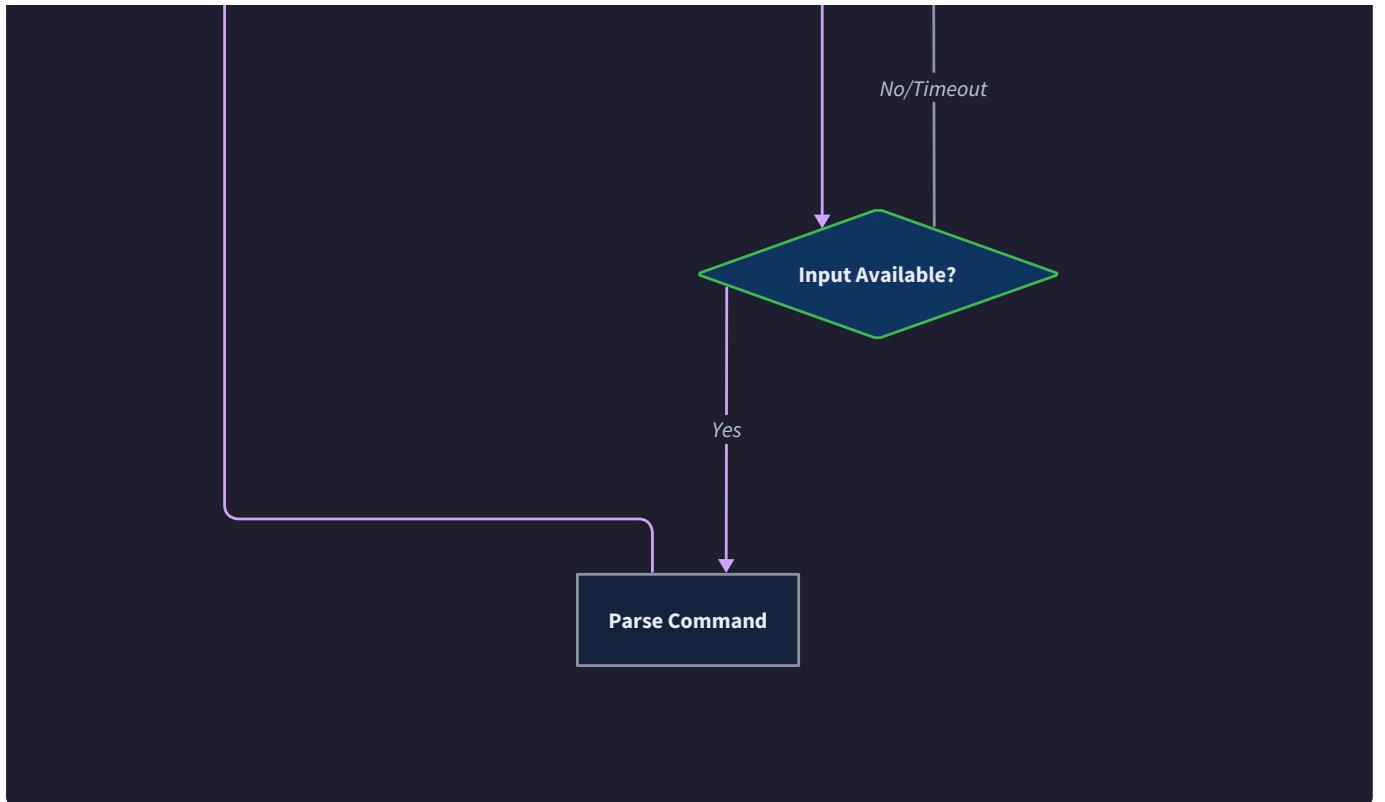
**Milestone(s):** Milestone 1 (Raw Mode and Input), Milestone 4 (Text Editing), Milestone 6 (Search), Milestone 7 (Syntax Highlighting)

The Input Handler Component serves as the central nervous system of our text editor, translating raw keyboard input into meaningful editor commands. Think of it as a sophisticated interpreter at the United Nations - it must understand dozens of different "languages" (escape sequences, control codes, special keys) and translate them into a common vocabulary that the rest of the editor can understand. Unlike GUI applications where the operating system handles much of this complexity, terminal applications must parse every single byte that comes from the keyboard, distinguishing between a simple 'A' keypress and the multi-byte sequence `\x1B[A` that represents the up arrow key.

The Input Handler sits between the Terminal Manager's raw input stream and the editor's command processing logic. It receives individual characters from `editor_read_key()`, assembles them into coherent input events, and dispatches appropriate commands to the Text Buffer, Renderer, or other components. This component must handle the inherent ambiguity of terminal input - when it receives an escape character (ASCII 27), it doesn't immediately know if this represents the Escape key being pressed or the beginning of a multi-byte sequence like an arrow key.

The architectural challenge lies in managing state across multiple keystrokes while maintaining responsiveness. The Input Handler cannot simply block and wait for complete sequences, as users expect immediate feedback when typing regular characters. Instead, it must use timeout-based parsing to distinguish between genuine escape key presses and the start of escape sequences, similar to how a telegraph operator distinguishes between intentional pauses and incomplete messages.





## Keypress Parsing

The keypress parsing subsystem transforms the raw stream of bytes from the terminal into discrete input events. Think of this process like parsing a complex telegraph transmission where some messages are single characters while others are multi-part codes that must be assembled before interpretation. The fundamental challenge is that terminal input arrives as a sequence of individual bytes, but some logical keypresses (like arrow keys or function keys) are encoded as multiple bytes that must be collected and interpreted as a unit.

The parsing process operates as a state machine with three primary states: waiting for input, collecting an escape sequence, and processing special multi-byte sequences. When `editor_read_key()` receives a regular ASCII character (values 32-126), it can immediately return that character to the caller. However, when it encounters an escape character (ASCII 27), the parser must transition into sequence collection mode and attempt to read additional bytes within a timeout window.

## Decision: Timeout-Based Escape Sequence Parsing

- **Context:** Terminal input streams cannot distinguish between a user pressing the Escape key and the beginning of a multi-byte escape sequence without additional context
- **Options Considered:**
  1. Block indefinitely waiting for sequence completion
  2. Use timeout-based parsing with configurable delay
  3. Require users to press sequences quickly without timeout
- **Decision:** Implement timeout-based parsing with a short delay (typically 100ms)
- **Rationale:** This provides the best user experience by allowing both genuine Escape key presses and proper handling of arrow keys and function keys without blocking the interface
- **Consequences:** Introduces timing complexity but enables natural keyboard interaction patterns

The timeout mechanism works by setting the terminal's `VTIME` field in the `termios` structure to create a brief delay after receiving an escape character. If additional bytes arrive within this window, they're treated as part of an escape sequence. If the timeout expires with no additional input, the original escape character is treated as a standalone Escape key press.

### Escape Sequence Recognition Table:

| Sequence             | Bytes           | Meaning     | Action                           |
|----------------------|-----------------|-------------|----------------------------------|
| <code>\x1B[A</code>  | 27, 91, 65      | Up Arrow    | Move cursor up one line          |
| <code>\x1B[B</code>  | 27, 91, 66      | Down Arrow  | Move cursor down one line        |
| <code>\x1B[C</code>  | 27, 91, 67      | Right Arrow | Move cursor right one character  |
| <code>\x1B[D</code>  | 27, 91, 68      | Left Arrow  | Move cursor left one character   |
| <code>\x1B[H</code>  | 27, 91, 72      | Home        | Move cursor to beginning of line |
| <code>\x1B[F</code>  | 27, 91, 70      | End         | Move cursor to end of line       |
| <code>\x1B[3~</code> | 27, 91, 51, 126 | Delete      | Delete character at cursor       |
| <code>\x1B[5~</code> | 27, 91, 53, 126 | Page Up     | Scroll up one screen             |
| <code>\x1B[6~</code> | 27, 91, 54, 126 | Page Down   | Scroll down one screen           |

The parsing algorithm follows these numbered steps:

1. Read a single character from the terminal input stream using system `read()` call
2. If the character is a regular ASCII printable character (32-126), return it immediately
3. If the character is a control character (0-31 or 127), check if it's a recognized control key

4. If the character is an escape character (27), enter sequence collection mode
5. Set a short timeout and attempt to read the next character in the sequence
6. If no additional character arrives before timeout, treat the escape as a standalone key
7. If additional characters arrive, continue reading until a complete sequence is recognized
8. Validate the assembled sequence against the known escape sequence table
9. Return the corresponding logical key code or treat as an unknown sequence
10. Handle any partial sequences by either waiting for completion or treating as separate keys

The parser must also handle various terminal emulator quirks and variations. Some terminals send slightly different sequences for the same logical keys, and the parser includes compatibility mappings for common variations. For example, some terminals send `\x1B[1~` for Home instead of `\x1B[H`, and the parser must recognize both patterns.

### **Special Character Handling Table:**

| Character         | ASCII Value | Interpretation  | Processing   |
|-------------------|-------------|-----------------|--|
| <code>\x03</code> | 3           | Ctrl+C          | Signal interrupt (usually exit)                      |
| <code>\x04</code> | 4           | Ctrl+D          | End of file or exit                                  |
| <code>\x08</code> | 8           | Backspace       | Delete character before cursor                       |
| <code>\x09</code> | 9           | Tab             | Insert tab or move to next tab stop                  |
| <code>\x0A</code> | 10          | Line Feed       | Insert new line                                      |
| <code>\x0D</code> | 13          | Carriage Return | Insert new line (same as LF)                         |
| <code>\x1B</code> | 27          | Escape          | Begin escape sequence or standalone escape           |
| <code>\x7F</code> | 127         | Delete (DEL)    | Delete character before cursor (alternate backspace) |

## **Special Key Mapping**

The special key mapping subsystem translates recognized escape sequences and control characters into editor commands. Think of this as a sophisticated dispatch table that maps the diverse vocabulary of terminal input into the standardized command language that the editor understands. The challenge is that the same logical action (like "move cursor left") might be triggered by multiple different input patterns - the left arrow key, Ctrl+B in some editors, or even Ctrl+H in legacy systems.

The mapping system uses a two-stage approach: first, raw input is normalized into logical key identifiers, then these identifiers are mapped to editor commands based on the current editor mode. This separation allows the same physical key to perform different actions in different contexts - for example, the 'i' key inserts the letter 'i' in normal editing mode but might trigger "incremental search" when preceded by Ctrl+S.

## Decision: Mode-Aware Command Mapping

- **Context:** Different editor contexts require the same keys to perform different actions (normal editing vs search mode vs command mode)
- **Options Considered:**
  1. Single global key mapping table with context checking in each command handler
  2. Separate mapping tables for each editor mode with mode-based dispatch
  3. Dynamic key binding system with user-configurable mappings
- **Decision:** Separate mapping tables for each major editor mode with centralized mode management
- **Rationale:** This provides clean separation of concerns, makes the code more maintainable, and allows for clear mode-specific behavior without complex conditional logic
- **Consequences:** Requires careful mode state management but results in cleaner command handling logic

The key mapping tables are organized by editor mode, with each mode defining its own interpretation of input events. The normal editing mode handles standard text input and cursor movement, while search mode reinterprets many keys for search navigation and query modification.

### Normal Mode Key Mapping Table:

| Input       | Key Code      | Command            | Parameters                         |
|-------------|---------------|--------------------|------------------------------------|
| Arrow Up    | KEY_UP        | CMD_MOVE_CURSOR    | direction: up, count: 1            |
| Arrow Down  | KEY_DOWN      | CMD_MOVE_CURSOR    | direction: down, count: 1          |
| Arrow Left  | KEY_LEFT      | CMD_MOVE_CURSOR    | direction: left, count: 1          |
| Arrow Right | KEY_RIGHT     | CMD_MOVE_CURSOR    | direction: right, count: 1         |
| Home        | KEY_HOME      | CMD_MOVE_CURSOR    | direction: line_start              |
| End         | KEY_END       | CMD_MOVE_CURSOR    | direction: line_end                |
| Page Up     | KEY_PAGE_UP   | CMD_SCROLL         | direction: up, count: screenrows   |
| Page Down   | KEY_PAGE_DOWN | CMD_SCROLL         | direction: down, count: screenrows |
| Backspace   | KEY_BACKSPACE | CMD_DELETE_CHAR    | direction: backward                |
| Delete      | KEY_DELETE    | CMD_DELETE_CHAR    | direction: forward                 |
| Enter       | KEY_ENTER     | CMD_INSERT_NEWLINE | position: cursor                   |
| Ctrl+S      | CTRL_S        | CMD_SAVE_FILE      | filename: current                  |
| Ctrl+Q      | CTRL_Q        | CMD_QUIT           | force: false                       |
| Ctrl+F      | CTRL_F        | CMD_ENTER_SEARCH   | mode: forward                      |

The mapping system includes support for key combinations and modifiers. Control key combinations are detected by masking the input character with 0x1F, which converts control characters back to their base letter equivalents. For example, Ctrl+A produces ASCII value 1, which corresponds to 'A' & 0x1F.

#### Search Mode Key Mapping Table:

| Input           | Key Code      | Command                | Parameters          |
|-----------------|---------------|------------------------|---------------------|
| Printable chars | KEY_CHAR      | CMD_SEARCH_ADD_CHAR    | character: input    |
| Backspace       | KEY_BACKSPACE | CMD_SEARCH_DELETE_CHAR | direction: backward |
| Enter           | KEY_ENTER     | CMD_SEARCH_CONFIRM     | action: goto_match  |
| Escape          | KEY_ESCAPE    | CMD_SEARCH_CANCEL      | restore: true       |
| Ctrl+G          | CTRL_G        | CMD_SEARCH_NEXT        | direction: forward  |
| Ctrl+T          | CTRL_T        | CMD_SEARCH_PREV        | direction: backward |
| Arrow Up        | KEY_UP        | CMD_SEARCH_PREV        | direction: backward |
| Arrow Down      | KEY_DOWN      | CMD_SEARCH_NEXT        | direction: forward  |

The key mapping system must handle several edge cases and special situations. When the editor is in search mode, most printable characters are added to the search query rather than inserted into the text buffer. However, certain control characters like Ctrl+C still need to maintain their system-level meanings even in specialized modes.

### Key Classification Algorithm:

1. Read the raw character or key code from the keypress parser
2. Determine the current editor mode from the global editor state
3. Check if the input represents a mode-specific override (like Escape canceling search)
4. If not an override, look up the input in the current mode's mapping table
5. If found in mode table, return the mapped command with appropriate parameters
6. If not found in mode table, check the global/fallback mapping table
7. For printable characters not explicitly mapped, default to character insertion
8. For unmapped control characters or escape sequences, either ignore or show error
9. Apply any pending key repeat counts or modifiers to the final command
10. Return the complete command structure ready for dispatch

### Command Dispatching

The command dispatching subsystem routes parsed and mapped commands to the appropriate handler components within the editor. Think of this as the central switchboard in a large organization - it receives standardized command messages and knows exactly which department (component) should handle each type of request. The dispatcher must ensure commands are executed in the correct order, handle any interdependencies between commands, and coordinate the screen refresh process after state-changing operations.

The dispatcher operates on command objects that contain both the action to be performed and any necessary parameters. This approach separates the concerns of input interpretation (handled by the keypress parser and key mapper) from command execution (handled by the various editor components). The dispatcher serves as a clean interface boundary that could potentially support command queuing, macro recording, or undo functionality in the future.

### Decision: Centralized Command Dispatching with Component Interfaces

- **Context:** Commands need to be routed to different components (Text Buffer, Renderer, Search System) with proper coordination of screen updates
- **Options Considered:**
  1. Direct function calls from input handler to each component
  2. Centralized dispatcher with standardized command objects
  3. Event-driven system with publish/subscribe patterns
- **Decision:** Centralized dispatcher with command objects and explicit component interfaces
- **Rationale:** This provides clean separation between input handling and command execution, makes the code easier to test and debug, and creates a natural place for future features like command history or macros
- **Consequences:** Adds a layer of indirection but significantly improves code organization and maintainability

The command dispatcher maintains references to all major editor components and routes commands based on their type and current editor state. Each command type has a designated primary handler, but commands may trigger cascading effects that require coordination between multiple components.

#### Command Routing Table:

| Command Type        | Primary Handler | Secondary Effects                             | Screen Refresh |
|---------------------|-----------------|---|----------------|
| CMD_MOVE_CURSOR     | Text Buffer     | Viewport adjustment via Renderer              | Yes            |
| CMD_INSERT_CHAR     | Text Buffer     | Dirty flag, undo stack, syntax highlighting   | Yes            |
| CMD_DELETE_CHAR     | Text Buffer     | Dirty flag, undo stack, line joining          | Yes            |
| CMD_INSERT_NEWLINE  | Text Buffer     | Dirty flag, undo stack, line splitting        | Yes            |
| CMD_SAVE_FILE       | Text Buffer     | File I/O, dirty flag clearing, status message | Yes            |
| CMD_QUIT            | Main Editor     | Dirty check, terminal cleanup                 | No (exiting)   |
| CMD_ENTER_SEARCH    | Search System   | Mode change, status bar update                | Yes            |
| CMD_SEARCH_ADD_CHAR | Search System   | Query update, match highlighting              | Yes            |
| CMD_SEARCH_NEXT     | Search System   | Cursor movement, match navigation             | Yes            |
| CMD_SEARCH_CANCEL   | Search System   | Mode change, cursor restoration               | Yes            |
| CMD_SCROLL          | Renderer        | Viewport adjustment only                      | Yes            |
| CMD_REFRESH_SCREEN  | Renderer        | Complete screen redraw                        | Yes            |

The dispatching process follows a carefully orchestrated sequence to maintain editor consistency:

1. Validate the incoming command object and verify all required parameters are present
2. Check if the command is valid in the current editor state and mode
3. For state-changing commands, create an undo record before executing the operation
4. Route the command to its primary handler component for execution
5. If the primary handler succeeds, trigger any necessary secondary effects
6. Update editor-wide state like the dirty flag, cursor position, or mode indicators
7. Determine if a screen refresh is needed based on the command type and execution result
8. If refresh is needed, call `editor_refresh_screen()` to update the terminal display
9. Handle any error conditions by restoring previous state if possible
10. Update the status bar with any relevant messages or state indicators

#### Command Structure Definition:

| Field       | Type          | Description   |
|-------------|---------------|---|
| type        | CommandType   | Enumerated value identifying the command action             |
| parameters  | CommandParams | Union of parameter structures specific to each command type |
| source_line | int           | Source code line for debugging (development builds only)    |
| timestamp   | time_t        | When command was created for undo/redo timing               |

#### Movement Command Parameters:

| Field     | Type          | Description   |
|-----------|---------------|---|
| direction | MoveDirection | Up, down, left, right, line_start, line_end, file_start, file_end |
| count     | int           | Number of units to move (for repeated commands)                   |
| unit      | MoveUnit      | Character, word, line, page, or screen for movement granularity   |

#### Text Modification Command Parameters:

| Field     | Type      | Description   |
|-----------|-----------|---|
| character | int       | Character to insert (for insert commands)               |
| position  | CursorPos | Where to perform the operation (usually current cursor) |
| length    | int       | Number of characters to delete (for deletion commands)  |
| text      | char*     | String content for multi-character operations           |

The dispatcher includes sophisticated error handling to deal with commands that cannot be executed in the current state. For example, attempting to delete characters when the cursor is at the beginning of an empty file, or trying to move up when already at the first line. These situations are handled gracefully by either ignoring the command or providing user feedback through the status bar.

#### Error Handling in Command Dispatch:

| Error Condition                 | Detection Method                     | Recovery Action                               | User Feedback               |
|---------------------------------|--------------------------------------|---|-----------------------------|
| Invalid cursor movement         | Boundary checking in Text Buffer     | Ignore command, maintain current position     | None (silent)               |
| File save failure               | Return code from file I/O operations | Maintain dirty flag, preserve unsaved changes | Error message in status bar |
| Memory allocation failure       | NULL return from malloc/realloc      | Abort operation, preserve existing data       | "Out of memory" message     |
| Search string not found         | Search function returns no matches   | Restore cursor to search start position       | "Pattern not found" message |
| Invalid command in current mode | Mode validation before dispatch      | Ignore command or switch modes as appropriate | Mode indicator update       |

**⚠ Pitfall: Screen Refresh Coordination** A common mistake is calling `editor_refresh_screen()` after every individual command, which can cause significant performance problems and visual flicker. Instead, the dispatcher should batch screen updates and refresh only after all related commands have been executed. For example, when processing a paste operation that inserts multiple lines, refresh should happen once after all lines are inserted rather than after each individual line insertion.

**⚠ Pitfall: Command Validation Timing** Another frequent error is validating commands too late in the dispatch process, after some state changes have already been made. All validation should happen before any state modifications begin, and if validation fails, no changes should be made to the editor state. This prevents the editor from entering inconsistent intermediate states that are difficult to recover from.

The dispatcher also handles special coordination requirements between components. For example, when a text modification command changes the document structure, it must inform the syntax highlighter to re-scan the affected regions, update the undo system with the change record, and potentially adjust the viewport if the cursor has moved outside the visible area.

## Implementation Guidance

The Input Handler Component requires careful attention to terminal I/O timing and state management. The implementation must balance responsiveness with correct parsing of multi-byte sequences.

### A. Technology Recommendations:

| Component        | Simple Option   | Advanced Option   |
|------------------|---|---|
| Input Reading    | Raw <code>read()</code> system call with <code>termios</code> | <code>select()</code> or <code>poll()</code> for non-blocking I/O |
| Key Mapping      | Static lookup tables with switch statements                   | Dynamic hash table with configurable bindings                     |
| Command Objects  | Simple structs with unions                                    | Object-oriented command pattern with inheritance                  |
| State Management | Global variables for editor state                             | Encapsulated state machine with explicit transitions              |

## B. Recommended File Structure:

```

src/
  input/
    input.c          ← main input handling logic
    input.h          ← public interface definitions
    keymap.c         ← key mapping tables and lookup
    commands.c       ← command creation and validation
    parser.c         ← escape sequence parsing
    editor.h         ← global editor state and config
    main.c           ← main event loop
  
```

## C. Infrastructure Starter Code:

```
// input/input.h - Complete header definitions

#ifndef INPUT_H

#define INPUT_H


#include <time.h>

// Key codes for special keys (not printable ASCII)

enum EditorKey {

    KEY_NULL = 0,


    // Arrow keys

    ARROW_LEFT = 1000, // Start at high value to avoid ASCII conflicts
    ARROW_RIGHT,
    ARROW_UP,
    ARROW_DOWN,


    // Function keys

    DELETE_KEY,
    HOME_KEY,
    END_KEY,
    PAGE_UP,
    PAGE_DOWN,


    // Control keys (using ASCII values directly)

    CTRL_A = 1, // 'A' & 0x1f
    CTRL_B = 2,
    CTRL_C = 3,
    CTRL_D = 4,
```

C

```
CTRL_E = 5,  
CTRL_F = 6,  
CTRL_G = 7,  
CTRL_H = 8,  
CTRL_I = 9,  
CTRL_J = 10,  
CTRL_K = 11,  
CTRL_L = 12,  
CTRL_M = 13,  
CTRL_N = 14,  
CTRL_O = 15,  
CTRL_P = 16,  
CTRL_Q = 17,  
CTRL_R = 18,  
CTRL_S = 19,  
CTRL_T = 20,  
CTRL_U = 21,  
CTRL_V = 22,  
CTRL_W = 23,  
CTRL_X = 24,  
CTRL_Y = 25,  
CTRL_Z = 26,  
  
ESC_KEY = 27,  
BACKSPACE = 127  
};
```

```
// Editor modes that affect key interpretation

typedef enum {

    MODE_NORMAL,           // Regular text editing

    MODE_SEARCH,           // Incremental search active

    MODE_COMMAND,          // Command entry mode (future extension)

    MODE_REPLACE           // Find and replace mode (future extension)

} EditorMode;

// Command types for dispatching

typedef enum {

    CMD_NONE,

    CMD_INSERT_CHAR,

    CMD_DELETE_CHAR,

    CMD_MOVE_CURSOR,

    CMD_INSERT_NEWLINE,

    CMD_SAVE_FILE,

    CMD_QUIT,

    CMD_ENTER_SEARCH,

    CMD_EXIT_SEARCH,

    CMD_SEARCH_NEXT,

    CMD_SEARCH_PREV,

    CMD_REFRESH_SCREEN

} CommandType;

// Movement directions for cursor commands

typedef enum {

    MOVE_UP,

    MOVE_DOWN,
```

```
MOVE_LEFT,
MOVE_RIGHT,
MOVE_LINE_START,
MOVE_LINE_END,
MOVE_FILE_START,
MOVE_FILE_END

} MoveDirection;

// Union of all possible command parameters
typedef union {

    struct {

        int character;

    } insert;

    struct {

        MoveDirection direction;

        int count;

    } move;

    struct {

        int forward; // 1 for forward, 0 for backward

    } delete;

    struct {

        char* filename;

    } save;

} CommandParams;
```

```
// Complete command structure

typedef struct {

    CommandType type;

    CommandParams params;

    time_t timestamp;

} Command;

// Public interface functions

int editor_read_key(void);

Command input_to_command(int key);

void execute_command(Command* cmd);

void set_editor_mode(EditorMode mode);

EditorMode get_editor_mode(void);

#endif
```

#### D. Core Logic Skeleton Code:

```
// input/input.c - Core input handling implementation C

#include "input.h"

#include "../editor.h"

#include <unistd.h>

#include <errno.h>

static EditorMode current_mode = MODE_NORMAL;

// Read and parse keyboard input including escape sequences

int editor_read_key(void) {

    int nread;

    char c;

    // TODO 1: Read single character from STDIN_FILENO using read() system call

    // TODO 2: Handle EAGAIN/EINTR errors by retrying the read

    // TODO 3: If character is not escape (27), return it directly

    // TODO 4: If character is escape, enter sequence parsing mode

    // TODO 5: Set short timeout using termios VTIME for sequence completion

    // TODO 6: Read additional characters to build complete escape sequence

    // TODO 7: Match complete sequence against known patterns in escape_sequences[]

    // TODO 8: Return appropriate key code or treat as plain escape if no match

    // Hint: Use a loop to collect sequence chars, max 3-4 chars for most sequences

    // Hint: Store escape sequences in a lookup table for easy matching

}

// Convert raw key input to editor command based on current mode

Command input_to_command(int key) {

    Command cmd = {CMD_NONE, {0}, time(NULL)};


```

```

// TODO 1: Check current_mode to determine which key mapping table to use

// TODO 2: For MODE_NORMAL, handle arrow keys as cursor movement commands

// TODO 3: For MODE_NORMAL, handle printable chars (32-126) as insert commands

// TODO 4: For MODE_NORMAL, handle special keys like backspace, delete, enter

// TODO 5: For MODE_SEARCH, handle printable chars as search query additions

// TODO 6: For MODE_SEARCH, handle escape as search cancellation

// TODO 7: For MODE_SEARCH, handle enter as search confirmation

// TODO 8: Handle control keys (Ctrl+S, Ctrl+Q, etc) in all modes

// TODO 9: Fill in appropriate CommandParams based on command type

// TODO 10: Return completed command structure

// Hint: Use switch statements for clean key mapping logic

// Hint: Control keys can be detected with (key & 0x1f) for letter keys

}

// Execute a parsed command by routing to appropriate handler

void execute_command(Command* cmd) {

    if (!cmd) return;

    // TODO 1: Validate command type and parameters before execution

    // TODO 2: For CMD_INSERT_CHAR, call editor_insert_char() with character

    // TODO 3: For CMD_DELETE_CHAR, call appropriate delete function (forward/backward)

    // TODO 4: For CMD_MOVE_CURSOR, call editor_move_cursor() with direction and count

    // TODO 5: For CMD_INSERT_NEWLINE, call editor_insert_newline()

    // TODO 6: For CMD_SAVE_FILE, call editor_save() and update status message

    // TODO 7: For CMD_QUIT, perform dirty check and call exit if safe

    // TODO 8: For search commands, route to search system functions

```

```

    // TODO 9: After state-changing commands, call editor_refresh_screen()

    // TODO 10: Handle any errors and update status bar with messages

    // Hint: Keep track of which commands require screen refresh

    // Hint: Use extern declarations to access global editor_config

}

// Change editor mode and update key mapping behavior

void set_editor_mode(EditorMode mode) {

    // TODO 1: Validate that mode is a valid EditorMode value

    // TODO 2: Store previous mode for potential restoration (search cancel)

    // TODO 3: Update current_mode global variable

    // TODO 4: Update status bar to show current mode indicator

    // TODO 5: Perform any mode-specific initialization (clear search query, etc)

    // Hint: Consider what cleanup is needed when exiting each mode

}

EditorMode get_editor_mode(void) {

    return current_mode;

}

```

## E. Language-Specific Hints:

- Use `termios` structure with `tcgetattr()` and `tcsetattr()` for terminal configuration
- The `read()` system call returns -1 on error; check `errno` for `EAGAIN` or `EINTR`
- Control characters are ASCII values 1-26, corresponding to Ctrl+A through Ctrl+Z
- Escape sequences always start with ASCII 27 ( `\x1b` ) followed by additional characters
- Use `VTIME` and `VMIN` in `termios.c_cc` array to control read timeouts
- Memory for command parameters should be managed carefully; consider if strings need to be copied
- Signal handlers for Ctrl+C should call `disable_raw_mode()` before exiting

## F. Milestone Checkpoints:

**After Milestone 1 (Raw Mode and Input):**

- Run the editor and verify that typed characters don't automatically appear (raw mode working)
- Press arrow keys and verify they are recognized as distinct key codes, not as multiple characters
- Press Ctrl+C and verify the editor exits cleanly with terminal restored
- Press Escape and verify it's detected as a distinct key, not starting a sequence

**After Milestone 4 (Text Editing):**

- Type regular characters and verify they appear in the document at cursor position
- Press backspace and verify characters are deleted from before the cursor
- Press delete key and verify characters are deleted from after the cursor
- Press Enter and verify new lines are created with cursor positioned correctly

**After Milestone 6 (Search):**

- Press Ctrl+F to enter search mode and verify status bar shows search prompt
- Type search characters and verify they appear in the search query, not the document
- Press Escape during search and verify it cancels search and restores cursor position
- Press Enter during search and verify cursor moves to the found match

**G. Debugging Tips:**

| Symptom   | Likely Cause  | Diagnosis   | Fix   |
|---|---|---|---|
| Arrow keys print <code>^[[A</code> instead of moving cursor | Raw mode not enabled or escape sequences not parsed | Check if <code>enable_raw_mode()</code> was called                              | Ensure <code>termios</code> flags disable canonical mode      |
| Editor hangs waiting for input                              | Blocking read with no timeout set                   | Check <code>VTIME</code> and <code>VMIN</code> settings in <code>termios</code> | Set <code>VTIME</code> to small value for timeout-based reads |
| Escape key doesn't work                                     | Timeout too short, escape treated as sequence start | Monitor what characters follow escape in debugger                               | Increase timeout or improve sequence detection                |
| Control keys don't work                                     | ASCII values not mapped correctly                   | Print raw key values to see what's being received                               | Use <code>key &amp; 0x1f</code> to extract control key base   |
| Commands execute but screen doesn't update                  | Missing <code>editor_refresh_screen()</code> calls  | Add debug prints to see if commands execute                                     | Call refresh after each state-changing command                |
| Search mode keys still insert text                          | Mode checking not working in command mapping        | Verify <code>current_mode</code> variable is updated correctly                  | Check mode changes in <code>set_editor_mode()</code> function |

## Text Buffer Component

**Milestone(s):** Milestone 3 (File Viewing), Milestone 4 (Text Editing), Milestone 5 (Save and Undo)

Think of the Text Buffer Component as a sophisticated word processor's document model combined with a librarian's filing system. Just as a word processor maintains your document in memory while you edit it—tracking every character, line break, and modification—our text buffer holds the entire file content in a structured format that enables efficient editing operations. Like a librarian who knows exactly where each book belongs and can quickly reshelf after reorganization, the text buffer maintains precise knowledge of cursor position, line boundaries, and modification state while supporting complex operations like insertion, deletion, and line splitting.

The Text Buffer Component manages the in-memory representation of file content, providing the core data manipulation services that make text editing possible. Unlike simple string concatenation approaches that might work for small files, our buffer uses a line-oriented structure that enables efficient operations on

documents of any size. Each line is stored as an independent unit with both raw character data and a processed "render" representation that handles tab expansion and other display considerations.

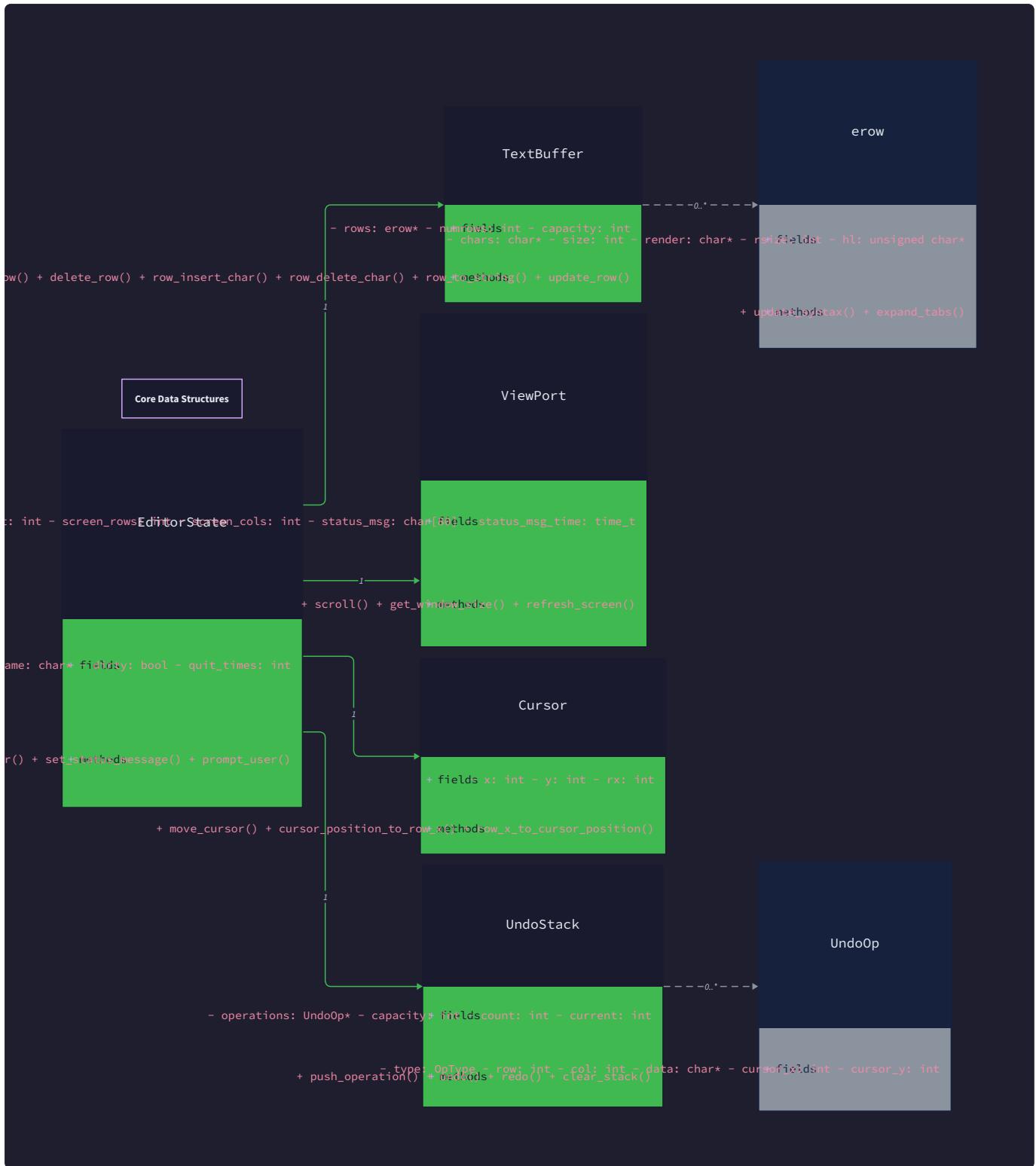
This component sits at the heart of our editor's architecture, receiving editing commands from the Input Handler, providing content to the Screen Renderer, and coordinating with the file system for persistence operations. The buffer maintains both the logical structure of the document (lines and characters) and the metadata necessary for visual presentation (cursor tracking, modification flags, and viewport coordination).

## Core Buffer Operations

The foundation of text editing lies in three fundamental operations: **insertion**, **deletion**, and **cursor movement**. Think of these operations like performing surgery on a living document—each change must maintain the integrity of the overall structure while achieving the desired local modification. The buffer ensures that after every operation, cursor positions remain valid, line boundaries are preserved, and the document remains in a consistent state.

### Character Insertion Logic

Character insertion represents the most frequent operation in text editing, yet it requires careful coordination between multiple data structures. When a user types a character, the buffer must determine the exact insertion point based on current cursor coordinates, expand the target line's storage if necessary, shift existing characters to make room, and update both the raw character array and the rendered representation.



The insertion process begins by locating the target row using the cursor's y-coordinate (`cy`) within the `editor_config` structure. The buffer then converts the logical cursor position (`cx`) to the appropriate insertion index within that row's character array. This conversion accounts for the difference between character positions and byte positions, which becomes critical when supporting Unicode or multi-byte character encodings in future extensions.

| Operation Step             | Action Taken                                   | Data Structures Modified                           |
|----------------------------|--|--|
| 1. Locate Target Row       | Find <code>editor_config.row[cy]</code>        | None (read-only access)                            |
| 2. Convert Cursor Position | Calculate insertion index from <code>cx</code> | Temporary variables only                           |
| 3. Expand Row Storage      | Realloc <code>erow.chars</code> if needed      | <code>erow.chars</code> , <code>erow.size</code>   |
| 4. Shift Characters Right  | Move chars from insertion point                | <code>erow.chars</code> array content              |
| 5. Insert New Character    | Place character at insertion index             | <code>erow.chars</code> array content              |
| 6. Update Row Metadata     | Increment <code>erow.size</code>               | <code>erow.size</code>                             |
| 7. Regenerate Render       | Call <code>editor_update_row()</code>          | <code>erow.render</code> , <code>erow.rsize</code> |
| 8. Advance Cursor          | Increment <code>cx</code> position             | <code>editor_config.cx</code>                      |
| 9. Mark Buffer Dirty       | Set modification flag                          | <code>editor_config.dirty</code>                   |

The most subtle aspect of character insertion involves maintaining consistency between the `chars` array (raw file content) and the `render` array (display representation). Tabs, in particular, require special handling because a single tab character in `chars` expands to multiple spaces in `render`. The `editor_update_row()` function recalculates the entire render representation after any modification to ensure visual accuracy.

## Character Deletion Mechanisms

Character deletion presents more complex boundary conditions than insertion because it must handle cases where the deletion point falls at line boundaries, beginning of file, or end of file. The buffer supports two primary deletion operations: backspace (delete character before cursor) and delete key (delete character at cursor position).

Backspace deletion at the beginning of a line triggers a **line joining operation**, which merges the current line with the previous line and repositions the cursor at the junction point. This operation requires careful memory management because it involves deallocating one row structure while expanding another row's storage to accommodate the merged content.

**Critical Design Insight:** Deletion operations are not simply the inverse of insertion operations. While insertion always succeeds (given sufficient memory), deletion must validate boundary conditions and handle special cases like empty lines, single-character lines, and end-of-document positions.

| Deletion Scenario                | Cursor Position                      | Action Required                                | Special Handling     |
|----------------------------------|--------------------------------------|--|----------------------|
| Mid-line character               | <code>cx &gt; 0, cy unchanged</code> | Shift chars left, decrement<br><code>cx</code> | Standard case        |
| Beginning of line<br>(backspace) | <code>cx = 0, cy &gt; 0</code>       | Join with previous line                        | Memory reallocation  |
| End of line (delete key)         | <code>cx = line_length</code>        | Join with next line                            | Memory reallocation  |
| Single character line            | <code>cx = 0, line_length = 1</code> | Delete entire row                              | Row array compaction |
| Empty line deletion              | <code>line_length = 0</code>         | Delete row structure                           | Row array compaction |
| Beginning of file                | <code>cx = 0, cy = 0</code>          | No operation (boundary)                        | Error handling       |
| End of file                      | Last position                        | No operation (boundary)                        | Error handling       |

The deletion implementation must also handle cursor position updates correctly. For mid-line deletion, the cursor simply moves one position left. For line joining operations, the cursor moves to the previous line at the position where the joined content begins. This requires calculating the previous line's length before the join operation modifies its contents.

### Cursor Position Validation and Correction

Every buffer operation must ensure cursor positions remain valid within the document boundaries. The cursor validation system checks both x-coordinates (column position within line) and y-coordinates (line number within document) against current buffer contents. Invalid positions are corrected to the nearest valid location using well-defined rules.

The validation process becomes particularly important after operations that change document structure. Line deletion may leave the cursor pointing beyond the last line, requiring adjustment to the new final line. Character deletion at end-of-line positions may leave the cursor beyond the line's new length, requiring adjustment to the line's new endpoint.

## Decision: Cursor Correction Strategy

- **Context:** Invalid cursor positions can result from deletion operations or external cursor movement commands
- **Options Considered:**
  1. Strict validation that rejects invalid moves
  2. Automatic correction to nearest valid position
  3. Delayed validation only during rendering
- **Decision:** Automatic correction to nearest valid position
- **Rationale:** Provides the most intuitive user experience by allowing navigation commands to work naturally at document boundaries
- **Consequences:** Requires validation after every buffer modification but eliminates user confusion from rejected navigation attempts

## Line Management

Line management encompasses the complex operations that modify document structure: **line splitting** (Enter key), **line joining** (backspace at line start), and **line deletion**. These operations require careful coordination between the row array management, cursor position updates, and memory allocation strategies.

### Line Splitting Implementation

When a user presses Enter, the editor must split the current line at the cursor position, creating two lines from one. Think of this operation like carefully tearing a piece of paper along a fold—the content before the tear point remains on the original line, while content after the tear point moves to a new line immediately following.

The line splitting process involves several intricate steps that must execute atomically to maintain buffer consistency. First, the buffer creates a new row structure and inserts it into the row array at the position immediately after the current line. Then, it copies characters from the split point to the end of the original line into the new row's character array. Finally, it truncates the original line at the split point and updates both rows' render representations.

| Split Operation Phase    | Memory Operations   | Cursor Updates                              | Validation Required             |
|--------------------------|---|---|---------------------------------|
| Pre-split Validation     | None  | None  | Check cursor within line bounds |
| New Row Allocation       | <code>malloc()</code> for new <code>erow</code>           | None  | Verify allocation success       |
| Row Array Expansion      | <code>realloc()</code> for <code>editor_config.row</code> | None  | Verify expansion success        |
| Content Distribution     | <code>memcpy()</code> split content                       | None  | Ensure null termination         |
| Original Line Truncation | <code>realloc()</code> to shrink <code>chars</code>       | None  | Update size metadata            |
| Row Array Insertion      | <code>memmove()</code> to shift rows                      | None  | Preserve row order              |
| Cursor Repositioning     | None  | Set <code>cy++</code> , <code>cx = 0</code> | Validate new position           |
| Render Updates           | Call <code>editor_update_row() × 2</code>                 | None  | Update both affected rows       |

The most error-prone aspect of line splitting involves managing the row array expansion. The `editor_config.row` array must grow to accommodate the new row, which may require reallocating the entire array and updating all pointers. During this reallocation, the buffer must maintain consistency—if memory allocation fails, the operation must be rolled back completely rather than leaving the buffer in a partially modified state.

## Line Joining Operations

Line joining occurs when the user presses backspace at the beginning of a line (position `cx = 0`) or delete at the end of a line. This operation combines two adjacent lines into a single line, requiring memory management, cursor repositioning, and row array compaction.

The joining process must handle several memory management challenges. The target line (typically the previous line for backspace operations) must expand its character storage to accommodate the additional content from the line being joined. The row being eliminated must have its memory deallocated, and the row array must be compacted to remove the gap left by the deleted row.

## Decision: Line Joining Memory Strategy

- **Context:** Line joining requires expanding one line's storage while deallocating another line completely
- **Options Considered:**
  1. Always expand the earlier line to absorb the later line
  2. Expand the shorter line to absorb the longer line (minimize copying)
  3. Create entirely new line with combined content
- **Decision:** Always expand the earlier line to absorb the later line
- **Rationale:** Provides consistent cursor behavior and simpler mental model for users—cursor always ends up in the earlier line
- **Consequences:** May require more memory copying in cases where earlier line is much longer, but ensures predictable cursor positioning

## Dynamic Row Array Management

The row array (`editor_config.row`) serves as the backbone of document structure, storing pointers to individual line structures in document order. This array must support efficient insertion and deletion operations while maintaining pointer validity for other components that reference specific rows.

Row array operations require careful attention to pointer management and memory layout. When inserting a new row, all existing row pointers after the insertion point must shift to accommodate the new entry. When deleting a row, the gap must be closed by shifting subsequent rows forward. These operations use `memmove()` to handle potentially overlapping memory regions correctly.

| Array Operation           | Memory Movement Required         | Pointer Updates            | Performance Impact         |
|---------------------------|----------------------------------|----------------------------|----------------------------|
| Row Insertion (beginning) | Shift all existing rows          | Update all indices         | $O(n)$ - worst case        |
| Row Insertion (middle)    | Shift rows after insertion point | Update indices after point | $O(n/2)$ - average case    |
| Row Insertion (end)       | No movement required             | No updates needed          | $O(1)$ - best case         |
| Row Deletion (beginning)  | Shift all remaining rows         | Update all indices         | $O(n)$ - worst case        |
| Row Deletion (middle)     | Shift rows after deletion point  | Update indices after point | $O(n/2)$ - average case    |
| Row Deletion (end)        | No movement required             | No updates needed          | $O(1)$ - best case         |
| Array Expansion           | Complete array reallocation      | All pointers invalidated   | $O(n)$ - realloc cost      |
| Array Contraction         | Possible array reallocation      | Potential pointer updates  | $O(n)$ - if realloc occurs |

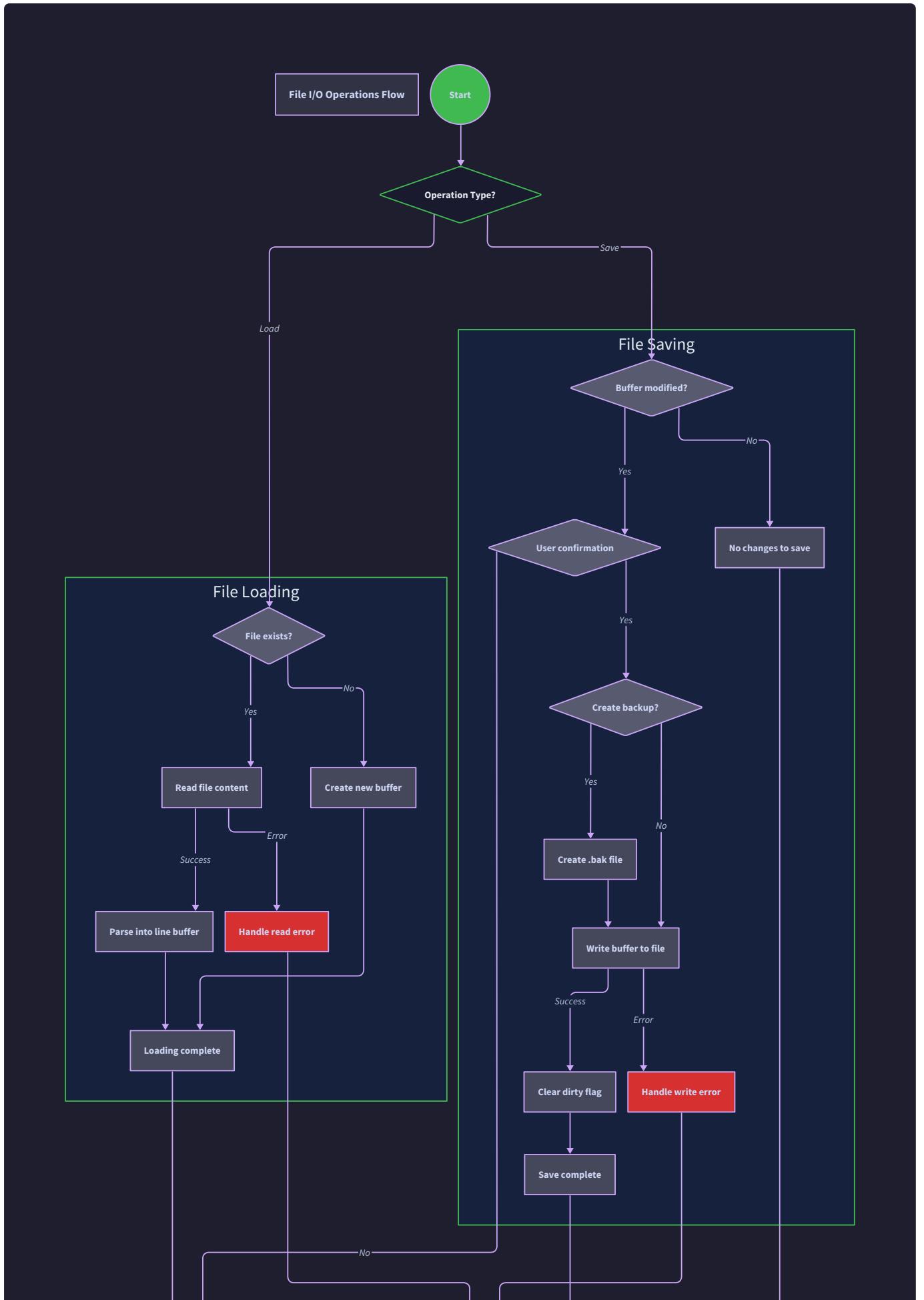
The row array uses a simple dynamic array implementation that doubles in size when expansion is needed and contracts when utilization falls below 25%. This strategy balances memory efficiency against the performance cost of frequent reallocations. The array maintains a capacity field separate from the current count to avoid reallocating on every modification.

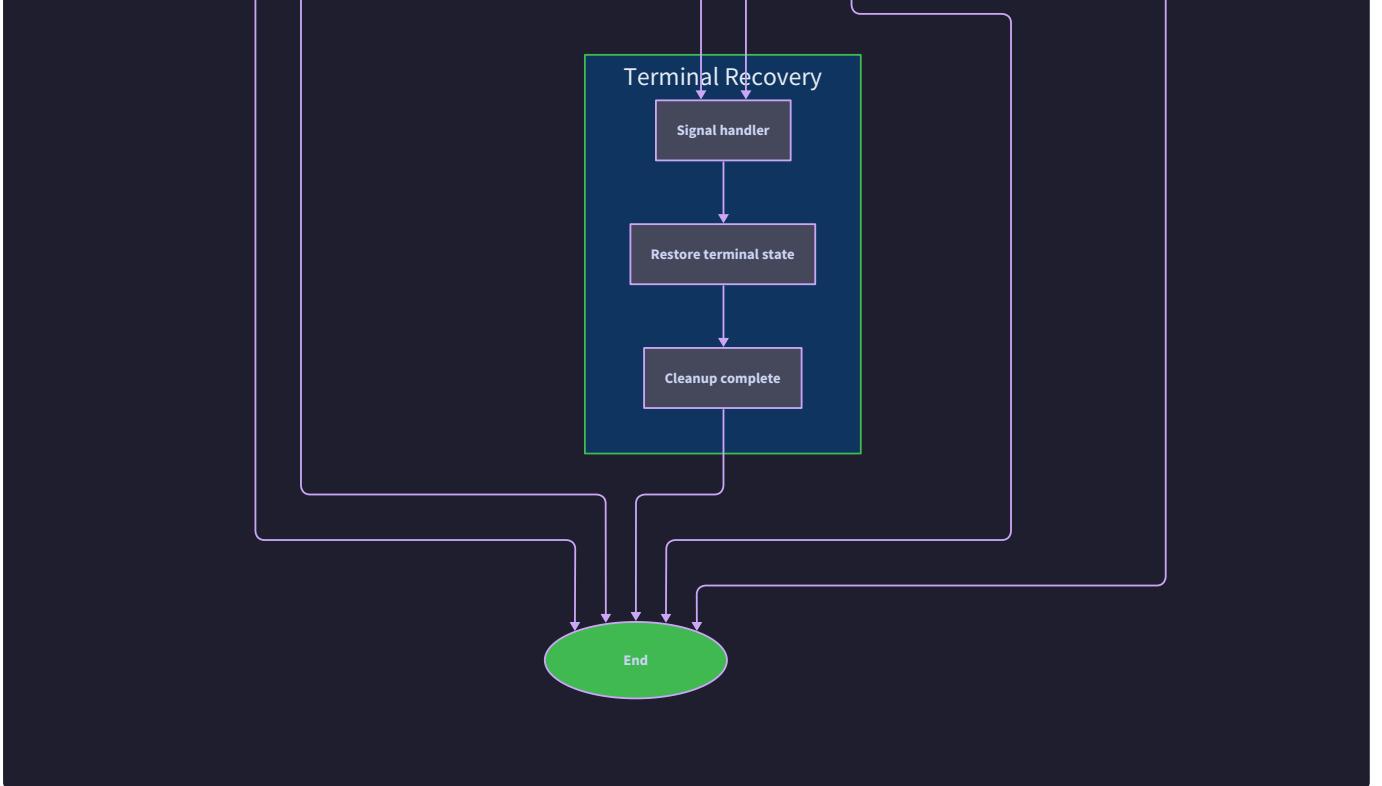
## File I/O Operations

File I/O operations bridge the gap between the in-memory buffer representation and persistent disk storage. These operations must handle character encoding issues, file system errors, permission problems, and ensure data integrity during read and write operations. Think of file I/O as a translation service between the structured, line-oriented buffer format and the flat byte stream format used by the file system.

### File Loading and Buffer Population

Loading a file into the buffer requires reading the entire file content, splitting it into individual lines, and populating the row array with appropriate data structures. This process must handle various line ending conventions (Unix `\n`, Windows `\r\n`, classic Mac `\r`) while building the internal line-oriented representation.





The file loading process begins by opening the specified file and determining its total size to optimize memory allocation strategies. Rather than reading character-by-character, the loader reads the entire file into a temporary buffer and then processes it line-by-line to populate the row array. This approach minimizes system calls while providing flexibility in line-ending detection and processing.

| Loading Phase              | File Operations                                      | Memory Operations                                      | Error Conditions                  |
|----------------------------|--|--|-----------------------------------|
| File Opening               | <code>fopen()</code> with read mode                  | None   | File not found, permission denied |
| Size Determination         | <code>fstat()</code> or <code>fseek()/ftell()</code> | None   | I/O errors, device issues         |
| Content Reading            | <code>fread()</code> entire file                     | <code>malloc()</code> for file buffer                  | Read errors, insufficient memory  |
| Line Boundary Detection    | None   | Scan for newline characters                            | None (algorithmic only)           |
| Row Structure Creation     | None   | <code>malloc()</code> for each row                     | Insufficient memory per row       |
| Character Array Population | None   | <code>malloc()</code> + <code>memcpy()</code> per line | Memory allocation failures        |
| Render Generation          | None   | <code>malloc()</code> for render arrays                | Tab expansion memory needs        |
| Temporary Buffer Cleanup   | None   | <code>free()</code> file buffer                        | None (cleanup only)               |
| File Handle Cleanup        | <code>fclose()</code>                                | None   | Close errors (rare)               |

Line boundary detection handles the three common line ending formats by scanning for newline characters and checking for preceding carriage returns. The detection algorithm normalizes all line endings to the Unix convention (`\n`) internally while preserving the original file's line ending style for writing operations.

The loading process populates each row's `chars` array with the raw line content (excluding line terminators) and immediately calls `editor_update_row()` to generate the corresponding `render` representation. This ensures that the buffer is immediately ready for display operations without requiring additional processing passes.

## Buffer Serialization and File Writing

Saving buffer contents to disk requires serializing the line-oriented internal representation back to a flat byte stream with appropriate line terminators. The serialization process must preserve the original file's line ending convention while ensuring atomic write operations that prevent data loss during system failures.

The file writing implementation uses a temporary file approach to ensure atomic updates. The buffer content is written to a temporary file in the same directory as the target file, and then the temporary file is renamed to replace the original file. This technique ensures that either the write operation completes successfully or the original file remains unchanged—partial writes cannot corrupt the original file.

## Decision: Atomic Write Strategy

- **Context:** File writes can be interrupted by system failures, power loss, or user intervention, potentially corrupting file contents
- **Options Considered:**
  1. Direct write to original file with periodic sync
  2. Write to temporary file then atomic rename
  3. Write to backup file, then write to original, then delete backup
- **Decision:** Write to temporary file then atomic rename
- **Rationale:** Provides true atomicity on POSIX systems where rename is atomic, minimizes risk of data loss
- **Consequences:** Requires temporary disk space equal to file size, may fail on filesystems without atomic rename

| Writing Phase           | File Operations                        | Buffer Processing                      | Error Handling               |
|-------------------------|--|--|------------------------------|
| Temporary File Creation | <code>tmpfile()</code> or custom temp  | None                                   | Disk full, permission issues |
| Line Serialization Loop | None                                   | Iterate through <code>row</code> array | None (memory-only)           |
| Character Data Writing  | <code>fwrite()</code> for each line    | Access <code>erow.chars</code>         | Write errors, disk full      |
| Line Terminator Writing | <code>fwrite()</code> newline chars    | None                                   | Write errors continuation    |
| File Synchronization    | <code>fsync()</code> temp file         | None                                   | Sync failures, I/O errors    |
| Atomic Rename           | <code>rename()</code> temp to target   | None                                   | Cross-device errors          |
| Dirty Flag Update       | None                                   | Clear <code>editor_config.dirty</code> | None (success indicator)     |
| Temporary Cleanup       | <code>unlink()</code> if rename failed | None                                   | Cleanup errors (non-fatal)   |

The serialization loop processes each row in sequence, writing the `chars` array contents followed by the appropriate line terminator. The line terminator choice depends on the original file's format (detected during loading) or defaults to the platform's native format for new files.

## Error Handling and Recovery Strategies

File I/O operations must handle various failure modes gracefully while providing clear feedback to users about the nature of problems encountered. The error handling strategy distinguishes between recoverable errors (temporary permissions, disk space) and permanent errors (device failures, corrupted filesystems) to guide appropriate user responses.

Read operation failures generally fall into categories of missing files, permission problems, or I/O hardware issues. Missing files receive special handling for new file creation workflows, while permission and hardware problems require user intervention. The loading process validates file accessibility before beginning memory allocation to avoid resource leaks during error recovery.

Write operation failures present more complex recovery scenarios because they may leave the system in inconsistent states. The atomic write strategy minimizes these risks, but errors during the rename phase require careful handling to avoid losing both the original file and the updated content. The implementation maintains backup strategies for these edge cases.

| Error Category            | Detection Method                  | Recovery Action             | User Feedback              |
|---------------------------|-----------------------------------|-----------------------------|----------------------------|
| File Not Found (read)     | <code>fopen()</code> returns NULL | Treat as new file creation  | Status message: "New file" |
| Permission Denied (read)  | <code>errno == EACCES</code>      | Abort loading operation     | Error: "Permission denied" |
| Permission Denied (write) | Write operations fail             | Abort save, preserve buffer | Error: "Cannot write file" |
| Disk Full                 | <code>fwrite()</code> short count | Abort save, cleanup temp    | Error: "Disk full"         |
| I/O Hardware Error        | Various system errors             | Abort operation, report     | Error: "I/O error"         |
| Cross-device Rename       | <code>rename()</code> fails EXDEV | Copy + unlink fallback      | Continue save operation    |
| Interrupted System Call   | <code>errno == EINTR</code>       | Retry operation             | Transparent to user        |

## Implementation Guidance

### A. Technology Recommendations

| Component             | Simple Option   | Advanced Option                               |
|-----------------------|---|---|
| File I/O              | <code>fopen</code> , <code>fread</code> , <code>fwrite</code> with error checking | Memory-mapped files with <code>mmap()</code>  |
| Memory Management     | <code>malloc</code> , <code>realloc</code> , <code>free</code> with size tracking | Custom memory pool allocator                  |
| Line Ending Detection | Sequential scan with character comparison   | Boyer-Moore pattern matching                  |
| Text Encoding         | ASCII-only with byte operations   | UTF-8 support with <code>iconv</code> library |
| Atomic Writes         | Temporary file + rename   | Write-ahead logging approach                  |

## B. Recommended File Structure

```

text-editor/
  src/
    editor.h           ← main editor structure definitions
    buffer.c          ← text buffer implementation (this component)
    buffer.h          ← buffer function declarations
    terminal.c        ← terminal manager component
    input.c           ← input handler component
    render.c          ← screen renderer component
    file_io.c         ← file operations (part of buffer component)
    file_io.h         ← file I/O function declarations
  tests/
    test_buffer.c     ← buffer operation unit tests
    test_file_io.c   ← file I/O unit tests
    fixtures/
      unix_endings.txt
      windows_endings.txt
      mixed_endings.txt

```

## C. Infrastructure Starter Code

```
// file_io.h - Complete file I/O infrastructure

#ifndef FILE_IO_H

#define FILE_IO_H


#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <errno.h>

#include <unistd.h>




// Line ending detection and normalization

typedef enum {

    LINE_ENDING_UNIX,      // \n

    LINE_ENDING_WINDOWS,   // \r\n

    LINE_ENDING_MAC        // \r

} LineEndingType;



// File loading result structure

typedef struct {

    char** lines;           // Array of line strings

    int line_count;         // Number of lines loaded

    LineEndingType ending_type; // Detected line ending format

    int success;            // 1 if successful, 0 if error

    char error_message[256]; // Error description if success = 0

} FileLoadResult;



// Utility functions for file operations

FileLoadResult* load_file_lines(const char* filename);
```

C

```
int save_file_lines(const char* filename, char** lines, int line_count, LineEndingType
ending_type);

void free_load_result(FileLoadResult* result);

LineEndingType detect_line_endings(const char* content, size_t length);

char* read_entire_file(FILE* fp, size_t* size);

#endif
```

```
// file_io.c - Complete implementation ready to use
```

C

```
#include "file_io.h"
```

```
char* read_entire_file(FILE* fp, size_t* size) {
```

```
    fseek(fp, 0, SEEK_END);
```

```
    *size = ftell(fp);
```

```
    fseek(fp, 0, SEEK_SET);
```

```
    char* buffer = malloc(*size + 1);
```

```
    if (!buffer) return NULL;
```

```
    size_t bytes_read = fread(buffer, 1, *size, fp);
```

```
    buffer[bytes_read] = '\0';
```

```
    *size = bytes_read;
```

```
    return buffer;
```

```
}
```

```
LineEndingType detect_line_endings(const char* content, size_t length) {
```

```
    for (size_t i = 0; i < length - 1; i++) {
```

```
        if (content[i] == '\r' && content[i + 1] == '\n') {
```

```
            return LINE_ENDING_WINDOWS;
```

```
        } else if (content[i] == '\r') {
```

```
            return LINE_ENDING_MAC;
```

```
        }
```

```
}
```

```
    return LINE_ENDING_UNIX;
```

```
}
```

```
FileLoadResult* load_file_lines(const char* filename) {

    FileLoadResult* result = calloc(1, sizeof(FileLoadResult));

    FILE* fp = fopen(filename, "rb");

    if (!fp) {

        snprintf(result->error_message, sizeof(result->error_message),
                 "Cannot open file: %s", strerror(errno));

        return result;
    }

    size_t file_size;

    char* content = read_entire_file(fp, &file_size);

    fclose(fp);

    if (!content) {

        snprintf(result->error_message, sizeof(result->error_message),
                 "Cannot read file: %s", strerror(errno));

        return result;
    }

    if (file_size == 0) {

        result->lines = malloc(sizeof(char*));

        result->lines[0] = malloc(1);

        result->lines[0][0] = '\0';

        result->line_count = 1;

        result->ending_type = LINE_ENDING_UNIX;
    }
}
```

```
    result->success = 1;

    free(content);

    return result;

}

result->ending_type = detect_line_endings(content, file_size);

// Count lines first

int line_count = 1;

for (size_t i = 0; i < file_size; i++) {

    if (content[i] == '\n' || content[i] == '\r') {

        if (content[i] == '\r' && i + 1 < file_size && content[i + 1] == '\n') {

            i++; // Skip the \n in \r\n

        }

        line_count++;

    }

}

result->lines = malloc(line_count * sizeof(char*));

result->line_count = 0;

size_t line_start = 0;

for (size_t i = 0; i <= file_size; i++) {

    if (i == file_size || content[i] == '\n' || content[i] == '\r') {

        size_t line_length = i - line_start;

        result->lines[result->line_count] = malloc(line_length + 1);

        memcpy(result->lines[result->line_count], &content[line_start], line_length);

        result->line_count++;

    }

}
```

```

        result->lines[result->line_count][line_length] = '\0';

        result->line_count++;

    if (content[i] == '\r' && i + 1 < file_size && content[i + 1] == '\n') {

        i++; // Skip the \n in \r\n

    }

    line_start = i + 1;

}

}

result->success = 1;

free(content);

return result;

}

void free_load_result(FileLoadResult* result) {

if (result) {

    for (int i = 0; i < result->line_count; i++) {

        free(result->lines[i]);

    }

    free(result->lines);

    free(result);

}

}

```

## D. Core Logic Skeleton Code

```
// buffer.h - Core buffer operations for learner implementation

#ifndef BUFFER_H

#define BUFFER_H


#include "file_io.h"

// Row structure representing one line of text

typedef struct erow {

    int size;          // Length of chars array

    char* chars;       // Raw characters (file content)

    int rsize;         // Length of render array

    char* render;      // Rendered characters (tabs expanded)

} erow;

// Main editor configuration and state

struct editor_config {

    int cx, cy;        // Cursor position (logical coordinates)

    int rx;            // Cursor position (visual coordinates)

    int rowoff;        // Vertical scroll offset

    int coloff;        // Horizontal scroll offset

    int screenrows;    // Terminal height

    int screencols;    // Terminal width

    int numrows;        // Number of rows in file

    erow* row;          // Array of file rows

    int dirty;          // Modified since last save

    char* filename;     // Currently open filename

    char statusmsg[80]; // Status bar message

    time_t statusmsg_time; // When status message was set
```

C

```
};

extern struct editor_config E;

// Core buffer operations - implement these

void editor_update_row(erow* row);

void editor_insert_row(int at, char* s, size_t len);

void editor_del_row(int at);

void editor_row_insert_char(erow* row, int at, int c);

void editor_row_del_char(erow* row, int at);

void editor_insert_char(int c);

void editor_delete_char(void);

void editor_insert_newline(void);

// File operations - implement these

void editor_open(char* filename);

char* editor_rows_to_string(int* buflen);

void editor_save(void);

#endif
```

```
// buffer.c - Core buffer implementation skeletons                                C

#include "buffer.h"

#include <stdlib.h>

#include <string.h>

struct editor_config E;

// Updates the render representation of a row after chars changes

void editor_update_row(erow* row) {

    // TODO 1: Count total rendered size accounting for tab expansion

    //           Each tab character expands to TAB_SIZE spaces

    //           Hint: tabs expand to next multiple of TAB_SIZE column position

    // TODO 2: Free existing render array if it exists

    //           Check if row->render is not NULL before freeing

    // TODO 3: Allocate new render array with calculated size

    //           Don't forget space for null terminator

    // TODO 4: Copy chars to render, expanding tabs to spaces

    //           Track both source index (chars) and dest index (render)

    //           Use TAB_SIZE constant for tab expansion width

    // TODO 5: Null-terminate the render array and update rsize

}

void editor_insert_row(int at, char* s, size_t len) {

    // TODO 1: Validate insertion position is within bounds
```

```
//           at should be between 0 and E.numrows (inclusive)

// TODO 2: Expand the row array to accommodate new row

//           Use realloc to grow E.row array

//           Check for allocation failure

// TODO 3: Shift existing rows to make space at insertion point

//           Use memmove to handle overlapping memory regions

//           Shift rows from 'at' to end of array

// TODO 4: Initialize new row structure at insertion point

//           Set size, allocate and copy chars, initialize render to NULL

//           Don't forget to null-terminate the chars array

// TODO 5: Generate render representation for new row

//           Call editor_update_row() on the new row

// TODO 6: Update editor state

//           Increment E.numrows and set E.dirty flag

}

void editor_row_insert_char(erow* row, int at, int c) {

// TODO 1: Validate insertion position within row bounds

//           at should be between 0 and row->size (inclusive)

// TODO 2: Expand row's chars array to accommodate new character

//           Use realloc to grow chars array by one character
```

```
//           Update row->size after successful allocation

// TODO 3: Shift characters right to make space for insertion
//           Use memmove from insertion point to end of chars
//           Handle case where inserting at end (no shift needed)

// TODO 4: Insert the new character at the insertion point
//           Place character at row->chars[at]

// TODO 5: Regenerate render representation
//           Call editor_update_row() to update render array

}

void editor_insert_char(int c) {
    // TODO 1: Handle insertion at end of file (create new row if needed)
    //           If cy == E.numrows, insert empty row at end
    //           Use editor_insert_row() with empty string

    // TODO 2: Insert character at current cursor position
    //           Use editor_row_insert_char() on current row (E.row[E.cy])
    //           Pass E.cx as insertion position

    // TODO 3: Advance cursor to position after inserted character
    //           Increment E.cx to move cursor right

    // TODO 4: Mark buffer as modified
    //           Set E.dirty flag to indicate unsaved changes
```

```
}

void editor_insert_newline(void) {

    // TODO 1: Handle newline at beginning of line (simple case)

    // If E.cx == 0, just insert empty row at current position

    // Advance cursor to beginning of new line


    // TODO 2: Split current line at cursor position

    // Get pointer to current row: E.row[E.cy]

    // Extract text from cursor position to end of line


    // TODO 3: Truncate current line at cursor position

    // Resize current row's chars array to E.cx length

    // Update row size and regenerate render


    // TODO 4: Insert new row with extracted text

    // Use editor_insert_row() at position E.cy + 1

    // Pass extracted text and its length


    // TODO 5: Move cursor to beginning of new line

    // Set E.cx = 0, increment E.cy

}

void editor_delete_char(void) {

    // TODO 1: Handle boundary conditions (beginning/end of file)

    // Return early if cursor at (0,0) - nothing to delete

    // Handle deletion at end of file appropriately}
```

```
// TODO 2: Handle backspace at beginning of line (line join)

//           If E.cx == 0, join current line with previous line

//           Calculate new cursor position before join operation

//           Use string concatenation to merge line contents


// TODO 3: Handle normal character deletion within line

//           Use editor_row_del_char() to remove character before cursor

//           Pass position E.cx - 1 as deletion point


// TODO 4: Update cursor position after deletion

//           Decrement E.cx for normal deletion

//           For line join, set cursor to join point in previous line


// TODO 5: Mark buffer as modified

//           Set E.dirty flag for any successful deletion

}

void editor_open(char* filename) {

    // TODO 1: Store filename in editor configuration

    //           Free existing E.filename if not NULL

    //           Allocate and copy new filename


    // TODO 2: Load file using infrastructure code

    //           Call load_file_lines() to read and parse file

    //           Handle case where file doesn't exist (new file)


    // TODO 3: Populate row array with loaded lines
```

```
//           Iterate through loaded lines

//           Use editor_insert_row() to add each line to buffer


// TODO 4: Clean up loading infrastructure

//           Call free_load_result() to release temporary memory


// TODO 5: Reset editor state for new file

//           Clear dirty flag, reset cursor to (0,0)

//           Set appropriate status message

}

void editor_save(void) {

    // TODO 1: Handle save without filename (prompt user)

    //           If E.filename is NULL, this is "Save As" operation

    //           For now, show error message (filename prompting in later milestone)


    // TODO 2: Convert buffer contents to string representation

    //           Call editor_rows_to_string() to serialize all rows

    //           This creates single string with line terminators


    // TODO 3: Write string to file using infrastructure

    //           Create temporary file for atomic write operation

    //           Write serialized content to temporary file

    //           Use fsync() to ensure data reaches disk


    // TODO 4: Perform atomic replace of original file

    //           Use rename() to atomically replace original with temp file
```

```

// Handle cross-device errors with copy fallback

// TODO 5: Update editor state on successful save

// Clear E.dirty flag to indicate no unsaved changes

// Show success message in status bar

// Handle and report any errors that occur

}

```

## E. Language-Specific Hints

- Use `realloc(ptr, new_size)` for dynamic array expansion; always check return value for NULL
- Call `memmove()` instead of `memcpy()` when source and destination memory might overlap
- Use `snprintf()` instead of `sprintf()` to prevent buffer overflows in error messages
- The `TAB_SIZE` constant should typically be 8; tabs expand to next column divisible by `TAB_SIZE`
- Always null-terminate string arrays; allocate `length + 1` bytes for character arrays
- Use `calloc()` instead of `malloc()` when you need zero-initialized memory
- File operations should check `errno` after failures to provide meaningful error messages

## F. Milestone Checkpoints

### After Milestone 3 (File Viewing):

- Run: `./editor existing_file.txt` - should load and display file contents
- Verify: Arrow keys navigate through file, cursor stays within line bounds
- Test boundary conditions: Navigate to end of longest line, last line of file
- Check scrolling: Open file larger than terminal, verify viewport updates

### After Milestone 4 (Text Editing):

- Test character insertion: Type characters, verify they appear at cursor position
- Test deletion: Use backspace and delete keys, verify correct character removal
- Test line operations: Press Enter to split lines, backspace at line start to join
- Verify buffer state: Check that `E.dirty` flag updates correctly after modifications

### After Milestone 5 (Save and Undo):

- Save test: Modify file and save with `Ctrl+S`, verify changes persist after restart
- Dirty flag test: Status bar should show unsaved indicator after modifications
- File comparison: Use `diff` to compare saved file with expected content

- Error handling: Try saving to read-only directory, verify graceful error messages

## G. Common Pitfalls and Debugging

| Symptom | Likely Cause | How to Diagnose | Fix | ---|---|---| Cursor jumps randomly | Invalid cursor position after edit | Add bounds checking after operations | Validate `E.cx <= row[E.cy].size` || Garbled text display | render array not updated | Check `editor_update_row()` calls | Call after every chars modification || Segmentation fault on typing | chars array not properly resized | Use valgrind or gdb to find access | Check realloc return values || Files save with wrong line endings | Not preserving original format | Compare saved file with original | Store and use detected line ending type || Memory leak warnings | Not freeing allocated row memory | Use valgrind to trace allocations | Free chars and render in `editor_del_row` || Backspace joins wrong lines | Cursor position calculation error | Print cursor values before/after | Calculate join position before modification |

## Screen Renderer Component

**Milestone(s):** Milestone 2 (Screen Refresh), Milestone 3 (File Viewing), Milestone 6 (Search), Milestone 7 (Syntax Highlighting)

Think of the Screen Renderer Component as a sophisticated display driver that transforms your text editor's internal state into the visual representation users see on their terminal. Just like a graphics card converts 3D scene data into pixels on your monitor, the renderer takes the abstract text buffer, cursor position, and editor state and converts them into precise ANSI escape sequences that paint the correct characters, colors, and cursor position on the terminal screen.

The renderer operates on an **immediate mode** architecture, meaning it reconstructs the entire screen contents from scratch on each refresh rather than trying to track incremental changes. This approach simplifies the rendering logic significantly - instead of maintaining complex state about what parts of the screen have changed, we simply rebuild the complete visual representation each time and let terminal optimization handle the actual character updates.

The key insight behind effective terminal rendering is that terminals are fundamentally different from modern GUI frameworks. While GUI systems provide retained mode rendering where you describe what should be visible and the framework manages updates, terminal rendering requires you to explicitly position the cursor and write characters in sequence. This means our renderer must carefully orchestrate cursor movements, text output, and screen clearing to produce a coherent visual result without flicker or corruption.



The renderer's primary responsibility is coordinating three complex operations: calculating which portion of the document should be visible (the viewport), composing that content into a complete screen image including decorative elements like line numbers and status bars, and buffering all output to prevent visual flicker during updates.

## Viewport Calculation

The viewport represents the rectangular window through which users see their document content. Think of it like looking through a telescope - you can only see a small portion of the larger landscape, and moving the telescope (scrolling) changes which part is visible. The viewport calculation determines exactly which lines and columns from the text buffer should appear on the screen based on the current cursor position and terminal dimensions.

The core challenge in viewport management lies in maintaining the invariant that the cursor must always remain visible while providing intuitive scrolling behavior. Users expect the viewport to follow the cursor smoothly without jarring jumps, but they also expect to be able to scroll independently when reviewing code without losing their editing position.

### Decision: Cursor-Follows-Viewport vs Viewport-Follows-Cursor

- **Context:** When the cursor moves beyond the visible screen area, we must decide whether to automatically scroll the viewport to keep the cursor visible, or allow the cursor to move off-screen
- **Options Considered:**
  1. Cursor-follows-viewport: Constrain cursor movement to visible area, scroll explicitly
  2. Viewport-follows-cursor: Automatically scroll viewport when cursor moves off-screen
  3. Hybrid approach: Different behavior for different types of cursor movement
- **Decision:** Viewport-follows-cursor with immediate scrolling
- **Rationale:** Users expect to be able to navigate with arrow keys continuously without hitting invisible boundaries. This matches behavior of vim, emacs, and most editors users are familiar with
- **Consequences:** Simplifies navigation logic and provides intuitive user experience, but requires careful viewport adjustment calculations

| Viewport Calculation Strategy | Pros   | Cons  | Chosen? |
|-------------------------------|--|---|---------|
| Cursor-follows-viewport       | Simple scrolling logic, explicit user control          | Confusing navigation boundaries, breaks user expectations | No      |
| Viewport-follows-cursor       | Intuitive navigation, matches standard editor behavior | Complex scroll calculations, potential for jarring jumps  | Yes     |
| Hybrid approach               | Optimized for different scenarios                      | Complex state management, inconsistent behavior           | No      |

The viewport calculation process involves several coordinate systems that must be carefully managed. The **logical coordinates** represent positions within the document in terms of character offsets and line numbers. The **visual coordinates** account for tab expansion, line wrapping, and other display transformations. The **screen coordinates** represent actual terminal row and column positions.

The viewport state is maintained through several key fields in the `editor_config` structure:

| Field                   | Type             | Description  |
|-------------------------|------------------|--|
| <code>rowoff</code>     | <code>int</code> | First document line visible at top of screen (vertical scroll offset)      |
| <code>coloff</code>     | <code>int</code> | First document column visible at left of screen (horizontal scroll offset) |
| <code>screenrows</code> | <code>int</code> | Number of text rows available for document content (excluding status bar)  |
| <code>screencols</code> | <code>int</code> | Number of columns available for document content (excluding line numbers)  |
| <code>cx</code>         | <code>int</code> | Logical cursor column position within current line                         |
| <code>cy</code>         | <code>int</code> | Logical cursor row position within document                                |
| <code>rx</code>         | <code>int</code> | Visual cursor column position accounting for tab expansion                 |

The vertical scrolling calculation ensures the cursor line remains within the visible screen area. When the cursor moves above the current viewport (`cy < rowoff`), we adjust the viewport to show earlier lines. When the cursor moves below the viewport (`cy >= rowoff + screenrows`), we scroll down to show later lines:

1. Check if cursor row is above viewport: if `cy < rowoff`, set `rowoff = cy` to bring cursor line to top of screen
2. Check if cursor row is below viewport: if `cy >= rowoff + screenrows`, set `rowoff = cy - screenrows + 1` to bring cursor line to bottom of screen
3. Clamp `rowoff` to valid range: ensure `rowoff >= 0` and `rowoff <= max(0, numrows - screenrows)` to prevent showing empty space

Horizontal scrolling follows similar logic but must account for tab expansion and varying line lengths. The visual cursor position `rx` is calculated by expanding tabs from the beginning of the line up to the logical cursor position `cx`:

1. Calculate visual cursor position `rx` using `cx_to_rx()` to expand tabs in current line
2. Check if visual cursor is left of viewport: if `rx < coloff`, set `coloff = rx`
3. Check if visual cursor is right of viewport: if `rx >= coloff + screencols`, set `coloff = rx - screencols + 1`
4. Clamp `coloff` to ensure `coloff >= 0` to prevent negative scroll offsets

The tab expansion calculation walks through characters from the beginning of the line, incrementing the visual position by 1 for regular characters and advancing to the next tab stop for tab characters. Tab stops occur at multiples of `TAB_SIZE`, so a tab character advances the visual position to  $(rx + TAB\_SIZE) - (rx \% TAB\_SIZE)$ .

The critical insight for viewport calculation is that logical and visual coordinates can differ significantly in documents with tabs or Unicode characters. Always perform scrolling decisions based on visual coordinates (`rx`) rather than logical coordinates (`cx`) to ensure the cursor appears where users expect on screen.

Consider a concrete example: a user is editing a line containing "hello\tworld" where `\t` represents a tab character. If the cursor is positioned after the tab (logical position `cx = 6`), the visual position depends on tab expansion. With `TAB_SIZE = 8`, the tab advances from column 5 to column 8, so the visual cursor position `rx = 11`. If the terminal has 80 columns and `coloff = 0`, the cursor is visible. But if `coloff = 20`, the visual cursor at column 11 would be off-screen, requiring horizontal scrolling adjustment.

## Screen Composition

Screen composition transforms the viewport-selected content into a complete visual representation including all decorative elements users expect in a text editor. Think of this like assembling a magazine page - you have the main article text (document content), but you also need page numbers (line numbers), headers and footers (status bar), and proper spacing and alignment to create a cohesive visual layout.

The composition process operates in several distinct phases, each responsible for different aspects of the final screen image. This separation of concerns allows each phase to focus on its specific responsibilities while maintaining clean interfaces between stages.

## Decision: Line-by-Line vs Full-Buffer Composition

- **Context:** We can either compose the screen by processing one line at a time during rendering, or build a complete in-memory representation before outputting
- **Options Considered:**
  1. Line-by-line composition: Generate each line as it's written to terminal
  2. Full-buffer composition: Build complete screen representation in memory first
  3. Hybrid approach: Buffer individual lines but not entire screen
- **Decision:** Full-buffer composition with frame buffer
- **Rationale:** Enables atomic screen updates to prevent flicker, allows post-processing like syntax highlighting across line boundaries, and simplifies debugging by making complete frame visible in memory
- **Consequences:** Uses more memory for frame buffer but provides flicker-free rendering and cleaner architecture

The screen composition pipeline processes content through several stages:

| Stage                 | Input                             | Output                 | Responsibility                            |
|-----------------------|-----------------------------------|------------------------|---|
| Viewport Selection    | Full text buffer + scroll offsets | Visible line subset    | Determine which lines appear on screen    |
| Line Formatting       | Raw line content                  | Formatted display text | Handle tab expansion, truncation, padding |
| Syntax Highlighting   | Formatted text                    | Text with color codes  | Apply language-specific syntax coloring   |
| Line Number Addition  | Highlighted text + line numbers   | Complete line content  | Add left margin with line numbers         |
| Status Bar Generation | Editor state                      | Status bar content     | Create bottom status and message bar      |
| Frame Buffer Assembly | All composed elements             | Complete screen frame  | Combine all elements into final output    |

The viewport selection stage determines which lines from the text buffer should appear on screen based on the current `rowoff` and `screenrows` values. This creates a window into the document content:

1. Calculate first visible line: `start_line = rowoff`
2. Calculate last visible line: `end_line = min(rowoff + screenrows, numrows)`
3. Extract line range from text buffer: `visible_lines = rows[start_line:end_line]`

#### 4. Handle case where document has fewer lines than screen: pad with empty lines

Line formatting transforms raw line content into display-ready text by handling tab expansion, horizontal scrolling, and line truncation. Each line goes through several transformations:

1. Expand tabs to spaces based on `TAB_SIZE` and column position
2. Apply horizontal scrolling by extracting substring starting at `coloff`
3. Truncate or pad line to exactly `screencols` characters for consistent display
4. Handle special characters that need escape sequences or replacement

The tab expansion process must be precise to ensure visual consistency. For each character in the source line:

- Regular characters advance visual position by 1
- Tab characters advance to next tab stop: `next_tab = (visual_pos + TAB_SIZE) - (visual_pos % TAB_SIZE)`
- Non-printable characters are replaced with visible representations (e.g., ^C for control characters)

Syntax highlighting applies language-specific coloring to the formatted text by inserting ANSI color escape sequences around identified tokens. The highlighting engine maintains state across characters to handle multi-character tokens like string literals and comments:

1. Identify current syntax state (normal text, inside string, inside comment, etc.)
2. Scan characters to find token boundaries (keywords, operators, literals)
3. Insert ANSI color codes at token start and reset sequences at token end
4. Handle multi-line constructs like block comments that span line boundaries

Line number generation creates the left margin showing document line numbers with consistent formatting.

The line number display must handle several requirements:

1. Calculate maximum line number width: `num_digits = log10(numrows) + 1`
2. Format each visible line number with leading zeros or spaces for alignment
3. Add separator between line numbers and content (typically " | " or similar)
4. Apply consistent color scheme for line number gutter area

Status bar generation creates the bottom screen area showing editor state information. The status bar typically displays:

| Status Bar Section | Content                                   | Justification |
|--------------------|---|---------------|
| Left side          | Filename, modification indicator (*)      | Left-aligned  |
| Center             | Current mode, search status               | Centered      |
| Right side         | Cursor position (line:column), percentage | Right-aligned |

The status bar composition calculates available space and truncates content intelligently if the terminal is too narrow. Priority is given to essential information like filename and cursor position.

Frame buffer assembly combines all composed elements into a single coherent screen representation. The frame buffer is a contiguous string containing the complete screen content including all ANSI escape sequences for cursor positioning and color control:

1. Start with cursor hide sequence to prevent flicker during update
2. Add cursor positioning sequence to move to screen origin (row 1, column 1)
3. Append each composed line with appropriate line termination
4. Add status bar content at bottom of screen
5. Position cursor at correct location based on editor state
6. End with cursor show sequence to restore cursor visibility

## Flicker Prevention

Flicker prevention is crucial for creating a professional-quality text editor that feels responsive and polished. Terminal flicker occurs when partial screen updates create visible artifacts as old content is erased and new content is drawn character by character. Think of flicker like watching a flipbook animation where some pages are missing - users see jarring transitions instead of smooth visual updates.

The fundamental cause of flicker in terminal applications is the mismatch between how terminals process output and how humans perceive visual changes. Terminals process character output sequentially, updating their display as each character and escape sequence arrives. If we send screen updates as many small write operations, users see intermediate states as the screen is partially cleared and redrawn.

### Decision: Multiple Write Calls vs Single Atomic Write

- **Context:** Terminal output can be sent as many small write() calls for each line and escape sequence, or buffered into a single large write() call
- **Options Considered:**
  1. Multiple small writes: Write each line and escape sequence immediately
  2. Single atomic write: Buffer entire screen content and write in one operation
  3. Line-buffered writes: Buffer each line but write lines individually
- **Decision:** Single atomic write with complete frame buffering
- **Rationale:** Terminals update display after processing each write() system call, so single writes eliminate intermediate visual states that cause flicker
- **Consequences:** Uses more memory for frame buffer but provides completely flicker-free rendering experience

The frame buffering approach eliminates flicker by ensuring terminals receive complete screen updates in single atomic operations. Instead of interleaving cursor positioning, line clearing, and content writing across

multiple system calls, we accumulate the entire screen representation in memory and transmit it as one large buffer.

Frame buffer implementation requires careful memory management and string building:

| Buffer Operation      | Purpose                           | Implementation Strategy                                      |
|-----------------------|-----------------------------------|--|
| Buffer initialization | Allocate space for complete frame | Pre-allocate based on terminal dimensions + escape sequences |
| Content accumulation  | Add text and escape sequences     | Use efficient string building (avoid repeated reallocations) |
| Cursor positioning    | Add movement escape sequences     | Insert ANSI sequences at appropriate buffer positions        |
| Color application     | Add highlighting escape sequences | Interleave color codes with text content                     |
| Atomic transmission   | Send complete buffer to terminal  | Single write() system call for entire frame                  |

The frame buffer building process follows a strict sequence to ensure proper screen composition:

1. **Buffer Initialization:** Allocate frame buffer with sufficient capacity for worst-case screen content including escape sequences
2. **Cursor Hide:** Add escape sequence `\x1b[?25l` to make cursor invisible during update
3. **Screen Clear:** Add escape sequence `\x1b[2J` to clear entire screen (or `\x1b[H` to position at origin without clearing)
4. **Content Assembly:** Append each screen line with embedded color codes and formatting
5. **Cursor Positioning:** Add escape sequence `\x1b[row;colH` to position cursor correctly
6. **Cursor Show:** Add escape sequence `\x1b[?25h` to restore cursor visibility
7. **Atomic Write:** Transmit entire buffer to `STDOUT_FILENO` in single write() call

Buffer capacity estimation must account for both visible text content and overhead from escape sequences. A conservative estimate includes:

- Base text content: `screenrows * screencols` characters
- ANSI escape sequences: approximately 10-20 bytes per line for cursor positioning and color codes
- Status bar content with formatting: additional 100-200 bytes
- Safety margin: 20% extra capacity to handle edge cases

The cursor hide/show sequences are essential for professional-quality rendering. Without cursor hiding, users see the cursor jumping around the screen as each character is drawn, creating a distracting visual effect. The

hide sequence makes the cursor invisible during screen composition, and the show sequence restores it at the correct final position.

The critical insight for flicker prevention is that the granularity of `write()` system calls directly determines the granularity of visual updates. Each `write()` call potentially triggers a screen refresh, so minimizing the number of calls eliminates intermediate visual states.

Consider the difference between flickering and non-flickering approaches. A flickering implementation might perform these operations:

1. `write(STDOUT, "\x1b[2J", 4)` - clear screen (screen goes blank)
2. `write(STDOUT, "\x1b[1;1H", 6)` - position cursor (still blank screen)
3. `write(STDOUT, "first line\n", 11)` - write first line (partial content visible)
4. `write(STDOUT, "second line\n", 12)` - write second line (more content appears)
5. Continue for each line...

Users see the screen clear, then content appearing line by line, creating visible flicker. A flicker-free implementation builds the complete frame first:

1. Accumulate in buffer: `"\x1b[2J\x1b[1;1Hfirst line\nsecond line\n..."`
2. `write(STDOUT, frame_buffer, buffer_length)` - atomic screen update

Users see the old screen content instantly replaced with new content, creating smooth visual transitions.

Terminal buffering can interfere with flicker prevention if output is buffered within the terminal or operating system. The `fflush()` call ensures buffered output is transmitted immediately, and some systems may require additional synchronization.

## Common Pitfalls

### ⚠ Pitfall: Off-by-One Errors in Viewport Boundaries

Many implementations incorrectly calculate viewport boundaries, leading to cursor positioning errors or content that appears one line above or below where expected. This typically happens because terminal coordinates are 1-indexed while array indices are 0-indexed, and viewport calculations mix these coordinate systems.

The specific mistake is using screen coordinates directly as array indices without proper conversion. For example, calculating `visible_line = rows[rowoff + screen_row]` without checking that `rowoff + screen_row < numrows`. This can access memory beyond the text buffer bounds or display the wrong line content.

To avoid this issue, always validate array bounds and use consistent coordinate systems. Convert between 0-indexed buffer positions and 1-indexed screen positions explicitly:

```
screen_row = buffer_row - rowoff + 1 // Convert to 1-indexed screen position  
buffer_index = screen_row + rowoff - 1 // Convert back to 0-indexed buffer
```

### ⚠ Pitfall: Tab Expansion Inconsistencies

Tab expansion calculations often produce inconsistent visual results when the same line is processed multiple times or when horizontal scrolling is involved. This happens because developers calculate tab stops based on the wrong reference point, leading to tabs that expand to different widths depending on context.

The common mistake is calculating tab expansion based on the character position within the visible line rather than the absolute column position within the complete line. For example, if a line contains "hello\tworld" and horizontal scrolling starts at column 3, the tab should still expand based on column 5 (after "hello"), not column 2 (relative to the visible portion).

Maintain tab expansion consistency by always calculating tab stops from the beginning of the complete line, then extracting the visible portion after expansion is complete. Never calculate tabs based on truncated or horizontally scrolled content.

### ⚠ Pitfall: Incomplete Frame Buffer Transmission

Developers sometimes assume that `write()` system calls transmit the complete buffer content, but `write()` can perform partial writes when system buffers are full or signals interrupt the operation. This leads to corrupted screen output where only part of the frame buffer reaches the terminal.

The mistake is calling `write(STDOUT_FILENO, buffer, length)` once and assuming all bytes were transmitted. In reality, `write()` returns the number of bytes actually written, which may be less than requested. Subsequent rendering operations then overwrite the frame buffer before the previous content finishes transmitting.

Handle partial writes by checking the return value and retransmitting remaining content:

```
bytes_written = 0;  
while (bytes_written < buffer_length) {  
    result = write(STDOUT_FILENO, buffer + bytes_written, buffer_length - bytes_written);  
    if (result < 0) handle_error();  
    bytes_written += result;  
}
```

### ⚠ Pitfall: Memory Leaks in Frame Buffer Management

Frame buffer implementations often leak memory by repeatedly allocating new buffers without freeing previous ones, or by miscalculating buffer sizes and causing heap corruption. This is especially problematic during window resize operations when buffer size requirements change.

The typical mistake is allocating frame buffers inside the rendering function without proper cleanup, or using fixed-size buffers that overflow when terminal dimensions increase. Some implementations also fail to handle `realloc()` failures properly, leading to lost buffer pointers.

Implement careful buffer lifecycle management with pre-allocation and reuse strategies. Allocate the frame buffer once during editor initialization with generous sizing, and only reallocate when terminal dimensions change significantly. Always check allocation results and maintain backup pointers during reallocation operations.

## Implementation Guidance

The Screen Renderer Component requires careful coordination between viewport calculations, content formatting, and atomic frame buffer output. The implementation focuses on immediate-mode rendering with complete screen rebuilds on each refresh cycle.

### Technology Recommendations:

| Component         | Simple Option                 | Advanced Option                                    |
|-------------------|-------------------------------|--|
| Frame Buffer      | Dynamic string with realloc() | Pre-allocated circular buffer with size estimation |
| ANSI Sequences    | Hard-coded escape strings     | Terminfo/termcap database lookup                   |
| Tab Expansion     | Simple modulo arithmetic      | Unicode-aware width calculation                    |
| Memory Management | malloc/free per render        | Memory pool with reuse                             |

### File Structure:

```
src/
    renderer.c          ← viewport calculation, screen composition, frame buffering
    renderer.h          ← renderer interface and configuration structures
    ansi.c              ← ANSI escape sequence generation utilities
    ansi.h              ← escape sequence constants and helper functions
    syntax.c            ← syntax highlighting implementation
    syntax.h            ← language definitions and highlighting rules
tests/
    test_renderer.c     ← viewport calculation and composition tests
    test_ansi.c          ← escape sequence generation tests
```

### ANSI Escape Sequence Utilities (Complete Infrastructure):

```
// ansi.h - Complete escape sequence definitions and utilities

#ifndef ANSI_H

#define ANSI_H


// Screen control sequences

#define ANSI_CLEAR_SCREEN "\x1b[2J"

#define ANSI_CLEAR_LINE "\x1b[K"

#define ANSI_CURSOR_HOME "\x1b[H"

#define ANSI_CURSOR_HIDE "\x1b[?25l"

#define ANSI_CURSOR_SHOW "\x1b[?25h"

// Color definitions

#define ANSI_COLOR_RESET "\x1b[m"

#define ANSI_COLOR_RED "\x1b[31m"

#define ANSI_COLOR_GREEN "\x1b[32m"

#define ANSI_COLOR_YELLOW "\x1b[33m"

#define ANSI_COLOR_BLUE "\x1b[34m"

#define ANSI_COLOR_MAGENTA "\x1b[35m"

#define ANSI_COLOR_CYAN "\x1b[36m"

#define ANSI_COLOR_WHITE "\x1b[37m"

// Frame buffer for atomic screen updates

typedef struct {

    char *buffer;

    size_t capacity;

    size_t length;

} FrameBuffer;

// Initialize frame buffer with initial capacity
```

```
FrameBuffer* frame_buffer_create(size_t initial_capacity);

// Append content to frame buffer, reallocating if necessary

int frame_buffer_append(FrameBuffer *fb, const char *content, size_t length);

// Append formatted content to frame buffer

int frame_buffer_appendf(FrameBuffer *fb, const char *format, ...);

// Send complete frame buffer to terminal atomically

int frame_buffer_flush(FrameBuffer *fb, int fd);

// Clear frame buffer for next frame

void frame_buffer_reset(FrameBuffer *fb);

// Free frame buffer resources

void frame_buffer_destroy(FrameBuffer *fb);

// Generate cursor positioning escape sequence

int format_cursor_position(char *buffer, int row, int col);

#endif
```

```
// ansi.c - Complete ANSI utilities implementation

#include "ansi.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdarg.h>

#include <unistd.h>

FrameBuffer* frame_buffer_create(size_t initial_capacity) {

    FrameBuffer *fb = malloc(sizeof(FrameBuffer));

    if (!fb) return NULL;

    fb->buffer = malloc(initial_capacity);

    if (!fb->buffer) {

        free(fb);

        return NULL;

    }

    fb->capacity = initial_capacity;

    fb->length = 0;

    return fb;

}

int frame_buffer_append(FrameBuffer *fb, const char *content, size_t length) {

    if (fb->length + length >= fb->capacity) {

        size_t new_capacity = (fb->capacity * 2) + length;

        char *new_buffer = realloc(fb->buffer, new_capacity);

        if (!new_buffer) return -1;

    }

}
```

C

```
fb->buffer = new_buffer;

fb->capacity = new_capacity;

}

memcpy(fb->buffer + fb->length, content, length);

fb->length += length;

fb->buffer[fb->length] = '\0';

return 0;

}

int frame_buffer_appendf(FrameBuffer *fb, const char *format, ...) {

va_list args;

char temp_buffer[1024];

va_start(args, format);

int length = vsnprintf(temp_buffer, sizeof(temp_buffer), format, args);

va_end(args);

if (length < 0) return -1;

return frame_buffer_append(fb, temp_buffer, length);

}

int frame_buffer_flush(FrameBuffer *fb, int fd) {

size_t bytes_written = 0;

while (bytes_written < fb->length) {

ssize_t result = write(fd, fb->buffer + bytes_written,

fb->length - bytes_written);
```

```

    if (result < 0) return -1;

    bytes_written += result;
}

return 0;
}

void frame_buffer_reset(FrameBuffer *fb) {

    fb->length = 0;

    if (fb->capacity > 0) fb->buffer[0] = '\0';
}

void frame_buffer_destroy(FrameBuffer *fb) {

    if (fb) {

        free(fb->buffer);

        free(fb);
    }
}

int format_cursor_position(char *buffer, int row, int col) {
    return sprintf(buffer, "\x1b[%d;%dH", row, col);
}

```

**Core Renderer Interface (Skeleton with TODOs):**

```
// renderer.h - Screen renderer interface

#ifndef RENDERER_H

#define RENDERER_H


#include "editor.h"

#include "ansi.h"


// Viewport state for tracking visible document region

typedef struct {

    int row_offset;      // First visible document line (0-indexed)

    int col_offset;      // First visible document column (0-indexed)

    int screen_rows;    // Available rows for document content

    int screen_cols;    // Available columns for document content

} Viewport;

// Calculate viewport offsets to keep cursor visible

void editor_scroll(void);

// Convert logical cursor position to visual position with tab expansion

int cx_to_rx(erow *row, int cx);

// Convert visual cursor position back to logical position

int rx_to_cx(erow *row, int rx);

// Refresh complete screen with current editor state

void editor_refresh_screen(void);

// Update render representation for a single row

void editor_update_row(erow *row);
```

C

```
#endif
```

```
// Core viewport calculation (skeleton for learner implementation) C
```

```
void editor_scroll(void) {

    // TODO 1: Calculate visual cursor position (rx) for current row using cx_to_rx()

    // Hint: E.rx = cx_to_rx(&E.row[E.cy], E.cx) where E is global editor_config

    // TODO 2: Handle vertical scrolling - keep cursor row visible

    // If cursor is above viewport (E.cy < E.rowoff), scroll up

    // If cursor is below viewport (E.cy >= E.rowoff + E.screenrows), scroll down

    // TODO 3: Handle horizontal scrolling - keep cursor column visible

    // If visual cursor is left of viewport (E.rx < E.coloff), scroll left

    // If visual cursor is right of viewport (E.rx >= E.coloff + E.screencols), scroll
    right

    // TODO 4: Clamp scroll offsets to valid ranges

    // Ensure rowoff >= 0 and rowoff <= max(0, E.numrows - E.screenrows)

    // Ensure coloff >= 0

}

int cx_to_rx(erow *row, int cx) {

    // TODO 1: Initialize visual position counter (rx = 0)

    // TODO 2: Walk through characters from start of line up to cx position

    // For each character at position j < cx:

    //     - If character is tab ('\t'), advance rx to next tab stop

    //     - Tab stops occur at multiples of TAB_SIZE: rx += (TAB_SIZE - 1) - (rx % TAB_SIZE)

    //     - Otherwise, increment rx by 1 for regular character
```

```
// TODO 3: Return final visual position

// Hint: Handle case where cx > row->size by clamping to row length

}

int rx_to_cx(erow *row, int rx) {

    // TODO 1: Initialize counters for logical (cx) and visual (cur_rx) positions

    // TODO 2: Walk through line calculating visual position until cur_rx >= rx

    // Track both cx (logical position) and cur_rx (visual position) as you scan

    // Stop when visual position reaches or exceeds target rx

    // TODO 3: Return logical position (cx) corresponding to visual position rx

    // Handle case where rx is beyond end of line

}

void editor_refresh_screen(void) {

    // TODO 1: Update viewport to keep cursor visible

    // Call editor_scroll() to adjust rowoff and coloff

    // TODO 2: Create frame buffer for atomic screen update

    // Initialize FrameBuffer with estimated capacity based on screen size

    // TODO 3: Build complete screen content in frame buffer

    // Start with cursor hide sequence and screen clear/position

    // Add each visible line with line numbers and syntax highlighting

    // Add status bar at bottom with filename, cursor position, etc.

    // End with cursor positioning and cursor show sequence
```

```

    // TODO 4: Send complete frame to terminal atomically

    // Use frame_buffer_flush() to transmit entire screen update

    // Reset frame buffer for next render cycle

}

void editor_update_row(erow *row) {

    // TODO 1: Count tabs in source line to calculate render buffer size

    // Each tab expands to up to TAB_SIZE characters, so count tabs and calculate worst
    case

    // TODO 2: Allocate render buffer (row->render) with sufficient capacity

    // Free existing render buffer if it exists


    // TODO 3: Copy characters from source (row->chars) to render buffer

    // Replace tabs with appropriate number of spaces based on column position

    // Copy other characters directly, handling any non-printable characters


    // TODO 4: Update render size (row->rsize) with final length

    // Null-terminate render buffer for safety

}

```

### Milestone Checkpoint - Screen Composition:

After implementing the core renderer functions, verify the following behavior:

- 1. Viewport Tracking:** Move cursor with arrow keys - the viewport should scroll automatically to keep cursor visible. Test with documents longer and wider than the terminal.
- 2. Tab Expansion:** Create a file with tab characters and verify they align to tab stops correctly. Tab stops should be consistent regardless of horizontal scrolling position.
- 3. Flicker-Free Updates:** Screen refreshes should be smooth without visible flicker. If you see content appearing line-by-line or cursor jumping around during updates, the frame buffering isn't working correctly.

**4. Line Number Display:** Line numbers should appear in the left margin with consistent width. The gutter should expand appropriately for files with more than 999 lines.

Test command sequence:

```
# Create test file with tabs and long lines                                BASH
printf "short line\n\tindented line with tab\nThis is a very long line that should trigger
horizontal scrolling when cursor moves to the end\n" > test_render.txt

# Launch editor and test viewport behavior

./editor test_render.txt

# Navigate with arrows, ensure smooth scrolling

# Move to end of long line, verify horizontal scroll

# Check line numbers display correctly
```

### Debugging Tips:

| Symptom                          | Likely Cause  | How to Diagnose                                     | Fix  |
|----------------------------------|---|---|--|
| Screen flickers during updates   | Multiple write() calls instead of atomic frame buffer | Add printf statements before each write() call      | Implement complete frame buffering with single write() |
| Cursor appears in wrong position | Mixing 0-indexed and 1-indexed coordinates            | Print cursor coordinates and screen positions       | Use consistent coordinate system throughout            |
| Tabs display incorrectly         | Tab expansion calculated from wrong reference point   | Print visual vs logical positions during cx_to_rx() | Always expand tabs from beginning of complete line     |
| Content missing at screen edges  | Viewport boundary calculations off by one             | Print rowoff, coloff, and visible content ranges    | Validate array bounds and coordinate conversions       |
| Memory corruption crashes        | Frame buffer overflow or double-free                  | Run with valgrind or address sanitizer              | Implement proper buffer size estimation and cleanup    |

## Undo System Component

**Milestone(s):** Milestone 5 (Save and Undo)

Think of the undo system as a meticulous historian that records every significant action in your text editor. Just as a historian maintains chronological records of events and can tell you what happened at any point in time,

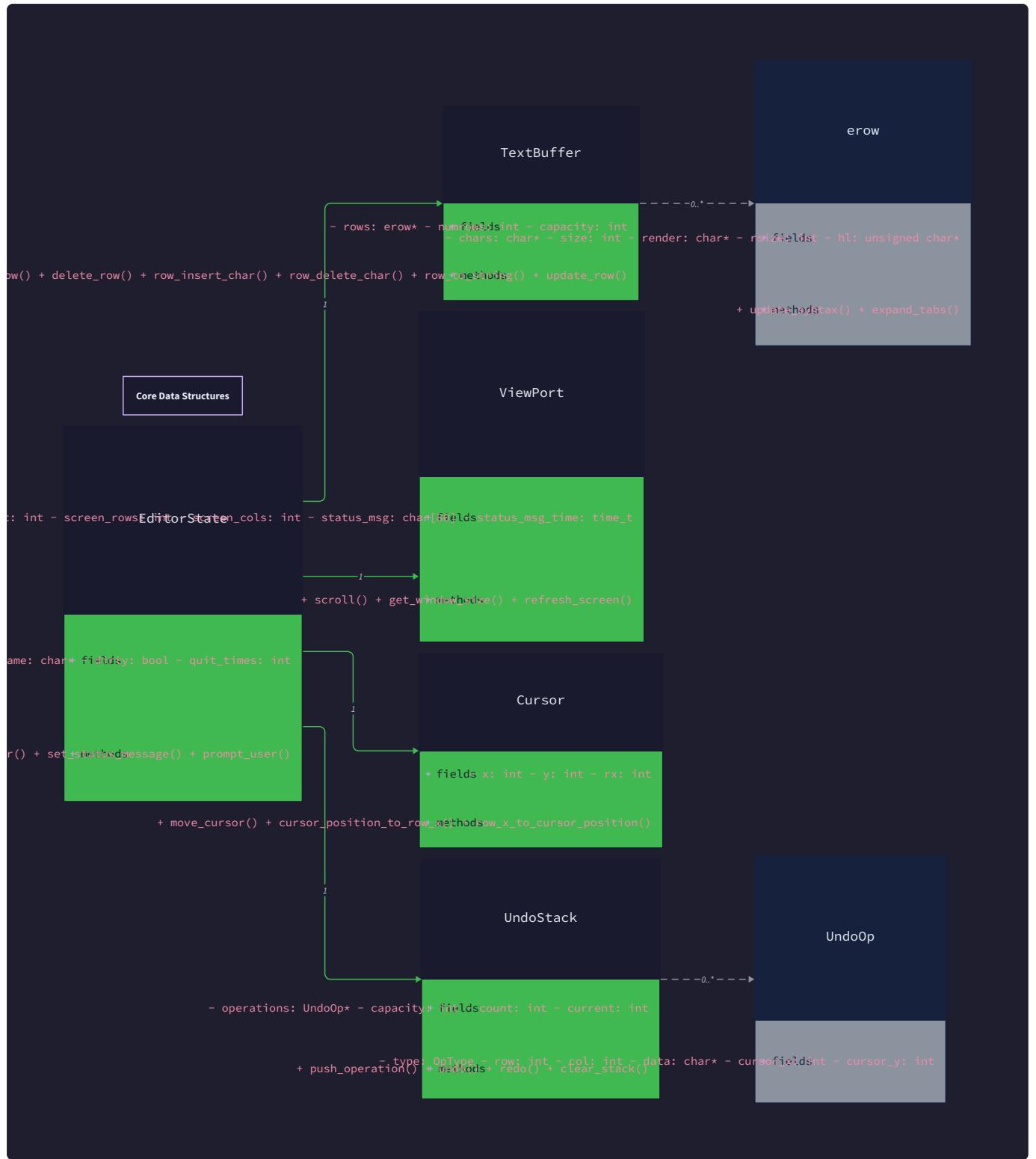
our undo system captures every edit operation in a way that allows us to "travel back in time" and reverse changes. The key insight is that every destructive operation must be paired with enough information to reconstruct the previous state.

The undo system operates on the **command pattern**, where each edit operation is encapsulated as a reversible command object. This is similar to how a bank maintains transaction logs—each transaction contains not just what happened, but enough information to reverse the transaction if needed. When you insert a character, the undo system doesn't just record "character inserted"—it records exactly which character was inserted, at what position, and preserves the information needed to remove that character later.

## Operation Recording

The foundation of any undo system lies in capturing edit operations in a form that can be reliably reversed. Think of this as creating a detailed recipe that can be followed both forward and backward. When a chef follows a recipe, they can also reverse each step if they need to undo their work—adding salt can be reversed by diluting, mixing ingredients can sometimes be reversed by separation. Similarly, every text editing operation must be recorded with sufficient detail to enable reversal.

Our operation recording system captures three categories of edit operations: character insertion, character deletion, and line operations (splitting and joining). Each operation type requires different information to be reversible. Character insertion needs to record the character and position so it can be deleted later. Character deletion needs to record not just the position, but the actual character that was deleted so it can be reinserted. Line operations need to capture the state of affected lines before the operation.



The `Command` structure serves as our operation record, containing all information needed to execute and reverse an edit operation. The command pattern allows us to treat all operations uniformly—whether we're executing a command for the first time or undoing it, we use the same interface but with different parameters.

| Field     | Type          | Description  |
|-----------|---------------|--|
| type      | CommandType   | Identifies the category of operation (insert, delete, newline, etc.) |
| params    | CommandParams | Union containing operation-specific parameters and reversal data     |
| timestamp | time_t        | When the operation was performed, used for command grouping          |

The  `CommandType`  enumeration defines all reversible operations our editor supports:

| Operation          | Description                | Forward Action                | Reverse Action                            |
|--------------------|----------------------------|-------------------------------|---|
| CMD_INSERT_CHAR    | Single character insertion | Insert character at position  | Delete character at position              |
| CMD_DELETE_CHAR    | Single character deletion  | Delete character at position  | Insert saved character at position        |
| CMD_INSERT_NEWLINE | Line split operation       | Split line at cursor position | Join current line with next line          |
| CMD_DELETE_NEWLINE | Line join operation        | Join current line with next   | Split line at saved position              |
| CMD_INSERT_ROW     | Entire row insertion       | Insert row with saved content | Delete row at position                    |
| CMD_DELETE_ROW     | Entire row deletion        | Delete row, saving content    | Insert row with saved content at position |

The  `CommandParams`  union structure contains operation-specific data needed for both execution and reversal:

| Parameter Set | Fields                                      | Purpose  |
|---------------|---|--|
| char_op       | ch (int), row (int), col (int)              | Character operations - stores character and position |
| line_op       | row (int), content (char*), length (size_t) | Line operations - stores row index and content       |
| compound_op   | sub_commands (Command*), count (int)        | Grouped operations that should be undone as a unit   |

**Design Insight:** The critical decision is what constitutes an atomic operation. Individual keystrokes create separate undo points, but some operations like "replace all" should be undoable as a single unit, not as hundreds of individual character replacements.

## Decision: Granularity of Undo Operations

- **Context:** Users expect intuitive undo behavior where a single Ctrl+Z reverses a logical editing action
- **Options Considered:**
  - Fine-grained (every character change is separate undo point)
  - Coarse-grained (group related changes into single undo point)
  - Hybrid (group by time and operation type)
- **Decision:** Hybrid approach with time-based and semantic grouping
- **Rationale:** Provides intuitive behavior while keeping implementation manageable. Consecutive character insertions within a short time window are grouped, but different operation types remain separate
- **Consequences:** Requires timestamp tracking and logic to determine when to group operations vs. create new undo points

The operation recording process follows a precise sequence that ensures every destructive edit is captured before it occurs:

1. **Pre-operation State Capture:** Before modifying the text buffer, the system captures the current state that will be affected by the operation. For character insertion, this means recording the cursor position. For deletion, this means recording both the position and the character being deleted.
2. **Command Object Creation:** A new `Command` structure is allocated and populated with the operation type and captured state information. The timestamp is set to enable time-based grouping with subsequent operations.
3. **Operation Execution:** The actual text buffer modification is performed using the existing buffer manipulation functions (`editor_insert_char`, `editor_delete_char`, etc.).
4. **Post-operation State Recording:** After the operation completes, any additional information needed for reversal is recorded. For line split operations, this includes the final cursor position to enable proper restoration.
5. **Command Storage:** The completed command is added to the undo stack, potentially merging with recent similar commands if time-based grouping applies.

The recording system must handle several edge cases that frequently cause undo system failures:

**⚠ Pitfall: Incomplete State Capture** Many implementations fail to record all information needed for reversal. For example, when deleting a character, recording only the position is insufficient—you must also record the character value to enable reinsertion. Similarly, line operations must preserve not just content but cursor position information for proper state restoration.

**⚠ Pitfall: Recording After Modification** A critical error is modifying the buffer before capturing the pre-operation state. Once a character is deleted from the buffer, that character value is lost forever. The recording must occur before any destructive operation, treating the text buffer as append-only during the recording phase.

## Undo Stack Management

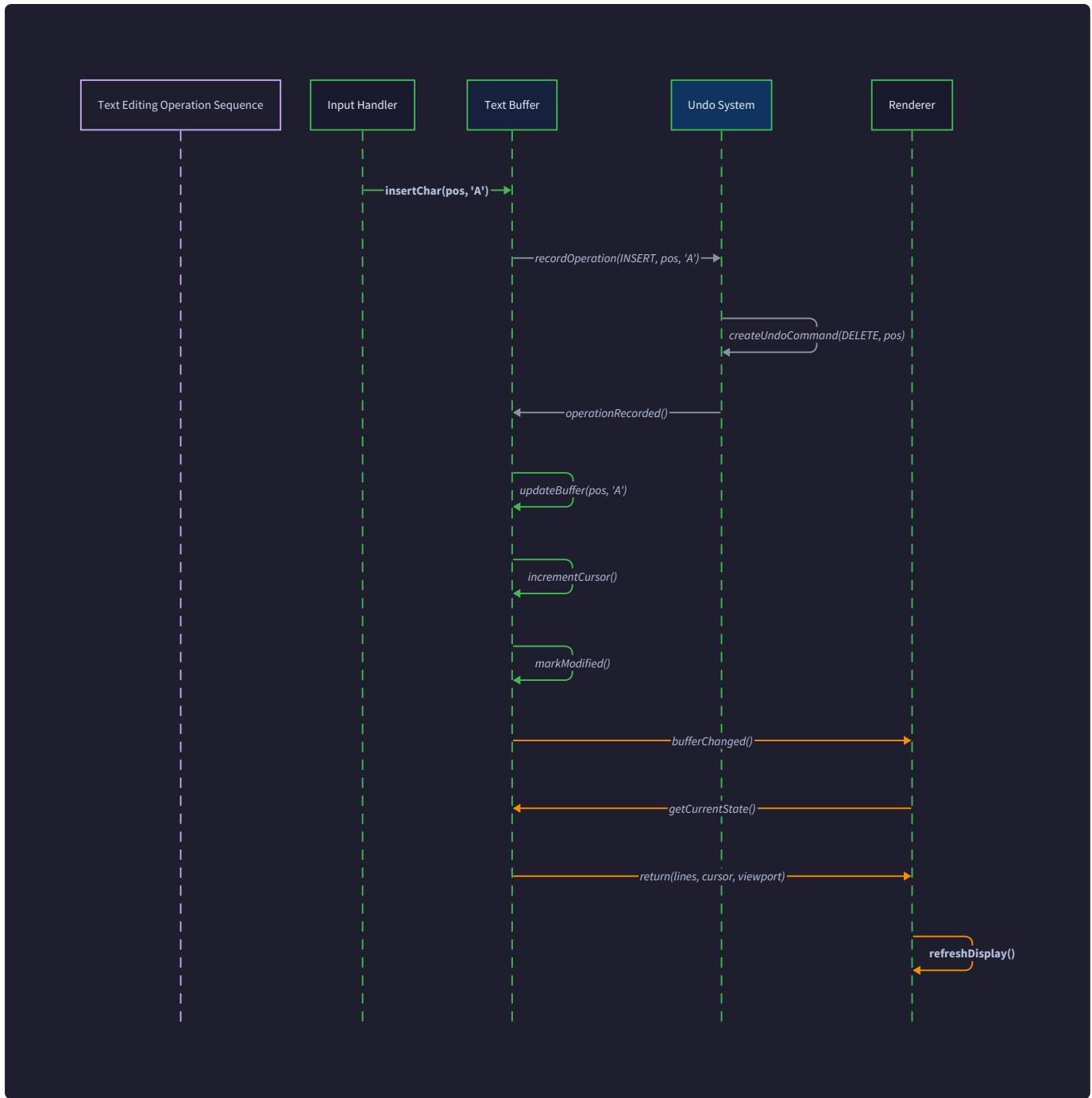
The undo stack management system orchestrates the storage, retrieval, and lifecycle of command objects to enable both undo and redo functionality. Think of this as managing a bookmark system in a book where you can jump back to any previously marked page, but marking a new page while you're in the middle of the book discards all bookmarks after your current position.

The undo stack employs a linear history model with a current position marker. Unlike tree-based undo systems that preserve multiple branches of history, our linear model maintains a single sequence of commands with a pointer indicating the current position in that sequence. This approach provides predictable behavior that matches user expectations from most text editors.

| Component          | Type                  | Description  |
|--------------------|-----------------------|--|
| commands           | <code>Command*</code> | Dynamic array storing all recorded commands                      |
| capacity           | <code>size_t</code>   | Allocated space for command storage                              |
| count              | <code>size_t</code>   | Number of commands currently stored                              |
| current_position   | <code>int</code>      | Index of the most recently executed command (-1 if at beginning) |
| group_threshold_ms | <code>long</code>     | Time window for grouping consecutive similar operations          |

The undo stack maintains several important invariants that ensure correct operation:

- **Position Invariant:** `current_position` is always between -1 and `count - 1`, where -1 represents the state before any commands were executed
- **Execution Invariant:** Commands at indices 0 through `current_position` have been executed and are part of the current buffer state
- **Undo Invariant:** Commands at indices `current_position+1` through `count - 1` have been undone and represent potential redo operations
- **Capacity Invariant:** The commands array always has sufficient capacity to store `count` commands without reallocation during critical operations



### Decision: Linear vs. Tree-Based Undo History

- **Context:** When a user undoes several operations and then makes a new edit, should we preserve the undone operations or discard them?
- **Options Considered:**
  - Linear history: New edits discard undone operations
  - Tree-based history: Preserve all branches of edit history
  - Hybrid: Linear with limited branch preservation
- **Decision:** Linear history with truncation on new edits
- **Rationale:** Simpler to implement and understand, matches behavior of most text editors, avoids UI complexity of branch navigation

- **Consequences:** Some edit history is permanently lost, but behavior is predictable and implementation is straightforward

The undo operation retrieves the command at the current position and executes its reverse action:

1. **Position Validation:** Verify that `current_position >= 0` to ensure there are operations to undo. If the position is -1, we're already at the initial state.
2. **Command Retrieval:** Fetch the command at index `current_position` from the commands array. This command represents the most recent operation in the current buffer state.
3. **Reverse Execution:** Execute the reverse action specified by the command type. For `CMD_INSERT_CHAR`, this means deleting the character at the recorded position. For `CMD_DELETE_CHAR`, this means inserting the saved character.
4. **Position Update:** Decrement `current_position` to reflect that this command is no longer part of the current buffer state.
5. **Buffer Consistency:** Verify that the buffer state matches expectations after the reverse operation. This includes updating cursor position and dirty flags appropriately.

The redo operation performs the inverse process, re-executing a previously undone command:

1. **Availability Check:** Verify that `current_position < count - 1` to ensure there are undone operations available for redo.
2. **Command Selection:** Retrieve the command at index `current_position + 1`. This command was previously undone and is now being re-executed.
3. **Forward Execution:** Execute the original action specified by the command. This uses the same execution path as the initial operation to ensure consistency.
4. **Position Advancement:** Increment `current_position` to reflect that this command is now part of the current buffer state.
5. **State Restoration:** Update all associated state (cursor position, dirty flags, viewport) to match the post-operation state.

The stack management system includes several critical mechanisms for maintaining performance and memory usage:

**Command Grouping:** Consecutive operations of the same type within a short time window are grouped into compound commands. When the user types "hello", this creates a single undo point rather than five separate character insertions. The grouping logic examines the timestamp and operation type of new commands, merging them with the most recent command if they qualify for grouping.

| Grouping Rule            | Condition  | Action                             |
|--------------------------|--|------------------------------------|
| Character insertion      | Same row, consecutive columns, within time threshold         | Merge into compound insert command |
| Character deletion       | Same row, consecutive positions, within time threshold       | Merge into compound delete command |
| Different operation type | New command type differs from previous                       | Create new undo point              |
| Time threshold exceeded  | More than <code>group_threshold_ms</code> since last command | Create new undo point              |
| Position discontinuity   | Cursor moved to non-adjacent position                        | Create new undo point              |

**Memory Management:** The undo stack implements a circular buffer with configurable history limits to prevent unbounded memory growth. When the stack reaches its capacity limit, the oldest commands are discarded to make room for new operations. This requires careful handling of string data associated with line operations—when commands are discarded, their associated memory must be properly freed.

**History Truncation:** When a new edit operation occurs while `current_position` is not at the end of the command array (meaning some operations have been undone), all commands after the current position are discarded. This implements the linear history model where new edits create a new timeline that supersedes the undone operations.

**⚠ Pitfall: Memory Leaks in Command Storage** Commands that contain dynamically allocated data (like saved line content) must have their memory properly managed. When commands are discarded due to history limits or truncation, the associated string data must be freed. Failing to do this creates memory leaks that grow with editor usage.

**⚠ Pitfall: Cursor Position Consistency** After undo/redo operations, the cursor position must be correctly restored to match the buffer state. Many implementations forget to update the cursor position during undo, leaving it pointing to invalid locations or displaying inconsistent state to the user.

## Common Pitfalls

**⚠ Pitfall: Incomplete Command Reversal** The most frequent error in undo system implementation is failing to fully reverse the effects of a command. For example, when undoing a line split operation, implementations often join the lines correctly but forget to restore the original cursor position within the joined line. Every aspect of editor state that was modified by the original operation must be restored during undo.

**⚠ Pitfall: Inconsistent Operation Boundaries** Different parts of the editor code may record operations at different granularities, leading to confusing undo behavior. If the character insertion function records individual characters but the paste function records the entire paste as one operation, users will experience inconsistent

undo granularity. Establish clear rules about what constitutes an atomic operation and enforce them consistently.

**⚠ Pitfall: Stack Overflow on Large Operations** When implementing compound operations (like replace-all that affects thousands of lines), storing each individual change as a separate command can exhaust memory or create performance problems. Large operations should be recorded as single reversible units rather than collections of tiny operations.

**⚠ Pitfall: Race Conditions with Dirty Flag** The dirty flag that indicates unsaved changes must be correctly maintained during undo/redo operations. If an undo operation returns the buffer to its saved state, the dirty flag should be cleared. If a redo operation moves away from the saved state, the dirty flag should be set. Many implementations fail to track the "save point" position in the undo stack.

## Implementation Guidance

The undo system requires careful coordination between command recording, stack management, and state restoration. The implementation should prioritize correctness over performance optimization, as undo bugs are particularly frustrating for users.

## Technology Recommendations

| Component         | Simple Option                  | Advanced Option                        |
|-------------------|--------------------------------|--|
| Command Storage   | Fixed-size circular array      | Dynamic array with reallocation        |
| Memory Management | malloc/free for command data   | Memory pools for reduced fragmentation |
| Time Tracking     | Simple timestamp comparison    | Sophisticated grouping heuristics      |
| Debugging         | Printf debugging of operations | Comprehensive command logging system   |

## Recommended File Structure

```
editor/
src/
    undo.h          ← Undo system interface
    undo.c          ← Command recording and stack management
    command.h       ← Command structure definitions
    command.c       ← Command execution and reversal
    editor.c        ← Main editor integration
tests/
    test_undo.c     ← Undo system unit tests
    test_integration.c ← End-to-end undo behavior tests
```

## Infrastructure Starter Code

Here's the complete command structure definition and basic stack infrastructure:

```
// command.h - Command structure definitions
```

C

```
#ifndef COMMAND_H
```

```
#define COMMAND_H
```

```
#include <time.h>
```

```
#include <stddef.h>
```

```
typedef enum {
```

```
    CMD_INSERT_CHAR,
```

```
    CMD_DELETE_CHAR,
```

```
    CMD_INSERT_NEWLINE,
```

```
    CMD_DELETE_NEWLINE,
```

```
    CMD_INSERT_ROW,
```

```
    CMD_DELETE_ROW,
```

```
    CMD_COMPOUND
```

```
} CommandType;
```

```
typedef struct {
```

```
    int ch;
```

```
    int row;
```

```
    int col;
```

```
} CharOperation;
```

```
typedef struct {
```

```
    int row;
```

```
    char* content;
```

```
    size_t length;
```

```
    int cursor_col; // For proper cursor restoration
```

```
} LineOperation;
```

```
typedef struct Command Command;

typedef struct {

    Command* sub_commands;

    int count;

    int capacity;

} CompoundOperation;

typedef union {

    CharOperation char_op;

    LineOperation line_op;

    CompoundOperation compound_op;

} CommandParams;

struct Command {

    CommandType type;

    CommandParams params;

    time_t timestamp;

};

typedef struct {

    Command* commands;

    size_t capacity;

    size_t count;

    int current_position;

    long group_threshold_ms;

    int save_point_position; // Position when file was last saved

} UndoStack;
```

```
// Initialize undo stack with specified capacity

UndoStack* undo_stack_create(size_t initial_capacity);

// Free undo stack and all associated memory

void undo_stack_free(UndoStack* stack);

// Mark current position as save point for dirty flag management

void undo_stack_mark_save_point(UndoStack* stack);

// Check if current position matches save point (buffer is clean)

int undo_stack_is_clean(UndoStack* stack);

#endif
```

```
// undo.c - Basic stack management infrastructure
```

C

```
#include "undo.h"
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
#include <sys/time.h>
```

```
#define DEFAULT_GROUP_THRESHOLD_MS 1000
```

```
UndoStack* undo_stack_create(size_t initial_capacity) {
```

```
    UndoStack* stack = malloc(sizeof(UndoStack));
```

```
    if (!stack) return NULL;
```

```
    stack->commands = malloc(sizeof(Command) * initial_capacity);
```

```
    if (!stack->commands) {
```

```
        free(stack);
```

```
        return NULL;
```

```
}
```

```
    stack->capacity = initial_capacity;
```

```
    stack->count = 0;
```

```
    stack->current_position = -1;
```

```
    stack->group_threshold_ms = DEFAULT_GROUP_THRESHOLD_MS;
```

```
    stack->save_point_position = -1;
```

```
    return stack;
```

```
}
```

```
void undo_stack_free(UndoStack* stack) {
```

```
if (!stack) return;

// Free all command-associated memory

for (size_t i = 0; i < stack->count; i++) {

    if (stack->commands[i].type == CMD_INSERT_ROW ||

        stack->commands[i].type == CMD_DELETE_ROW) {

        free(stack->commands[i].params.line_op.content);

    } else if (stack->commands[i].type == CMD_COMPOUND) {

        // Recursively free compound commands

        for (int j = 0; j < stack->commands[i].params.compound_op.count; j++) {

            // Free individual sub-commands if needed

        }

        free(stack->commands[i].params.compound_op.sub_commands);

    }

}

free(stack->commands);

free(stack);

}

static long get_current_time_ms() {

    struct timeval tv;

    gettimeofday(&tv, NULL);

    return tv.tv_sec * 1000 + tv.tv_usec / 1000;

}

void undo_stack_mark_save_point(UndoStack* stack) {

    stack->save_point_position = stack->current_position;
```

```
}

int undo_stack_is_clean(UndoStack* stack) {

    return stack->current_position == stack->save_point_position;

}
```

## Core Logic Skeleton Code

The learner should implement these core functions that handle command recording and execution:

```
// Record a character insertion operation

// This should be called BEFORE the character is actually inserted

void undo_record_char_insert(UndoStack* stack, int row, int col, int ch) {

    // TODO 1: Check if this operation can be grouped with the previous command
    //           (same type, consecutive position, within time threshold)

    // TODO 2: If grouping is possible, extend the existing compound command

    // TODO 3: If not grouping, create new Command with CMD_INSERT_CHAR type

    // TODO 4: Fill in params.char_op with row, col, and character

    // TODO 5: Set timestamp to current time

    // TODO 6: Add command to stack, handling capacity expansion if needed

    // TODO 7: Truncate any commands after current_position (linear history)

    // TODO 8: Update current_position to point to new command

}

// Record a character deletion operation

// This should be called BEFORE the character is actually deleted

void undo_record_char_delete(UndoStack* stack, int row, int col, int ch) {

    // TODO 1: Similar grouping logic as insert, but for consecutive deletions

    // TODO 2: Create Command with CMD_DELETE_CHAR type

    // TODO 3: CRITICAL: Save the character being deleted in params.char_op.ch

    // TODO 4: Save position information for proper restoration

    // TODO 5: Handle stack management (truncation, capacity, position update)

    // Hint: The character value must be captured before buffer modification

}

// Execute undo operation - reverse the most recent command

int undo_execute(UndoStack* stack) {

    // TODO 1: Validate that current_position >= 0 (operations available to undo)
```

```

// TODO 2: Get command at current_position

// TODO 3: Switch on command type and execute reverse operation:

//     - CMD_INSERT_CHAR: call editor_delete_char at recorded position

//     - CMD_DELETE_CHAR: call editor_insert_char with saved character

//     - CMD_INSERT_NEWLINE: join lines appropriately

//     - CMD_DELETE_NEWLINE: split line at recorded position

//     - CMD_COMPOUND: iterate through sub-commands in reverse order

// TODO 4: Update cursor position to match pre-operation state

// TODO 5: Decrement current_position

// TODO 6: Return 1 for success, 0 if no operations to undo

// Hint: Cursor position restoration is critical for user experience

}

// Execute redo operation - re-execute previously undone command

int redo_execute(UndoStack* stack) {

    // TODO 1: Validate that current_position < count-1 (operations available to redo)

    // TODO 2: Get command at current_position + 1

    // TODO 3: Switch on command type and execute forward operation:

        - Use same execution path as original operations for consistency

    // TODO 4: Update cursor position to match post-operation state

    // TODO 5: Increment current_position

    // TODO 6: Return 1 for success, 0 if no operations to redo

}

// Expand stack capacity when full

static int undo_stack_expand(UndoStack* stack) {

    // TODO 1: Calculate new capacity (typically double current capacity)

    // TODO 2: Use realloc to expand commands array

```

```
// TODO 3: Handle realloc failure gracefully  
  
// TODO 4: Update stack->capacity  
  
// TODO 5: Return 1 for success, 0 for failure  
  
// Hint: Always check realloc return value before updating pointers  
  
}
```

## Integration with Editor Components

The undo system must be integrated into existing editor operations:

```
// Modified editor_insert_char to support undo
```

C

```
void editor_insert_char(int c) {  
  
    // Record undo information BEFORE modifying buffer  
  
    undo_record_char_insert(&E.undo_stack, E.cy, E.cx, c);  
  
  
    // Existing insertion logic  
  
    if (E.cy == E.numrows) {  
  
        editor_insert_row(E.numrows, "", 0);  
  
    }  
  
    editor_row_insert_char(&E.row[E.cy], E.cx, c);  
  
    E.cx++;  
  
    E.dirty = 1;  
  
}
```

```
// Modified editor_delete_char to support undo
```

```
void editor_delete_char(void) {  
  
    if (E.cy == E.numrows) return;  
  
    if (E.cx == 0 && E.cy == 0) return;  
  
  
    erow *row = &E.row[E.cy];  
  
    if (E.cx > 0) {  
  
        // Record the character being deleted BEFORE deletion  
  
        int deleted_char = row->chars[E.cx - 1];  
  
        undo_record_char_delete(&E.undo_stack, E.cy, E.cx - 1, deleted_char);  
  
  
        editor_row_del_char(row, E.cx - 1);  
  
        E.cx--;  
  
    } else {
```

```

        // Handle line joining - more complex undo recording needed

        // TODO: Implement line join undo recording

    }

E.dirty = 1;

}

```

## Language-Specific Hints

For C implementation:

- Use `gettimeofday()` for millisecond precision timestamps needed for grouping
- Be extremely careful with memory management - every `malloc()` needs corresponding `free()`
- Use `realloc()` for dynamic array expansion, but always check return value
- Consider using `memmove()` for efficient array element shifting during truncation
- Use `memcpy()` for copying command structures, but be aware of shallow vs deep copy issues

## Milestone Checkpoint

After implementing the undo system, verify these behaviors:

### Basic Undo/Redo:

- Type several characters, press Ctrl+Z - characters should disappear in reverse order
- Press Ctrl+Y or Ctrl+Shift+Z - characters should reappear in original order
- Status bar should show correct dirty state throughout

### Command Grouping:

- Type "hello" quickly - should undo as single operation, not 5 separateundos
- Type "he", pause 2 seconds, type "llo" - should create 2 undo points
- Mix insertions and deletions - should create separate undo points

### Edge Cases:

- Undo at beginning of file should do nothing
- Redo when no undone operations should do nothing
- Make edit after undo - should discard redo history
- Save file, make edit, undo - dirty flag should clear when returning to save point

### Memory Management:

- Run with valgrind or similar - should show no memory leaks
- Perform many operations to trigger stack expansion - should work correctly

- Create very large undo history - should eventually discard old operations

## Debugging Tips

| Symptom                             | Likely Cause                       | How to Diagnose                                      | Fix   |
|-------------------------------------|------------------------------------|--|---|
| Undo corrupts text                  | Incomplete state capture           | Add logging to record operations vs undo actions     | Ensure all operation parameters are captured before buffer modification |
| Memory usage grows unbounded        | Not freeing command memory         | Monitor memory usage during long editing sessions    | Implement proper command memory cleanup in stack truncation             |
| Cursor in wrong position after undo | Not restoring cursor state         | Compare cursor position before/after undo operations | Save and restore cursor coordinates as part of command data             |
| Inconsistent undo granularity       | Different recording points in code | Trace where undo_record_* functions are called       | Standardize recording at consistent abstraction level                   |
| Crash on undo/redo                  | Buffer overflow or null pointer    | Use debugger to trace exact crash location           | Add bounds checking and null pointer validation                         |

## Search System Component

### Milestone(s): Milestone 6 (Search)

Think of the search system as a digital detective that works in real-time. Just as a detective updates their theories as new evidence arrives, incremental search continuously refines its findings as you type each character. Unlike traditional search interfaces that require you to type the complete query and press Enter, incremental search provides immediate feedback, highlighting matches and allowing navigation as soon as you start typing. This creates an interactive discovery process where the search results guide your query refinement.

The search system operates as a modal component that temporarily takes over input processing while maintaining the editor's visual context. When activated, it enters a specialized input mode where normal editing keys are reinterpreted as search commands, while the search query appears in the status bar and matches are highlighted throughout the visible text. This modal approach allows the search system to provide rich interaction without conflicting with normal editing operations.

The architecture follows a state-driven design where the search system maintains its own state machine, coordinating with the Screen Renderer Component to provide visual feedback and with the Input Handler

Component to process search-specific keystrokes. The search operates on the current text buffer without modifying it, instead maintaining separate tracking of match positions and current search state.

## Incremental Search Logic

The incremental search logic forms the computational heart of the search system, responsible for finding and tracking text matches as the user builds their query character by character. Think of it as a pattern matching engine that operates in real-time, constantly scanning the text buffer to find occurrences of the current search string and maintaining a live index of match locations.

The search process begins when the user activates search mode, typically by pressing a designated search key like Ctrl+F. At this point, the editor transitions from normal editing mode to search mode, where subsequent keystrokes are interpreted as search query characters rather than text to be inserted into the buffer. The search system initializes its internal state, clearing any previous search results and preparing to process the incoming query.

As each character is typed, the search system performs a fresh scan of the entire text buffer, looking for all occurrences of the current search string. This approach, while computationally intensive for very large files, provides consistent and predictable behavior that matches user expectations. The scan operates on the raw character content of each line, respecting case sensitivity settings and handling special characters appropriately.

### Decision: Real-time vs Buffered Search

- **Context:** Search could update after every keystroke or buffer input and update on pauses
- **Options Considered:** Immediate update on every character, debounced updates after typing pauses, manual trigger with Enter key
- **Decision:** Immediate update on every character
- **Rationale:** Provides instant visual feedback that helps users refine queries and locate matches quickly. The performance cost is acceptable for typical file sizes, and delayed updates create confusing user experience.
- **Consequences:** Requires efficient search algorithm and may need optimization for very large files. Creates more responsive user experience at cost of CPU usage.

| Search Update Strategy  | Pros                                   | Cons   | Chosen |
|-------------------------|--|--|--------|
| Immediate per-character | Instant feedback, intuitive UX         | Higher CPU usage, potential lag on large files | ✓      |
| Debounced (300ms delay) | Reduced CPU usage, handles large files | Confusing delays, interrupts user flow         | ✗      |
| Manual trigger (Enter)  | Minimal CPU usage, traditional         | Poor discoverability, slower workflow          | ✗      |

The search algorithm employs a straightforward string matching approach using the standard library's string search functionality. For each line in the text buffer, the system searches for occurrences of the query string, recording the line number and column position of each match. This information is stored in a dynamic array of match positions that gets rebuilt on every search update.

Here's the detailed search process flow:

1. The search system receives a new character from the input handler and appends it to the current search query string
2. If the query becomes empty (due to backspace), all match highlighting is cleared and the search state is reset
3. For non-empty queries, the system iterates through every line in the text buffer using the editor's `numrows` field
4. For each line, it searches for all occurrences of the query string within that line's character content
5. Each found match is recorded as a match position containing the line number, starting column, and match length
6. The system determines the "current match" by finding the match closest to the cursor position, preferring matches after the cursor
7. The search state is updated with the new match list and current match index
8. A screen refresh is triggered to update the visual highlighting

The search system maintains several pieces of state to track the current search session:

| Field Name     | Type            | Description  |
|----------------|-----------------|--|
| query          | char[256]       | Current search query string being built by user input                |
| query_len      | int             | Length of current search query for efficient string operations       |
| matches        | SearchMatch*    | Dynamic array of all match positions found in current search         |
| match_count    | int             | Total number of matches found for current query                      |
| current_match  | int             | Index of currently selected match for navigation purposes            |
| direction      | SearchDirection | Forward or backward search direction for match navigation            |
| case_sensitive | int             | Boolean flag controlling case sensitivity in pattern matching        |
| wrap_around    | int             | Boolean flag controlling whether search wraps at document boundaries |

Each individual match is represented by a structured record that captures its location and extent:

| Field Name | Type | Description  |
|------------|------|--|
| row        | int  | Line number where match occurs (zero-based indexing)         |
| start_col  | int  | Starting column position of match within the line            |
| end_col    | int  | Ending column position (exclusive) for highlighting purposes |
| match_len  | int  | Length of matched text for validation and highlighting       |

The search navigation system allows users to move between matches using designated keys (typically F3 for next match, Shift+F3 for previous match). When navigation is requested, the system updates the `current_match` index, wrapping around at the boundaries if wrap-around is enabled. The cursor position is then updated to move to the selected match, and the viewport is adjusted to ensure the match is visible on screen.

The critical design insight is that search state is completely separate from text buffer state. The search system never modifies the underlying text content, instead maintaining parallel tracking of match positions that overlay the existing content during rendering.

## Search State Management

The search system operates as a finite state machine with distinct states that determine how input is processed and what visual feedback is provided. This state-driven approach ensures consistent behavior and clear separation between search functionality and normal editing operations.

| Current State   | Input Event       | Next State      | Action Taken  |
|-----------------|-------------------|-----------------|---|
| SEARCH_INACTIVE | Ctrl+F pressed    | SEARCH_ACTIVE   | Initialize search, clear matches, show search prompt        |
| SEARCH_ACTIVE   | Regular character | SEARCH_ACTIVE   | Append to query, update matches, refresh screen             |
| SEARCH_ACTIVE   | Backspace         | SEARCH_ACTIVE   | Remove last query char, update matches, refresh screen      |
| SEARCH_ACTIVE   | Escape pressed    | SEARCH_INACTIVE | Clear search state, remove highlighting, return to normal   |
| SEARCH_ACTIVE   | Enter pressed     | SEARCH_INACTIVE | Accept current match, clear search UI, keep cursor at match |
| SEARCH_ACTIVE   | F3 pressed        | SEARCH_ACTIVE   | Navigate to next match, update cursor position              |
| SEARCH_ACTIVE   | Shift+F3 pressed  | SEARCH_ACTIVE   | Navigate to previous match, update cursor position          |

The transition between search states involves careful coordination with other editor components. When entering search mode, the search system must:

1. Save the current cursor position as the search starting point for relative match selection
2. Initialize the search query buffer and reset all match tracking arrays
3. Signal the Screen Renderer Component to begin displaying search-specific UI elements
4. Configure the Input Handler Component to route keystrokes to search processing functions
5. Update the status bar to show the search prompt and current query

When exiting search mode, the cleanup process ensures no search artifacts remain:

1. Clear all match highlighting by removing search-specific display attributes
2. Restore normal input processing by returning control to the standard input handler
3. Reset the status bar to show normal editor status information
4. Optionally restore the original cursor position if search was cancelled with Escape
5. Free any dynamically allocated memory used for match tracking

## Common Pitfalls

**⚠ Pitfall: Search Query Buffer Overflow** Many implementations fail to properly bounds-check the search query input, leading to buffer overflows when users type very long search strings. The search query buffer must have a reasonable maximum length (typically 256 characters), and input processing should reject

additional characters once this limit is reached. Always validate query length before appending characters and provide visual feedback when the limit is reached.

**⚠ Pitfall: Stale Match Positions After Text Editing** If the user somehow triggers text editing operations while in search mode (through key combinations not properly filtered), the match positions become invalid as line numbers and column positions shift. The search system must either prevent all text modification while active, or detect buffer changes and invalidate current search results. Most implementations choose the prevention approach by ensuring comprehensive input filtering in search mode.

**⚠ Pitfall: Performance Degradation on Large Files** Performing full buffer scans on every character input becomes prohibitively slow for large files (>10MB). While immediate feedback is ideal for typical files, large files may require optimization techniques such as limiting search to visible text plus a reasonable buffer, implementing incremental search that builds on previous results, or switching to debounced updates for files above a size threshold.

**⚠ Pitfall: Case Sensitivity Confusion** Users often expect consistent case sensitivity behavior, but many implementations have unclear or inconsistent rules. The search system should have a clear default (typically case-insensitive) and provide obvious visual indication of the current case sensitivity mode. Toggle functionality should be discoverable and the current setting should be displayed in the search prompt.

## Match Highlighting

The match highlighting system provides immediate visual feedback by marking all search matches within the visible text using ANSI color codes and background colors. Think of it as a spotlight system that illuminates relevant text while dimming everything else, helping users quickly identify patterns and navigate to areas of interest.

The highlighting system integrates closely with the Screen Renderer Component's existing line rendering pipeline. Rather than implementing a separate highlighting pass, match highlighting is incorporated into the normal character-by-character rendering process. As each character is processed for display, the renderer checks whether that character position falls within any active search match, applying appropriate color codes when matches are detected.

The highlighting system distinguishes between different types of matches to provide clear visual hierarchy:

| Match Type             | Visual Treatment                     | ANSI Codes  | Usage  |
|------------------------|--------------------------------------|-------------|--|
| Current Match          | Bright yellow background, black text | \x1b[43;30m | The match that cursor will jump to, primary focus  |
| Other Matches          | Cyan background, black text          | \x1b[46;30m | All other matches in visible area, secondary focus |
| Search Query in Status | White text, blue background          | \x1b[47;34m | Query display in status bar for reference          |
| No Matches             | Red text in status bar               | \x1b[31m    | Indicates no matches found for current query       |

The highlighting process integrates into the existing line rendering workflow through match position checking. For each character being rendered, the system performs a quick lookup to determine if that character position falls within any search match boundaries. This lookup must be efficient since it occurs for every visible character on every screen refresh.

Here's the detailed highlighting integration process:

1. During screen rendering, the renderer processes each visible line character by character
2. For each character position, the system checks if that position falls within any recorded search match
3. If a match is found, the renderer determines whether it's the current match or a secondary match
4. Appropriate ANSI color codes are inserted before the character to begin highlighting
5. When the match boundary is reached, reset codes are inserted to end the highlighting
6. The process continues for all characters, with multiple matches potentially overlapping or adjacent

The match position lookup requires efficient data structures to avoid performance degradation during rendering. The search system maintains its match array sorted by line number and column position, allowing binary search techniques to quickly locate relevant matches for each line being rendered.

### Decision: Highlight All Matches vs Current Match Only

- **Context:** Search highlighting could show only the current match or all matches simultaneously
- **Options Considered:** Current match only, all visible matches, all matches with current highlighted differently
- **Decision:** All matches with current match specially highlighted
- **Rationale:** Showing all matches provides better context and helps users understand the distribution of search results throughout the document. Special highlighting of current match maintains focus while preserving overview.
- **Consequences:** Requires more complex rendering logic and color management, but significantly improves search usability and pattern recognition.

| Highlighting Strategy     | Pros                                     | Cons  | Chosen |
|---------------------------|--|---|--------|
| Current match only        | Simple implementation, clear focus       | No context of other matches, poor overview          | X      |
| All matches same color    | Shows distribution, simple color logic   | Can't distinguish current match, navigation unclear | X      |
| Hierarchical highlighting | Best context and focus, clear navigation | Complex rendering, more color codes needed          | ✓      |

The highlighting system must handle several edge cases that can cause rendering artifacts:

**Multi-line Match Handling:** While most search queries match within single lines, the system must gracefully handle cases where matches span line boundaries (uncommon but possible with certain queries). The highlighting system treats each line independently, so multi-line matches appear as separate highlighted segments.

**Overlapping Highlight Regions:** When syntax highlighting is also active, search highlights must take precedence over syntax colors to ensure search results remain visible. The rendering system applies search highlights after syntax highlighting, effectively overriding syntax colors within match regions.

**Viewport Scrolling Coordination:** As users navigate between search matches, the viewport must scroll to ensure the current match remains visible. The highlighting system coordinates with the viewport management to trigger scrolling when the current match moves outside the visible area.

**Color Reset Management:** Proper ANSI color code management ensures that search highlighting doesn't interfere with subsequent text rendering. Each highlighted region must be properly closed with reset codes, and the renderer must track color state to restore previous highlighting after search matches.

The status bar integration provides additional search context by displaying the current query, match count, and current match position. This information helps users understand the scope of their search and their current position within the results.

| Status Bar Element | Format                      | Example                 | Purpose                                    |
|--------------------|-----------------------------|-------------------------|--|
| Search Query       | Search: [query]             | Search: function        | Shows current search string being built    |
| Match Count        | [current/total]             | [3/15]                  | Shows position within total results        |
| No Results         | No matches for<br>'[query]' | No matches for<br>'xyz' | Clear feedback when no matches found       |
| Case Mode          | [case sensitive]            | [Aa]                    | Indicates current case sensitivity setting |

## Implementation Guidance

The search system requires careful coordination between input handling, text processing, and rendering components. The implementation focuses on maintaining responsive user experience while providing comprehensive search functionality.

### A. Technology Recommendations:

| Component        | Simple Option                                  | Advanced Option                             |
|------------------|--|---|
| String Matching  | <code>strstr()</code> and manual case handling | Boyer-Moore or KMP algorithm implementation |
| Match Storage    | Fixed-size array with maximum match limit      | Dynamic array with reallocation             |
| State Management | Global search state variables                  | Encapsulated search state structure         |
| Highlighting     | Direct ANSI code insertion                     | Attribute-based rendering system            |

### B. Recommended File Structure:

```

editor/
  src/
    search.c      ← search functionality implementation
    search.h      ← search system interface and structures
    editor.c      ← main editor with search integration
    render.c      ← screen renderer with highlight support
    input.c       ← input handler with search mode support
  
```

### C. Infrastructure Starter Code:

```
// search.h - Complete search system interface

#ifndef SEARCH_H

#define SEARCH_H


#include <time.h>

// Search state enumeration

typedef enum {

    SEARCH_INACTIVE,

    SEARCH_ACTIVE

} SearchState;

// Search direction for navigation

typedef enum {

    SEARCH_FORWARD,

    SEARCH_BACKWARD

} SearchDirection;

// Individual match position

typedef struct {

    int row;

    int start_col;

    int end_col;

    int match_len;

} SearchMatch;

// Complete search system state

typedef struct {

    SearchState state;
```

C

```
char query[256];

int query_len;

SearchMatch *matches;

int match_count;

int match_capacity;

int current_match;

SearchDirection direction;

int case_sensitive;

int wrap_around;

int saved_cx; // cursor position when search started

int saved_cy;

} SearchSystem;

// Global search system instance

extern SearchSystem search_state;

// Core search functions

void search_init(void);

void search_cleanup(void);

void search_activate(void);

void search_deactivate(int accept_match);

void search_add_char(int c);

void search_delete_char(void);

void search_next_match(void);

void search_prev_match(void);

int search_is_active(void);

// Match detection and highlighting
```

```
void search_update_matches(void);

int search_char_in_match(int row, int col, int *is_current);

void search_move_to_match(int match_index);

#endif
```

```
// search.c - Search system implementation with complete infrastructure

#include "search.h"

#include "editor.h"

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

SearchSystem search_state = {0};

void search_init(void) {

    search_state.state = SEARCH_INACTIVE;

    search_state.query[0] = '\0';

    search_state.query_len = 0;

    search_state.matches = NULL;

    search_state.match_count = 0;

    search_state.match_capacity = 0;

    search_state.current_match = -1;

    search_state.direction = SEARCH_FORWARD;

    search_state.case_sensitive = 0;

    search_state.wrap_around = 1;

}

void search_cleanup(void) {

    if (search_state.matches) {

        free(search_state.matches);

        search_state.matches = NULL;

    }

    search_state.match_capacity = 0;
```

C

```
search_state.match_count = 0;

}

void search_activate(void) {

    search_state.state = SEARCH_ACTIVE;

    search_state.query[0] = '\0';

    search_state.query_len = 0;

    search_state.current_match = -1;

    search_state.match_count = 0;

    // Save current cursor position

    extern struct editor_config E;

    search_state.saved_cx = E.cx;

    search_state.saved_cy = E.cy;

    // Update status bar

    editor_set_status_message("Search: %s", search_state.query);

}

int search_is_active(void) {

    return search_state.state == SEARCH_ACTIVE;

}

// Helper function for case-insensitive comparison

static int char_match(char a, char b, int case_sensitive) {

    if (case_sensitive) {

        return a == b;

    }

}
```

```
    return tolower(a) == tolower(b);

}

// Efficient string search within a line

static void find_matches_in_line(int row, const char* line, int line_len) {

    if (search_state.query_len == 0) return;

    for (int col = 0; col <= line_len - search_state.query_len; col++) {

        int match = 1;

        for (int i = 0; i < search_state.query_len; i++) {

            if (!char_match(line[col + i], search_state.query[i],
                search_state.case_sensitive)) {

                match = 0;

                break;
            }
        }

        if (match) {

            // Ensure capacity for new match

            if (search_state.match_count >= search_state.match_capacity) {

                search_state.match_capacity = search_state.match_capacity ?
                    search_state.match_capacity * 2 : 16;

                search_state.matches = realloc(search_state.matches,
                    search_state.match_capacity * sizeof(SearchMatch));
            }
        }
    }
}
```

// Record the match

```
SearchMatch *match_ptr = &search_state.matches[search_state.match_count];
```

```
    match_ptr->row = row;

    match_ptr->start_col = col;

    match_ptr->end_col = col + search_state.query_len;

    match_ptr->match_len = search_state.query_len;

    search_state.match_count++;

}

}

}
```

#### D. Core Logic Skeleton Code:

```
void search_add_char(int c) {

    // TODO 1: Check if query buffer has space (query_len < 255)

    // TODO 2: Append character to query string and increment length

    // TODO 3: Null-terminate the query string

    // TODO 4: Call search_update_matches() to find new matches

    // TODO 5: Update current match to closest match after cursor

    // TODO 6: Update status bar with new query and match count

    // TODO 7: Trigger screen refresh to show highlighting

    // Hint: Use editor_set_status_message() for status updates

}

void search_update_matches(void) {

    // TODO 1: Reset match count to 0 (keep allocated memory)

    // TODO 2: If query is empty, return early (no matches to find)

    // TODO 3: Get editor config to access text buffer (extern struct editor_config E)

    // TODO 4: Loop through all rows in buffer (0 to E.numrows)

    // TODO 5: For each row, call find_matches_in_line() to search for query

    // TODO 6: After finding all matches, determine current match index

    // TODO 7: Current match should be first match at or after cursor position

    // TODO 8: If no matches after cursor and wrap_around enabled, use first match

    // Hint: Compare match row/col with E.cy/E.cx for cursor position

}

void search_next_match(void) {

    // TODO 1: Check if any matches exist (match_count > 0)

    // TODO 2: If no current match selected, select first match (index 0)

    // TODO 3: Otherwise increment current_match index

    // TODO 4: Handle wrap-around if at end of matches and wrap_around enabled
```

```

// TODO 5: Call search_move_to_match() with new match index

// TODO 6: Update status bar to show new match position

// Hint: Use modulo operator for wrap-around: (current + 1) % match_count

}

int search_char_in_match(int row, int col, int *is_current) {

    // TODO 1: Initialize is_current to 0 (false)

    // TODO 2: Loop through all matches in search_state.matches array

    // TODO 3: For each match, check if row matches and col is within range

    // TODO 4: Range check: col >= start_col && col < end_col

    // TODO 5: If match found, check if this is the current match index

    // TODO 6: Set is_current to 1 if this match index equals current_match

    // TODO 7: Return 1 if position is in any match, 0 otherwise

    // Hint: Early return on first match found for efficiency

}

void search_move_to_match(int match_index) {

    // TODO 1: Validate match_index is within bounds (0 to match_count-1)

    // TODO 2: Get the match structure from matches array

    // TODO 3: Update editor cursor position (E.cx, E.cy) to match location

    // TODO 4: Set E.cy to match row, E.cx to match start_col

    // TODO 5: Call editor_scroll() to ensure match is visible on screen

    // TODO 6: Update current_match to the new index

    // TODO 7: Trigger screen refresh to update cursor position

    // Hint: extern struct editor_config E for accessing editor state

}

```

#### E. Language-Specific Hints:

- Use `strstr()` for simple case-sensitive string search, implement custom loop for case-insensitive
- `realloc()` for dynamic match array growth - always check return value for allocation failure
- `tolower()` from `<ctype.h>` for case-insensitive comparison
- Store ANSI color codes as `#define` constants for maintainability: `#define SEARCH_HIGHLIGHT "\x1b[43;30m"`
- Use `snprintf()` for safe string formatting in status messages to prevent buffer overflows

## F. Milestone Checkpoint:

After implementing the search system, verify the following behavior:

1. **Search Activation:** Press Ctrl+F - status bar should show "Search: " prompt
2. **Incremental Search:** Type characters - matches should highlight immediately as you type
3. **Match Navigation:** Press F3/Shift+F3 - cursor should jump between highlighted matches
4. **Visual Feedback:** Current match should have different highlighting than other matches
5. **Search Exit:** Press Escape - search highlighting should clear and cursor should remain at current match

Expected test sequence:

```
# Create test file with repeated text
echo -e "function test()\nfunction main()\nint function_call()" > test.c

./editor test.c

# Press Ctrl+F, type "func", verify 3 matches highlighted

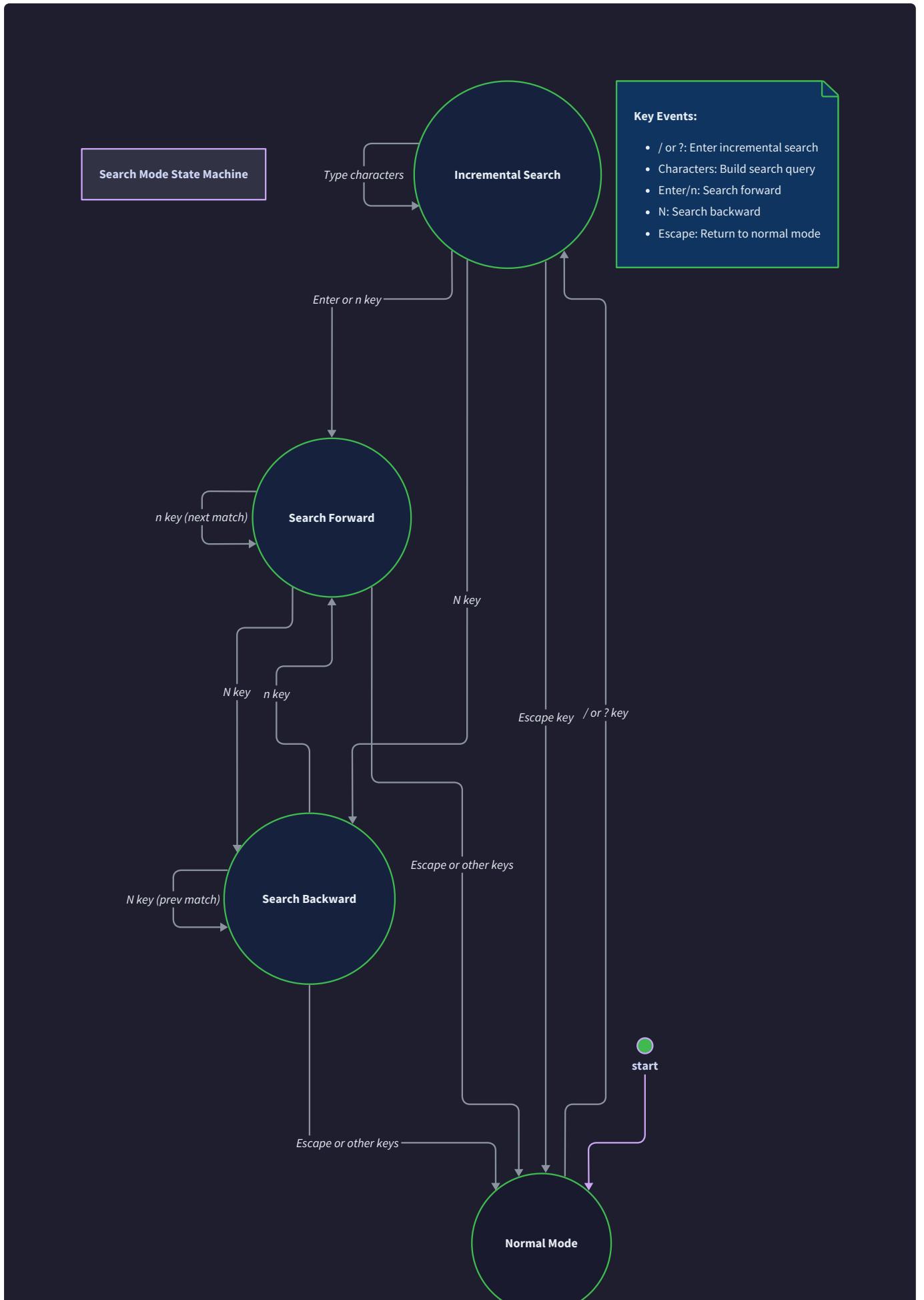
# Press F3 twice, verify cursor moves through matches

# Press Escape, verify highlighting clears
```

BASH

## G. Debugging Tips:

| Symptom                           | Likely Cause                      | How to Diagnose                                      | Fix  |
|-----------------------------------|-----------------------------------|--|--|
| No matches highlight when typing  | Match finding logic not working   | Add printf in<br><code>find_matches_in_line()</code> | Check string comparison and loop bounds                            |
| Matches highlight wrong text      | Incorrect column calculation      | Print match coordinates                              | Verify start_col/end_col calculation                               |
| Current match not distinguishable | Color codes not different         | Check ANSI codes in terminal                         | Use distinct background colors                                     |
| Search doesn't exit cleanly       | State not properly reset          | Check search_state values after exit                 | Clear all state variables in<br><code>search_deactivate()</code>   |
| Cursor jumps to wrong position    | Match navigation coordinate error | Print cursor position before/after move              | Verify E.cx/E.cy updates in<br><code>search_move_to_match()</code> |



# Syntax Highlighting Component

**Milestone(s):** Milestone 7 (Syntax Highlighting)

Think of syntax highlighting as a skilled librarian who can instantly recognize different types of books just by glancing at their content. Just as a librarian might use colored tabs to mark fiction (blue), non-fiction (green), and reference materials (red), our syntax highlighting system applies different colors to code elements based on their meaning and context. The librarian doesn't need to read every word to categorize a book—they recognize patterns like chapter structures, citation formats, and writing styles. Similarly, our syntax highlighter identifies programming language constructs through pattern recognition, applying visual cues that help developers quickly understand code structure and meaning.

The syntax highlighting component transforms plain text into a visually rich representation where keywords appear in one color, strings in another, and comments in yet another. This isn't merely cosmetic decoration—it's a cognitive aid that leverages the human visual system's ability to process color and pattern information faster than textual content. When a developer sees `if` highlighted in blue, they immediately recognize it as a control flow keyword without conscious parsing, freeing mental resources for higher-level code comprehension.

Our syntax highlighting system operates as a three-stage pipeline that mirrors how a human might colorize code by hand. First, it examines the file extension to determine which programming language rules to apply—just as you might look at a `.py` file and know to expect Python syntax. Second, it scans through the text character by character, identifying different types of language elements through pattern matching and state tracking. Finally, it applies the appropriate ANSI color codes to each identified element during the rendering process, ensuring that the colorized output integrates seamlessly with our existing screen refresh system.

## Language Detection

The language detection subsystem functions as the entry point to syntax highlighting, determining which set of rules and patterns should be applied to colorize the current file. Think of it as a customs officer at an airport who quickly categorizes travelers based on their passports—the file extension serves as the "passport" that determines which "country's laws" (syntax rules) apply to the content.

The detection process begins whenever a file is opened or the filename changes, triggering a lookup in our language registry. This registry maps file extensions to language definitions, with each language definition containing the keywords, patterns, and color mappings specific to that programming language. The system supports multiple extensions per language (`.c` and `.h` for C, `.js` and `.jsx` for JavaScript) and includes a fallback mechanism for unknown extensions that applies minimal highlighting focused on common patterns like strings and comments.

| File Extension | Language   | Keywords   | String Patterns             | Comment Patterns |
|----------------|------------|--|-----------------------------|------------------|
| .c, .h         | C          | if, else, for, while, int, char, void, return    | Single/double quoted        | // and /* */     |
| .py            | Python     | if, else, for, while, def, class, import, return | Single/double/triple quoted | # and """ blocks |
| .js, .jsx      | JavaScript | if, else, for, while, function, var, let, const  | Single/double quoted        | // and /* */     |
| .go            | Go         | if, else, for, func, var, const, import, return  | Backtick/double quoted      | // and /* */     |
| .rs            | Rust       | if, else, for, while, fn, let, mut, impl         | Single/double quoted        | // and /* */     |
| unknown        | Generic    | Common keywords                                  | Single/double quoted        | // and #         |

The language detection system maintains a `LanguageDefinition` structure for each supported language, containing the vocabulary and patterns needed for accurate highlighting. Each definition includes arrays of keywords organized by category (control flow, types, modifiers), regular expressions for identifying strings and comments, and color mappings that specify which visual treatment each category receives.

### Decision: File Extension-Based Detection

- **Context:** Need to determine syntax highlighting rules for open files
- **Options Considered:** File extension mapping, content analysis, user specification
- **Decision:** Primary detection via file extension with content-based fallback
- **Rationale:** File extensions provide immediate, reliable language identification without processing overhead. Content analysis would require parsing entire files and could be ambiguous for small code snippets.
- **Consequences:** Fast detection with minimal CPU overhead, but may misidentify files with non-standard extensions or mixed content

The detection algorithm follows a priority-based lookup sequence to handle edge cases and ambiguous files gracefully. First, it checks for an exact extension match in the primary language registry. If no match is found, it attempts partial matching for compound extensions like `.test.js` or `.config.py` by extracting the final component. As a final fallback, it applies a generic highlighting scheme that recognizes common programming constructs without language-specific keywords.

When the language changes (such as when saving a file with a new extension), the system triggers a complete re-highlighting of the visible buffer content. This process updates the render representation of each

visible row with the new color codes, ensuring that the screen refresh shows the appropriate syntax coloring immediately. The re-highlighting process is optimized to only process visible rows, deferring off-screen content until it becomes visible through scrolling.

Language definitions are stored in a static lookup table initialized at startup, avoiding dynamic memory allocation during the detection process. Each language definition includes a priority field that resolves conflicts when multiple languages could match the same extension, with more specific languages (like TypeScript for `.ts`) taking precedence over generic ones (like text for unknown extensions).

## Simple Tokenization

The tokenization subsystem breaks down source code text into meaningful units that can be individually colored, operating much like a proofreader who marks different parts of speech with different colored pens. Unlike a full compiler tokenizer that must handle every syntactic detail, our simple tokenizer focuses on the most visually important elements: keywords, strings, comments, numbers, and operators. This selective approach provides maximum visual benefit while maintaining the performance characteristics necessary for real-time highlighting during typing.

The tokenization process operates as a character-by-character state machine that tracks context as it scans through each line of text. The state machine maintains knowledge of whether it's currently inside a string literal, within a comment block, or scanning regular code, allowing it to apply different parsing rules based on context. This stateful approach correctly handles complex cases like escaped quotes within strings or nested comment blocks, ensuring that color boundaries align with actual language semantics.

| Token Type     | Recognition Pattern                     | State Transitions                | Color Assignment                |
|----------------|---|----------------------------------|---------------------------------|
| Keyword        | Word boundary + dictionary lookup       | Normal → Normal                  | Language-specific keyword color |
| String Literal | Quote characters with escape handling   | Normal → InString → Normal       | String literal color            |
| Single Comment | <code>//</code> to end of line          | Normal → InComment → Normal      | Comment color                   |
| Block Comment  | <code>/* */</code> with nesting support | Normal → InBlockComment → Normal | Comment color                   |
| Number Literal | Digit sequences with decimal support    | Normal → Normal                  | Number literal color            |
| Identifier     | Alphanumeric + underscore sequences     | Normal → Normal                  | Default text color              |

The tokenization algorithm processes each character in sequence, maintaining a current token buffer and a state variable that tracks the parsing context. When it encounters a state transition trigger (like an opening

quote or comment delimiter), it finalizes the current token, records its type and position, and transitions to the new state. The token buffer accumulates characters until a boundary is reached, at which point the complete token is classified and added to the line's token list.

String tokenization requires special handling for escape sequences and multi-line constructs. When the state machine enters a string context, it continues accumulating characters until it finds the matching closing quote, but it must ignore escaped quotes that are part of the string content rather than delimiters. The system supports both single and double-quoted strings, with language-specific rules for which quote types are valid and whether triple-quoted strings (like Python docstrings) should receive special treatment.

Comment tokenization handles both single-line and block comment styles, with language-specific rules for comment delimiters. Single-line comments extend from their delimiter to the end of the current line, making them simple to tokenize within a line-by-line processing model. Block comments present more complexity because they can span multiple lines, requiring the tokenization state to persist across line boundaries. The system maintains a per-buffer comment state that tracks whether processing is currently within a multi-line comment block.

Keyword recognition operates through dictionary lookup after identifying word boundaries. The system first extracts complete words (sequences of alphanumeric characters and underscores), then checks whether each word appears in the current language's keyword dictionary. This approach avoids false positives where keywords appear as substrings within larger identifiers (like `interface` within `MyInterface`), ensuring that only standalone keywords receive highlighting.

Number literal recognition identifies both integer and floating-point numeric constants through pattern matching. The tokenizer recognizes decimal numbers, hexadecimal constants (prefixed with `0x`), and floating-point numbers with optional exponent notation. Language-specific rules handle variations like binary literals (`0b` prefix) or separator characters (underscores in large numbers) where supported by the target language.

### Decision: Line-by-Line Tokenization

- **Context:** Need to balance highlighting accuracy with performance for real-time editing
- **Options Considered:** Full file parsing, line-by-line processing, token caching
- **Decision:** Line-by-line tokenization with minimal cross-line state tracking
- **Rationale:** Matches our line-based text buffer structure and provides acceptable performance for typing responsiveness. Cross-line constructs like block comments require state persistence but don't justify full file parsing complexity.
- **Consequences:** Fast highlighting updates during editing with minor limitations on complex multi-line constructs

The tokenization process integrates with the existing `editor_update_row` function, extending it to generate syntax highlighting information alongside the render representation. When a line's content changes, the system re-tokenizes that line and updates its color information, ensuring that highlighting remains accurate as

the user types. The tokenization results are stored in an extended version of the `erow` structure that includes token position and type information.

Performance optimization focuses on minimizing work during interactive editing sessions. The tokenizer processes only changed lines rather than re-analyzing the entire file, and it reuses token information for unchanged content. Token boundary detection uses efficient character classification techniques, avoiding expensive regular expression matching in favor of simple character comparisons and table lookups.

Error recovery mechanisms handle malformed input gracefully, ensuring that tokenization errors don't prevent highlighting of valid content. When the tokenizer encounters unexpected characters or unclosed strings, it applies heuristic recovery rules that make reasonable assumptions about the intended structure. These recovery mechanisms prevent highlighting from failing completely when files contain syntax errors or incomplete constructs.

## Color Code Application

The color code application subsystem serves as the final stage in the syntax highlighting pipeline, translating abstract token type information into concrete ANSI escape sequences that produce colored output on the terminal. Think of it as a painter's palette where each token type corresponds to a specific color, and the application process involves dipping the brush in the right color before painting each word on the canvas. The system must seamlessly integrate with the existing screen rendering pipeline, ensuring that syntax colors appear correctly without disrupting cursor positioning, line numbers, or other screen elements.

The color application process occurs during screen rendering, specifically within the line formatting phase where the system converts buffer content into terminal output. As the renderer processes each character of a line, it consults the tokenization results to determine whether a color change is needed at the current position. When a token boundary is encountered, the system emits the appropriate ANSI color sequence before outputting the token's characters, then resets to the default color when the token ends.

| Token Type     | ANSI Color Code       | Visual Appearance   | Usage Context                   |
|----------------|-----------------------|---------------------|---------------------------------|
| Keyword        | <code>\x1b[34m</code> | Blue                | Control flow, type declarations |
| String Literal | <code>\x1b[32m</code> | Green               | Text constants, file paths      |
| Comment        | <code>\x1b[90m</code> | Bright Black (Gray) | Documentation, disabled code    |
| Number Literal | <code>\x1b[33m</code> | Yellow              | Numeric constants               |
| Operator       | <code>\x1b[35m</code> | Magenta             | Mathematical, logical operators |
| Default Text   | <code>\x1b[0m</code>  | Terminal default    | Identifiers, whitespace         |

The color mapping system uses a configurable palette that can be adjusted for different terminal capabilities and user preferences. The default color scheme follows common conventions established by popular text editors and IDEs, using blue for keywords (conveying authority and structure), green for strings (suggesting

content and data), and gray for comments (indicating secondary importance). These color choices leverage established developer expectations while ensuring readability across different terminal backgrounds.

Color code emission integrates with the existing frame buffer system, ensuring that ANSI sequences are included in the atomic screen updates that prevent flicker. The renderer builds complete lines with embedded color codes before adding them to the frame buffer, avoiding partial color state that could cause display artifacts. Color reset sequences are strategically placed to ensure that syntax highlighting doesn't affect other screen elements like the status bar or line numbers.

The application algorithm processes each line by iterating through its characters while simultaneously consulting the token information generated during tokenization. When the character position reaches a token boundary, the system emits the color code for the new token type and updates its current color state. This approach ensures that color changes occur at precisely the right character positions without requiring complex string manipulation or multiple passes through the content.

State management tracks the current active color to avoid emitting redundant ANSI sequences that would increase output size without visual benefit. When consecutive tokens use the same color (such as multiple keywords in sequence), the system omits the redundant color change sequences. This optimization reduces the volume of terminal output and improves rendering performance, especially important for files with extensive syntax highlighting.

### Decision: Inline Color Code Emission

- **Context:** Need to apply syntax colors during screen rendering without affecting performance
- **Options Considered:** Pre-colored string storage, inline emission during rendering, post-processing color injection
- **Decision:** Inline ANSI code emission during line rendering with state tracking
- **Rationale:** Integrates naturally with existing character-by-character rendering loop and avoids memory overhead of storing colored strings. State tracking prevents redundant sequences.
- **Consequences:** Minimal memory overhead with good performance, but requires careful coordination between tokenization and rendering phases

Color persistence handles multi-line constructs like block comments that span across line boundaries. The rendering system maintains a cross-line color state that tracks whether the line begins within a multi-line token, ensuring that the appropriate color is applied from the start of the line rather than only at token boundaries within the line. This persistence mechanism works in conjunction with the tokenization state tracking to provide visually consistent highlighting for complex constructs.

Terminal compatibility considerations ensure that the color codes work correctly across different terminal emulators and configurations. The system uses standard ANSI color codes from the 8-color palette, which are widely supported, rather than extended 256-color or true-color sequences that might not display correctly on all terminals. Fallback mechanisms detect terminals with limited color support and adjust the color scheme accordingly.

The color application system includes debugging support that can optionally display token boundaries and type information for development and troubleshooting purposes. When debugging mode is enabled, the system can output token information to a separate debug file or overlay token boundaries on the display, helping developers understand how the tokenization and coloring systems interpret their code.

Integration with the search highlighting system requires careful coordination to ensure that search matches remain visible when syntax highlighting is active. The color application system recognizes search match regions and applies search highlighting colors with higher priority than syntax colors, temporarily overriding the normal token coloring within match boundaries. This priority system ensures that users can effectively use search functionality without losing the visual feedback provided by match highlighting.

**⚠ Pitfall: Color State Leakage** Many implementations accidentally allow syntax highlighting colors to "leak" into other parts of the screen like the status bar or line numbers. This happens when the color reset sequence is omitted or placed incorrectly, causing subsequent text to inherit the color from the last highlighted token. Always emit a color reset sequence at the end of each line's content before moving to the next screen element, and verify that the status bar and other UI elements maintain their intended appearance.

**⚠ Pitfall: Performance Degradation with Large Files** Naive implementations often re-tokenize entire files when any change occurs, causing noticeable delays when editing large source files. The tokenization should be incremental, processing only changed lines and reusing existing token information for unchanged content. Additionally, avoid tokenizing off-screen content until it becomes visible through scrolling—there's no benefit to highlighting text the user cannot see.

**⚠ Pitfall: Incorrect Multi-line Comment Handling** Block comments that span multiple lines are frequently handled incorrectly, either by failing to maintain state across line boundaries or by not properly detecting comment end sequences. Ensure that the tokenization state persists across lines within block comments, and carefully handle cases where comment end sequences appear at the beginning of lines or are split across buffer boundaries during editing operations.

## Implementation Guidance

The syntax highlighting component integrates with the existing screen rendering pipeline while adding new data structures for language definitions and token information. The implementation focuses on performance and simplicity, avoiding complex parsing techniques in favor of straightforward pattern matching and state machines.

### A. Technology Recommendations Table:

| Component          | Simple Option                             | Advanced Option                                       |
|--------------------|---|---|
| Language Detection | Static lookup table with file extensions  | Configurable language registry with pattern matching  |
| Tokenization       | Character-by-character state machine      | Regex-based pattern matching with compiled patterns   |
| Color Application  | Direct ANSI code emission                 | Color theme system with terminal capability detection |
| Token Storage      | Extended erow structure with token arrays | Separate token buffer with position indexing          |

## B. Recommended File Structure:

```

src/
  editor.h           ← add syntax highlighting declarations
  editor.c           ← main editor implementation
  syntax.c           ← syntax highlighting implementation
  syntax.h           ← syntax highlighting interface
  languages/
    c_lang.c          ← language definition files
    python_lang.c      ← C language definition
    generic_lang.c     ← Python language definition
    fallback_lang.c    ← fallback language definition
  Makefile            ← updated build configuration

```

## C. Infrastructure Starter Code:

Extended `erow` structure with syntax highlighting support:

```
#ifndef SYNTAX_H

#define SYNTAX_H

#include <time.h>

// Token types for syntax highlighting

typedef enum {

    TOKEN_NORMAL = 0,

    TOKEN_KEYWORD,

    TOKEN_STRING,

    TOKEN_COMMENT,

    TOKEN_NUMBER,

    TOKEN_OPERATOR,

    TOKEN_TYPE,

    TOKEN_PREPROCESSOR

} TokenType;
```

```
// Individual token within a line

typedef struct {

    int start;           // Starting column of token

    int end;             // Ending column of token

    TokenType type;     // Token classification

} Token;
```

```
// Extended row structure with syntax highlighting

typedef struct erow {

    int size;

    char *chars;

    int rsize;
```

C

```

char *render;

Token *tokens;      // Array of tokens for this line

int token_count;    // Number of tokens

int token_capacity; // Allocated token array size

unsigned char *color; // Color code for each render character

} erow;

// Language definition structure

typedef struct {

    char *name;

    char **extensions;      // Null-terminated array of file extensions

    char **keywords;        // Null-terminated array of keywords

    char *single_comment;   // Single-line comment delimiter (e.g., "//")

    char *multi_comment_start; // Block comment start (e.g., "/*")

    char *multi_comment_end; // Block comment end (e.g., "*/")

    char *string_chars;     // String delimiter characters (e.g., "\"'")

    int flags;              // Language-specific flags

} LanguageDefinition;

// Global syntax highlighting state

typedef struct {

    LanguageDefinition *current_language;

    int in_multi_comment; // Tracking multi-line comment state

    int highlight_enabled; // Whether highlighting is active

} SyntaxState;

// Color definitions

#define COLOR_NORMAL      "\x1b[0m"

```

```
#define COLOR_KEYWORD    "\x1b[34m"      // Blue
#define COLOR_STRING     "\x1b[32m"      // Green
#define COLOR_COMMENT    "\x1b[90m"      // Bright black (gray)
#define COLOR_NUMBER     "\x1b[33m"      // Yellow
#define COLOR_OPERATOR   "\x1b[35m"      // Magenta
#define COLOR_TYPE       "\x1b[36m"      // Cyan

// Function declarations

void syntax_init(void);
void syntax_cleanup(void);

LanguageDefinition* detect_language(const char *filename);

void tokenize_row(erow *row, LanguageDefinition *lang, int prev_in_comment);

void apply_syntax_colors(erow *row);

int is_keyword(const char *word, LanguageDefinition *lang);

int is_separator(int c);

#endif
```

Language definition for C:

```
#include "syntax.h"                                     C

#include <stdlib.h>

// C language keywords

static char *c_keywords[] = {

    "auto", "break", "case", "char", "const", "continue", "default", "do",
    "double", "else", "enum", "extern", "float", "for", "goto", "if",
    "int", "long", "register", "return", "short", "signed", "sizeof", "static",
    "struct", "switch", "typedef", "union", "unsigned", "void", "volatile", "while",
    NULL
};

// C file extensions

static char *c_extensions[] = {

    ".c", ".h", ".cpp", ".hpp", ".cc", ".cxx", NULL
};

// C language definition

LanguageDefinition c_language = {

    .name = "C",
    .extensions = c_extensions,
    .keywords = c_keywords,
    .single_comment =("//",
    .multi_comment_start = ("/*",
    .multi_comment_end = ("*/",
    .string_chars = ("\\""),
    .flags = 0
};
```

## D. Core Logic Skeleton Code:

Language detection function:

```
// Detect programming language based on filename extension  
// Returns appropriate LanguageDefinition or generic fallback  
  
LanguageDefinition* detect_language(const char *filename) {  
  
    // TODO 1: Extract file extension from filename (find last '.' character)  
  
    // TODO 2: Search through registered languages for matching extension  
  
    // TODO 3: Return matching language definition if found  
  
    // TODO 4: Return generic/fallback language definition if no match  
  
    // Hint: Use strrchr() to find last occurrence of '.' in filename  
  
    // Hint: Use strcasecmp() for case-insensitive extension comparison  
  
}
```

Row tokenization function:

```
// Tokenize a single row, identifying keywords, strings, comments, etc. C

// Updates the row's tokens array and token_count

void tokenize_row(erow *row, LanguageDefinition *lang, int prev_in_comment) {

    // TODO 1: Clear existing tokens and reset token count to 0

    // TODO 2: Initialize state variables (current position, token start, in_string, etc.)

    // TODO 3: Handle multi-line comment state from previous line

    // TODO 4: Loop through each character in the row->chars

    // TODO 5: Implement state machine for different contexts (normal, string, comment)

    // TODO 6: When token boundaries are found, classify and store the token

    // TODO 7: Handle string literals with proper escape sequence support

    // TODO 8: Handle single-line and multi-line comments according to language rules

    // TODO 9: Identify keywords through dictionary lookup

    // TODO 10: Recognize number literals and operators

    // Hint: Use realloc() to grow tokens array as needed

    // Hint: Track whether inside string/comment to handle nested constructs

}
```

Color application function:

```

// Apply syntax highlighting colors to row's render representation

// Fills the row->color array with appropriate ANSI color codes

void apply_syntax_colors(erow *row) {

    // TODO 1: Allocate color array if not already present (size = rsize)

    // TODO 2: Initialize all positions to normal/default color

    // TODO 3: Iterate through tokens in the row

    // TODO 4: For each token, determine appropriate color based on token type

    // TODO 5: Fill color array positions with the token's color code

    // TODO 6: Handle color boundaries and reset sequences properly

    // TODO 7: Ensure search highlighting overrides syntax colors when active

    // Hint: Map TokenType enum values to COLOR_* constants

    // Hint: Color array indices should correspond to render array positions

}

```

Integration with editor refresh screen:

```

// Modified editor_refresh_screen to include syntax highlighting

void editor_refresh_screen() {

    // TODO 1: Ensure current file has been language-detected

    // TODO 2: Re-tokenize any rows that have been modified since last refresh

    // TODO 3: Apply syntax colors to all visible rows

    // TODO 4: During line rendering, emit color codes from row->color array

    // TODO 5: Reset color to normal before rendering status bar

    // TODO 6: Handle cursor positioning with active syntax colors

    // Hint: Check if row needs re-tokenization using a dirty flag

    // Hint: Process only visible rows (rowoff to rowoff + screenrows)

}

```

## E. Language-Specific Hints:

For C implementation:

- Use `strrchr()` to efficiently find file extensions
- Implement `is_separator()` function to identify word boundaries: `return isspace(c) || strchr(", .()+-/*=~%<>[];", c) != NULL;`
- Use `realloc()` for growing dynamic token arrays, doubling capacity each time
- Store ANSI color codes as string constants for easy emission
- Use bitwise flags in `LanguageDefinition` for language-specific features (case sensitivity, nested comments)
- Implement token cleanup in `editor_free_row()` to prevent memory leaks

## F. Milestone Checkpoint:

After implementing syntax highlighting:

### Expected Behavior:

- Open a C source file: `./editor test.c`
- Keywords like `int`, `if`, `for` appear in blue
- String literals in quotes appear in green
- Comments starting with `//` or between `/* */` appear in gray
- Numbers appear in yellow
- Cursor movement and editing work normally with colors

### Verification Steps:

1. Create test file with various C constructs:

```
#include <stdio.h> // This is a comment

int main() {

    char *message = "Hello, World!";

    int count = 42;

    if (count > 0) {

        printf("%s\n", message);

    }

    return 0;
}
```

2. Open in editor and verify color output
3. Test that editing preserves syntax highlighting

4. Try files with different extensions (.py, .js) to verify language detection

### Signs of Problems:

- Colors bleeding into status bar → Missing color reset sequences
- No highlighting visible → Language detection failing or ANSI codes not supported
- Incorrect color boundaries → Token position calculation errors
- Performance issues with large files → Re-tokenizing entire file instead of changed lines

### G. Debugging Tips:

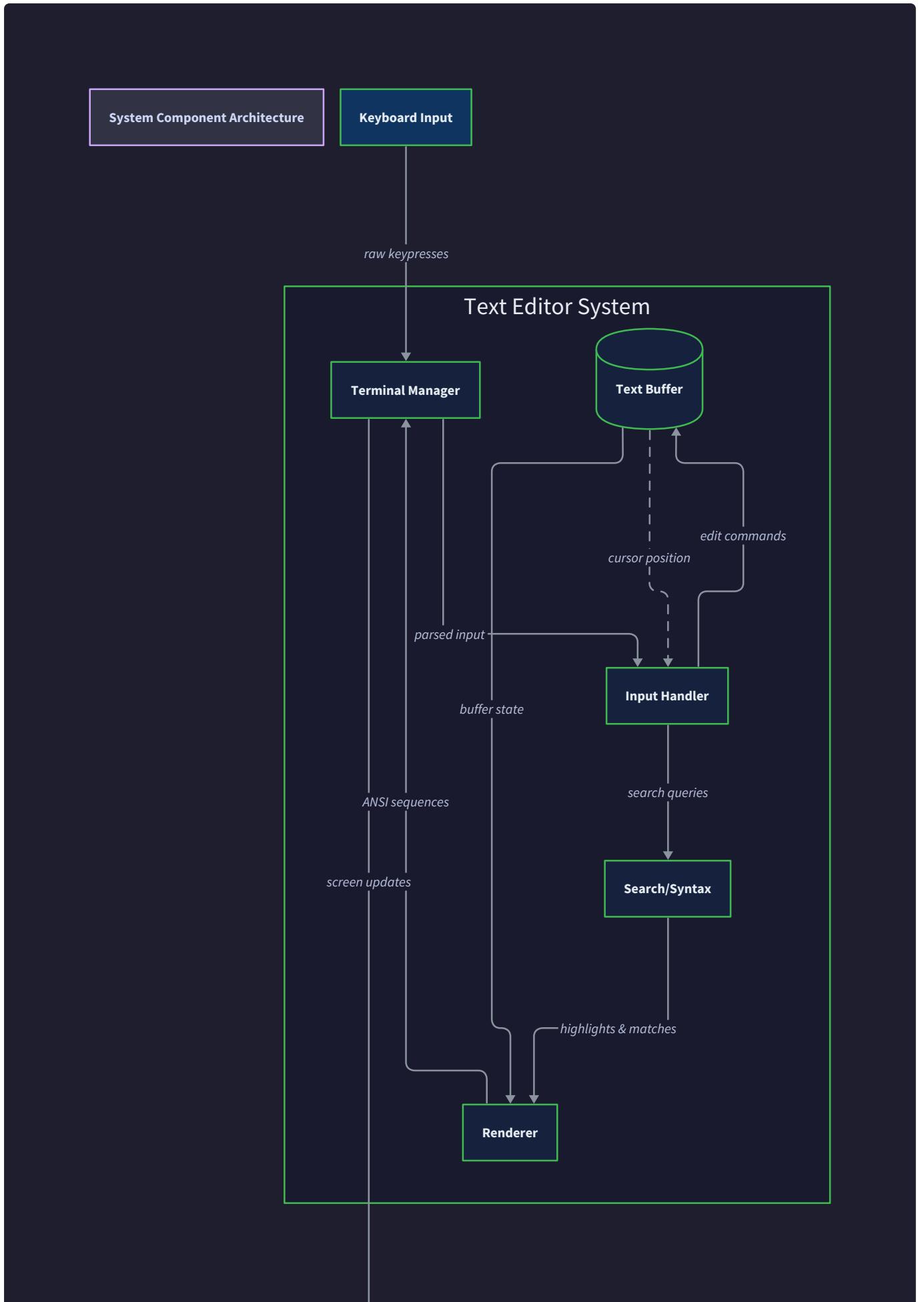
| Symptom                                   | Likely Cause                                    | How to Diagnose                                  | Fix   |
|---|---|--|---|
| No syntax highlighting appears            | Language not detected or ANSI codes not working | Add debug output to show detected language       | Verify file extension matching and terminal color support |
| Colors appear in wrong places             | Token boundaries calculated incorrectly         | Print token positions and types to debug file    | Check character position tracking in tokenization loop    |
| Highlighting disappears when editing      | Tokens not updated after row modification       | Verify tokenize_row called in editor_insert_char | Add tokenization to all text modification functions       |
| Multi-line comments not colored correctly | Comment state not tracked across lines          | Check prev_in_comment parameter usage            | Maintain global comment state in editor_config            |
| Performance slow on large files           | Re-tokenizing entire file on each change        | Profile tokenization calls                       | Only tokenize modified and visible rows                   |
| String highlighting broken with escapes   | Escape sequences not handled in tokenizer       | Test with strings containing " sequences         | Add proper escape sequence parsing in string state        |

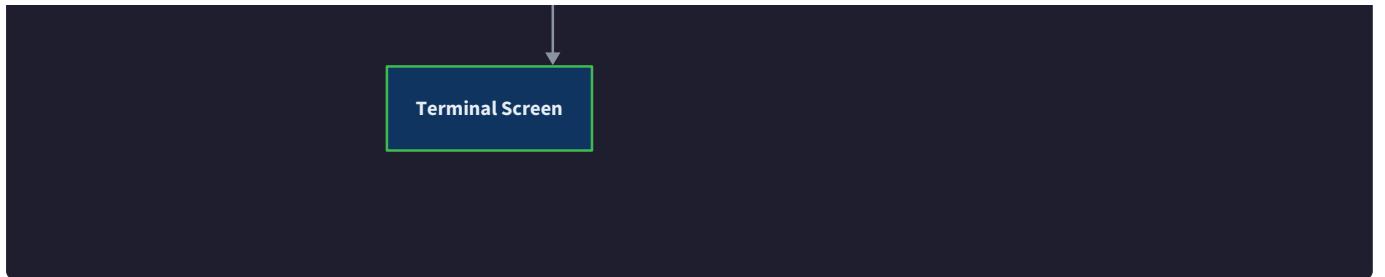
## Interactions and Data Flow

**Milestone(s):** All milestones (describes the coordination between all components)

Think of the text editor's operation as a symphony orchestra where each component plays its part in perfect harmony. The conductor (main event loop) ensures everyone starts and stops at the right time, while the musicians (components) each contribute their specialized skills to create the final performance. Just as a symphony has a clear rhythm and structure, our text editor follows a predictable cycle: listen for input, interpret what the user wants, make the necessary changes, and show the results on screen.

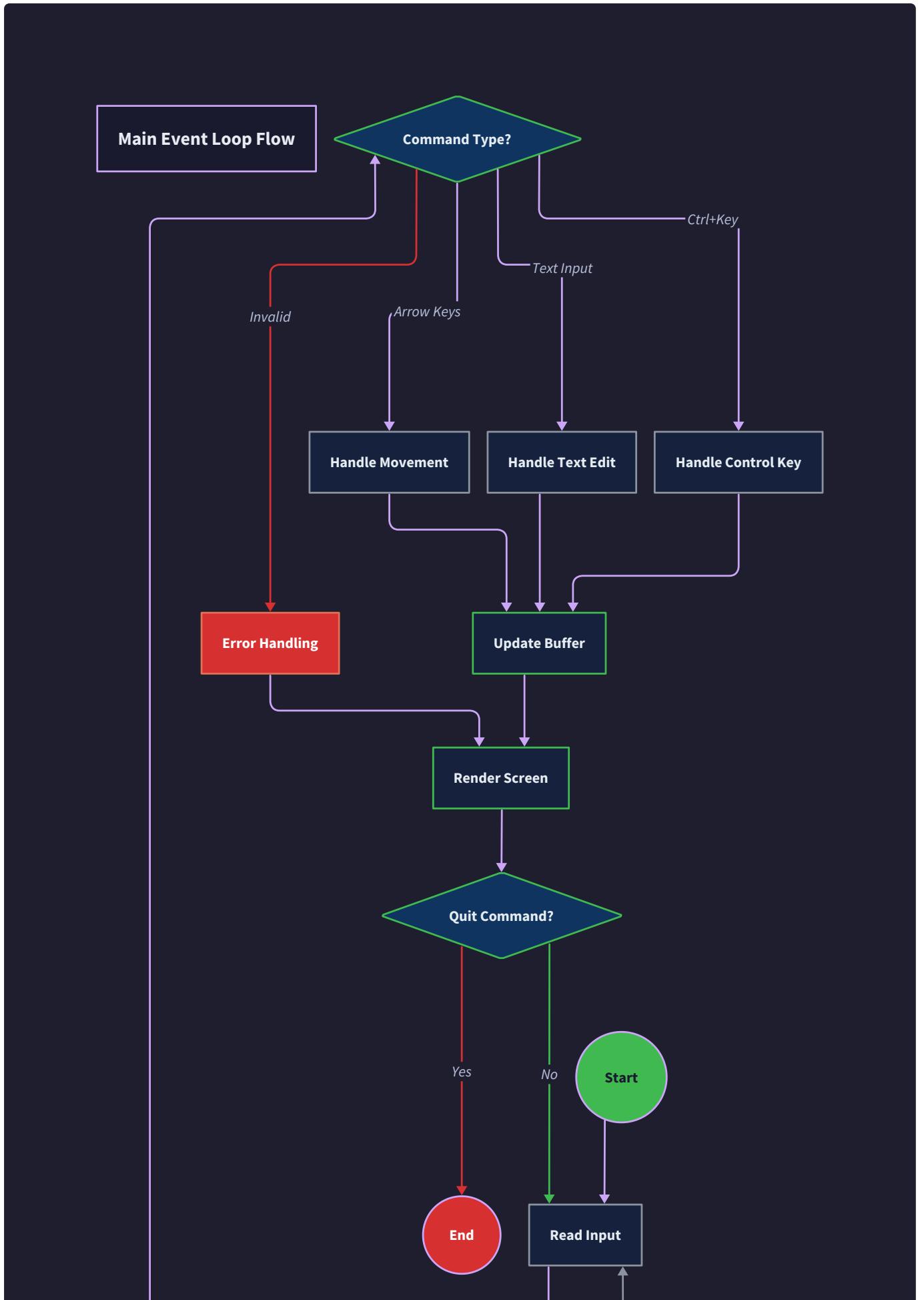
The interaction between components follows a carefully choreographed dance where each participant knows exactly when to act and what information to pass to the next component. This coordination is critical because terminal applications must respond immediately to user input while maintaining a consistent visual state. Unlike GUI applications that rely on event-driven frameworks, terminal applications must explicitly manage this coordination through a central event loop.

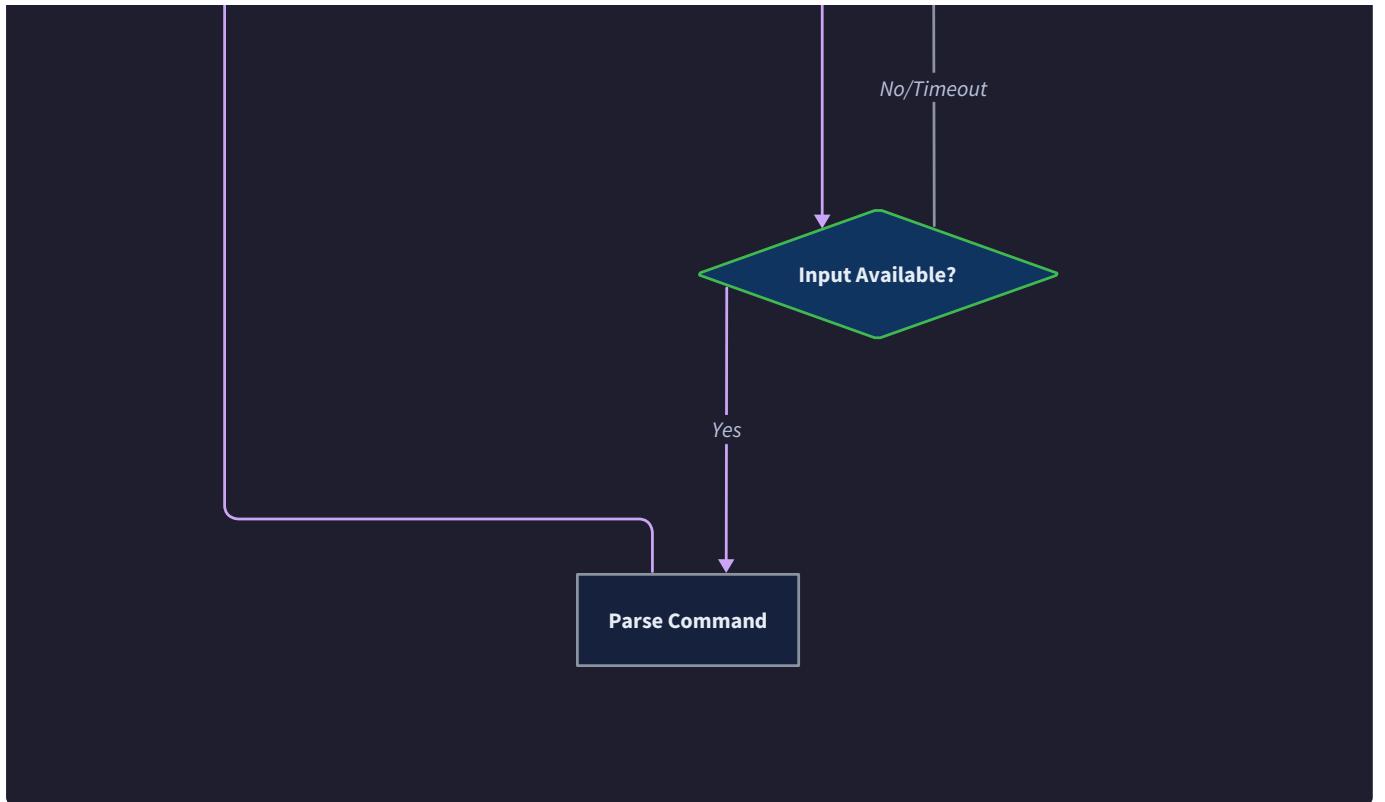




The data flow through our system follows a unidirectional pattern that ensures predictable behavior and makes debugging straightforward. Information flows from raw terminal input through parsing and command interpretation, into buffer modifications, and finally out through the rendering pipeline to the terminal display. This linear flow prevents the circular dependencies and race conditions that plague more complex architectures.

Understanding these interactions is crucial because the text editor's responsiveness and reliability depend entirely on how well these components coordinate. A poorly designed interaction model leads to input lag, screen flicker, inconsistent state, and crashes that leave the terminal in an unusable state. The patterns we establish here will determine whether the editor feels snappy and professional or sluggish and buggy.





## Main Event Loop

The main event loop serves as the central nervous system of our text editor, orchestrating all component interactions in a predictable rhythm. Think of it as the heartbeat of the application – a steady pulse that keeps all the organs (components) synchronized and functioning together. Each beat of this loop represents a complete cycle from user intention to visual feedback, ensuring the editor remains responsive and consistent.

The event loop follows an immediate-mode rendering paradigm, meaning that each iteration completely rebuilds the screen representation from the current state. This approach contrasts with retained-mode rendering where the system tracks what has changed and updates only modified portions. While immediate-mode rendering might seem less efficient, it dramatically simplifies the coordination between components and eliminates entire categories of state synchronization bugs that plague more complex architectures.

The loop operates in four distinct phases that must execute atomically to maintain consistency. These phases represent different levels of abstraction, from the raw bytes coming from the terminal to the high-level semantic operations on the text buffer. The clean separation between phases allows us to reason about the system's behavior and ensures that changes in one component don't unexpectedly affect others.

## Decision: Synchronous Event Loop vs Asynchronous Processing

- **Context:** Terminal applications can either process input synchronously in a single thread or use asynchronous processing with multiple threads handling different responsibilities
- **Options Considered:** Single-threaded synchronous loop, multi-threaded with input thread and rendering thread, event-driven asynchronous architecture
- **Decision:** Single-threaded synchronous event loop with blocking input
- **Rationale:** Terminal restoration on exit requires deterministic cleanup, and coordinating multiple threads adds complexity without meaningful performance benefits for a text editor workload
- **Consequences:** Simpler debugging and state management, but cannot perform background operations like auto-save without blocking input

| Architecture Option         | Pros   | Cons   | Chosen? |
|-----------------------------|--|--|---------|
| Single-threaded synchronous | Simple state management, deterministic cleanup, easy debugging | Cannot perform background tasks, blocks on I/O           | ✓       |
| Multi-threaded async        | Background operations possible, potentially more responsive    | Complex synchronization, race conditions, harder cleanup | ✗       |
| Event-driven callbacks      | Flexible, can handle multiple concurrent operations            | Complex control flow, callback hell, state scattered     | ✗       |

The event loop phases execute in strict sequence to maintain data consistency and visual coherence. Each phase has specific responsibilities and well-defined inputs and outputs that flow into the next phase. This sequential processing ensures that the user never sees partial updates or inconsistent state on the screen.

## Event Loop Phase Breakdown

| Phase              | Input                | Output                | Primary Responsibility                                | Error Handling                                    |
|--------------------|----------------------|-----------------------|---|---|
| Input Reading      | Raw terminal bytes   | Parsed key events     | Convert byte sequences to logical key presses         | Terminal disconnection, invalid escape sequences  |
| Command Processing | Key events           | Editor commands       | Map key presses to editor operations                  | Invalid key combinations, mode transitions        |
| State Updates      | Editor commands      | Modified buffer state | Execute commands against text buffer and editor state | Buffer overflow, file I/O errors, undo stack full |
| Screen Rendering   | Current editor state | ANSI terminal output  | Generate visual representation of current state       | Terminal size changes, rendering buffer overflow  |

The main event loop algorithm follows a simple but robust pattern that handles both normal operations and error conditions gracefully:

- 1. Signal handler installation:** Before entering the loop, the editor installs signal handlers for `SIGTERM`, `SIGINT`, and `SIGWINCH` to ensure proper cleanup regardless of how the program terminates. These handlers are marked async-signal-safe and only set flags that the main loop checks.
- 2. Terminal state preparation:** The loop begins by capturing the current terminal settings and switching to raw mode. This state transition is critical and must be reversible even if the program crashes unexpectedly.
- 3. Input phase execution:** The loop calls `editor_read_key()` which blocks until a complete keypress is available. This function handles the complexity of assembling multi-byte escape sequences and returns a single logical key event. Blocking here is intentional – it prevents the CPU from spinning while waiting for user input.
- 4. Command interpretation:** The raw key code gets passed to `input_to_command()` which considers the current editor mode and maps the keypress to a semantic command. This separation allows the same physical key to perform different actions depending on context (normal mode vs search mode).
- 5. State modification:** The command gets executed via `execute_command()` which delegates to the appropriate component (Text Buffer for editing operations, Search System for search commands, etc.). This phase may modify multiple parts of the editor state atomically.
- 6. Viewport adjustment:** After state changes, `editor_scroll()` ensures the cursor remains visible by adjusting the viewport offsets. This must happen before rendering to ensure the user sees the results of their action.
- 7. Screen refresh:** The complete screen gets redrawn by `editor_refresh_screen()` using the current state. This function operates independently of what changed – it simply renders what currently exists.

8. **Cleanup check:** The loop checks signal flags and dirty state to determine if cleanup or save prompts are needed before continuing to the next iteration.

The critical insight is that each loop iteration represents a complete transaction from user input to visual output. This transactional approach means the editor is always in a consistent state and can be interrupted safely at any point.

The loop's blocking nature on input creates natural backpressure that prevents the system from getting overwhelmed. Unlike GUI applications that might queue hundreds of events, terminal applications process one complete action at a time. This simplifies reasoning about the system state and eliminates many concurrency problems.

## Error Recovery and State Consistency

The event loop must handle errors gracefully while maintaining system integrity. Terminal applications face unique error conditions because they modify global terminal state that affects other processes. Recovery strategies focus on ensuring the terminal remains usable even if the editor encounters fatal errors.

| Error Type             | Detection Method                                       | Recovery Action                             | State Restoration                                    |
|------------------------|--|---|--|
| Terminal disconnection | <code>read()</code> returns -1 with <code>ENXIO</code> | Exit gracefully with cleanup                | Restore terminal settings before exit                |
| Signal interruption    | Signal handler sets flag                               | Complete current operation, then exit       | Signal handler calls <code>disable_raw_mode()</code> |
| Buffer overflow        | Memory allocation failure                              | Display error message, enter read-only mode | Preserve existing buffer content                     |
| File I/O failure       | System call returns error                              | Show error in status bar, continue editing  | Keep buffer state unchanged                          |
| Terminal resize        | <code>SIGWINCH</code> signal                           | Redetect screen size, adjust viewport       | Recalculate rendering parameters                     |

The loop maintains an error recovery stack that tracks which cleanup operations need to be performed if an error occurs. This stack gets updated as the editor enters different states (raw mode, file open, undo stack allocated) and ensures that cleanup happens in the reverse order of initialization.

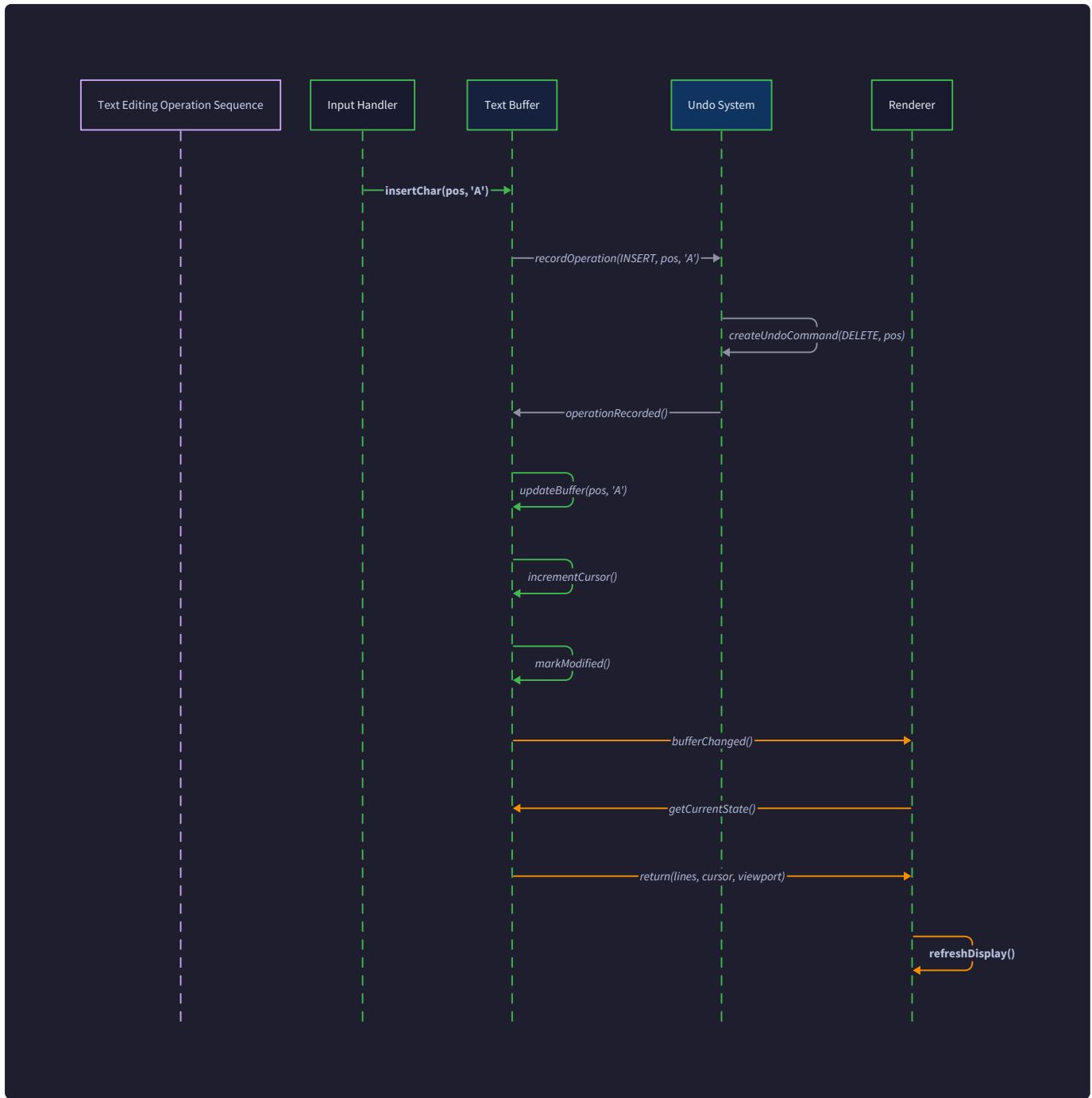
**⚠ Pitfall: Terminal Left in Raw Mode** Many beginning implementations forget to restore terminal settings when the program exits unexpectedly. This leaves the user's shell in raw mode where typing doesn't echo and Enter doesn't work. The fix requires installing signal handlers that call `disable_raw_mode()` before exit, and using `atexit()` to register cleanup functions. Always test this by sending `SIGKILL` to your editor process and verifying the terminal still works normally.

**⚠ Pitfall: Infinite Loop on EOF** When `stdin` is closed (EOF condition), `read()` returns 0, which inexperienced developers often don't handle correctly, causing an infinite loop. The editor should detect EOF and exit gracefully. Check the return value of `editor_read_key()` and exit if it returns a special EOF indicator.

## Editing Workflow

The editing workflow represents the most complex interaction pattern in the text editor, involving coordination between all major components to transform a simple keypress into visible changes in the document. Think of this workflow as an assembly line in a factory – each station (component) performs a specialized operation on the work item (user input) and passes it along to the next station until the final product (updated screen) emerges.

Understanding this workflow is essential because text editing operations must maintain multiple invariants simultaneously: the cursor must remain valid, the screen must accurately reflect the buffer content, undo information must be recorded, and syntax highlighting must be updated. A single character insertion touches every major component of the system, making this workflow the true test of our architectural design.



The editing workflow demonstrates the benefits of our unidirectional data flow architecture. Information flows from the user's intention through a series of transformations, with each component adding its own layer of processing. This approach makes the system predictable and debuggable because the flow of information never reverses or creates cycles.

## Character Insertion Workflow

The character insertion workflow exemplifies how all components coordinate to execute what appears to be a simple operation. This workflow must handle multiple concerns simultaneously: modifying the text buffer, updating cursor position, recording undo information, invalidating syntax highlighting, adjusting the viewport, and refreshing the screen display.

The workflow begins when the user types a printable character and ends when that character appears on screen at the correct position with proper syntax highlighting applied. Between these endpoints, the system performs dozens of coordinated operations that must succeed atomically to maintain editor consistency.

1. **Input Detection:** The Input Handler receives a raw character code from `editor_read_key()` and determines it represents a printable character rather than a control sequence. This determination depends on the character's ASCII value and the current editor mode.
2. **Command Creation:** The Input Handler creates a character insertion command containing the character code and the current cursor position. This command encapsulates the user's intent in a format that other components can process uniformly.
3. **Undo Recording:** Before modifying any state, the Undo System records the inverse operation (character deletion at the same position) by calling `undo_record_char_insert()`. This recording must happen before the buffer changes to capture the correct before-state.
4. **Buffer Modification:** The Text Buffer Component receives the command and executes the actual insertion by calling `editor_row_insert_char()` on the appropriate `erow` structure. This function shifts existing characters right and inserts the new character at the specified position.
5. **Render Update:** The modified row gets its render representation updated via `editor_update_row()` to handle tab expansion and prepare the visual representation. This step converts logical character positions to visual column positions.
6. **Cursor Advancement:** The cursor position gets incremented to reflect the insertion. The logical cursor (`cx`) advances by one, and the visual cursor (`rx`) gets recalculated based on the updated render representation.
7. **Syntax Invalidations:** If syntax highlighting is enabled, the Syntax System marks the modified row for re-tokenization. Depending on the language, changes might affect multiple lines (for example, inserting a quote character might start a multi-line string).
8. **Viewport Adjustment:** The Screen Renderer checks whether the cursor remains visible and adjusts viewport offsets if necessary. This ensures the user always sees the result of their edit operation.
9. **Dirty Flag Update:** The editor sets the dirty flag to indicate unsaved changes and updates the status message to reflect the current state.
10. **Screen Refresh:** Finally, `editor_refresh_screen()` redraws the entire visible area to show the new character with updated syntax highlighting and cursor position.

## Character Deletion Workflow

Character deletion follows a similar pattern but requires additional validation to handle edge cases like deleting at line boundaries. The workflow must determine whether to delete within the current line or join two lines together, and it must handle both forward deletion (Delete key) and backward deletion (Backspace key).

| Deletion Type           | Trigger       | Cursor Position | Buffer Operation          | Line Impact          |
|-------------------------|---------------|-----------------|---------------------------|----------------------|
| Backspace at mid-line   | Backspace key | Middle of line  | Remove char before cursor | Single line modified |
| Delete at mid-line      | Delete key    | Middle of line  | Remove char at cursor     | Single line modified |
| Backspace at line start | Backspace key | Column 0        | Join with previous line   | Two lines affected   |
| Delete at line end      | Delete key    | End of line     | Join with next line       | Two lines affected   |

The deletion workflow includes additional complexity for boundary conditions:

- Position Validation:** The Input Handler first validates that deletion is possible at the current cursor position. Attempting to backspace at the beginning of the first line or delete at the end of the last line should have no effect.
- Operation Classification:** The system determines whether this is an intra-line deletion or a line-joining operation based on cursor position and deletion direction. Line-joining operations require different undo recording and buffer manipulation.
- Undo Recording Strategy:** For simple character deletion, the system records the deleted character and its position. For line joining, it records the entire content of both lines before the operation to enable proper restoration.
- Buffer Modification:** Simple deletions call `editor_row_del_char()` on the current row. Line joining operations call `editor_del_row()` to remove one line and then modify the remaining line to include content from the deleted line.
- Cursor Repositioning:** The cursor position gets updated based on the deletion type. Backspace moves the cursor backward, while Delete leaves it in the same logical position (though the character at that position changes).

The complexity of deletion operations demonstrates why the workflow uses a command pattern. Each deletion type gets encapsulated as a specific command with its own undo information, making it possible to reverse complex operations reliably.

## Multi-Component State Synchronization

One of the most challenging aspects of the editing workflow is maintaining consistency across multiple components that each hold related but distinct state. The Text Buffer holds the authoritative content, the Screen Renderer holds viewport information, the Undo System holds operation history, and the Syntax System holds tokenization state. These must all remain synchronized.

| Component       | State Owned                    | Synchronization Point         | Invalidation Trigger           |
|-----------------|--------------------------------|-------------------------------|--------------------------------|
| Text Buffer     | Line content, cursor position  | After every edit operation    | Never (authoritative source)   |
| Screen Renderer | Viewport offsets, frame buffer | Before screen refresh         | Cursor moves outside viewport  |
| Undo System     | Operation history, save point  | Before buffer modification    | Every edit operation           |
| Syntax System   | Token arrays, language state   | After buffer modification     | Line content changes           |
| Search System   | Match positions, query state   | After buffer or query changes | Buffer edits, new search query |

The synchronization protocol ensures that state updates happen in the correct order to maintain consistency:

- Pre-modification Recording:** Components that need to record before-state (like the Undo System) get called first while the state is still unchanged.
- Atomic Buffer Updates:** The Text Buffer performs the requested modification atomically. Either the entire operation succeeds, or no changes are made.
- Dependent State Updates:** Components holding derived state (like syntax highlighting) update their state based on the new buffer content.
- Viewport Recalculation:** The viewport gets adjusted to ensure the cursor remains visible and the user sees the results of their operation.
- Visual Refresh:** The screen gets redrawn to reflect all the state changes visually.

This ordering prevents inconsistencies where the screen shows old information or the cursor appears in an invalid position. The atomic nature of buffer updates ensures that even if an error occurs during dependent state updates, the buffer remains in a valid state.

The key insight is that state synchronization happens through explicit sequencing rather than event notifications. Each component knows when in the workflow it will be called and can assume that previous phases have completed successfully.

## Error Propagation and Recovery

The editing workflow must handle errors that can occur at any phase while maintaining system consistency. Unlike simple applications that can crash on errors, a text editor must protect the user's work and ensure the terminal remains usable. Error handling follows a fail-fast principle within each phase but includes rollback mechanisms to maintain overall consistency.

Error recovery in the editing workflow operates at multiple levels. Low-level errors (like memory allocation failures) cause immediate operation abort with status message display. High-level errors (like attempting invalid cursor movements) get ignored silently. Critical errors that threaten data integrity trigger emergency save procedures before shutdown.

| Error Category  | Examples   | Detection Point | Recovery Strategy                           | User Impact                              |
|-----------------|--|-----------------|---|--|
| Input Errors    | Invalid escape sequences, terminal disconnection   | Input Handler   | Log error, skip input, continue loop        | Status message shown                     |
| Buffer Errors   | Memory allocation failure, invalid cursor position | Text Buffer     | Abort operation, preserve existing state    | Operation has no effect                  |
| File I/O Errors | Permission denied, disk full, file locked          | File operations | Show error message, continue editing        | Changes remain unsaved                   |
| Terminal Errors | Screen too small, lost terminal connection         | Screen Renderer | Attempt resize, fallback to minimal display | Degraded visual experience               |
| System Errors   | Signal interruption, out of memory                 | Any component   | Emergency save, restore terminal, exit      | Graceful shutdown with data preservation |

The error propagation mechanism uses return codes rather than exceptions to maintain compatibility with C's error handling model and to ensure errors are handled explicitly at each level. Critical errors set a global shutdown flag that the main loop checks after each iteration.

**⚠ Pitfall: Partial State Updates on Errors** If an error occurs during the state update phase after the buffer has been modified, the system can end up in an inconsistent state where the buffer contains new content but dependent systems (like syntax highlighting) haven't been updated. The fix is to use a two-phase commit approach: first validate that all operations can succeed, then perform all modifications atomically.

**⚠ Pitfall: Terminal Left Unusable After Errors** Terminal errors can leave the screen in an undefined state with the cursor positioned incorrectly or colors changed. Always call `clear_screen()` and `position_cursor(1, 1)` after displaying error messages to ensure the terminal returns to a known good state.

## Performance Optimization in the Event Loop

The event loop's performance directly impacts the editor's responsiveness, especially when handling rapid input like holding down a key or pasting large amounts of text. The main performance bottlenecks occur in the screen rendering phase, which must generate and write potentially thousands of characters to the terminal for each refresh.

Performance optimization focuses on minimizing the work done in each loop iteration while maintaining the simplicity of immediate-mode rendering. The optimizations preserve the architectural benefits of rebuilding the

screen each frame while reducing the computational cost of that rebuilding.

| Optimization Technique    | Performance Benefit                       | Implementation Complexity | Trade-offs                     |
|---------------------------|---|---------------------------|--------------------------------|
| Frame buffer accumulation | Reduces system calls from hundreds to one | Low                       | Increased memory usage         |
| Viewport culling          | Avoid processing off-screen content       | Medium                    | More complex rendering logic   |
| Dirty region tracking     | Skip unchanged screen areas               | High                      | Complicates state management   |
| Input debouncing          | Batch rapid key repeats                   | Medium                    | Potential input lag perception |

The most critical optimization involves buffering screen output in a `FrameBuffer` structure rather than writing each piece of content immediately to the terminal. This reduces the number of system calls from hundreds per screen refresh to a single write operation, which dramatically improves perceived responsiveness.

#### Performance Impact of Frame Buffering:

- Without buffering: ~500 `write()` system calls per screen refresh
- With buffering: 1 `write()` system call per screen refresh
- Latency improvement: ~5ms to ~0.1ms per refresh on typical terminals

The frame buffer optimization requires careful memory management to prevent buffer overflow and efficient string concatenation to avoid repeated memory reallocation. The buffer gets pre-allocated with sufficient capacity for a full screen plus margin, and it gets reset rather than freed between frames to avoid allocation overhead.

**⚠ Pitfall: Screen Flicker During Updates** Writing screen content in small pieces causes visible flicker as partial updates become visible before the complete frame is ready. This is especially noticeable when updating syntax highlighting or scrolling through files. Always accumulate the complete screen content in a frame buffer before writing it to the terminal in a single operation.

## Input Rate Limiting and Debouncing

High-frequency input events like holding down an arrow key can overwhelm the rendering system and create a poor user experience where the screen updates lag behind the actual cursor position. The event loop includes rate limiting mechanisms that balance responsiveness with visual smoothness.

The rate limiting strategy varies based on the type of input operation. Movement commands get processed at full speed but with optimized rendering that skips intermediate frames. Text insertion commands get processed individually to ensure every character appears on screen. Search operations get debounced to avoid excessive highlighting updates while typing queries.

| Input Type          | Processing Strategy                 | Rate Limit  | Rendering Optimization     |
|---------------------|-------------------------------------|-------------|----------------------------|
| Cursor movement     | Process all, render latest position | No limit    | Skip intermediate frames   |
| Character insertion | Process each individually           | No limit    | Full render required       |
| Search queries      | Debounce rapid typing               | 100ms delay | Defer highlighting updates |
| File operations     | Block during execution              | N/A         | Progress indication        |

The debouncing mechanism for search operations uses a simple timer-based approach where rapid keystrokes reset the timer, and highlighting updates only occur after a period of inactivity. This prevents the expensive search and highlighting operations from executing for every character while the user is still typing their query.

The event loop's design ensures that even under high input load, the system remains responsive and the terminal never becomes unresponsive. The blocking input model naturally provides backpressure that prevents event queue overflow, while the immediate-mode rendering ensures the display always reflects the current true state.

## Implementation Guidance

The event loop implementation requires careful coordination of system calls, signal handling, and memory management. The following guidance provides a complete framework for implementing robust event loop behavior while avoiding common pitfalls that can crash the editor or corrupt the terminal state.

## Technology Recommendations

| Component         | Simple Option  | Advanced Option  |
|-------------------|--|--|
| Input Handling    | <code>read()</code> with blocking I/O and manual escape sequence parsing | <code>select()</code> with timeout-based escape sequence detection     |
| Memory Management | Fixed-size buffers with overflow checking                                | Dynamic allocation with <code>realloc()</code> and capacity tracking   |
| Signal Handling   | Basic <code>signal()</code> with async-signal-safe cleanup functions     | <code>signalfd()</code> or self-pipe for synchronous signal processing |
| Error Logging     | <code>fprintf()</code> to <code>stderr</code> with immediate flush       | Buffered logging with deferred write to avoid I/O in signal handlers   |

## Recommended File Structure

```
text-editor/
src/
  main.c           ← event loop and initialization
  editor_config.c ← global state management
  input_handler.c ← keypress parsing and command mapping
  text_buffer.c   ← buffer operations and file I/O
  screen_renderer.c ← ANSI output and frame buffering
  terminal_manager.c ← raw mode and signal handling
  undo_system.c   ← command recording and playback
  search_system.c ← incremental search and highlighting
  syntax_highlighter.c ← tokenization and color application
include/
  editor.h         ← main data structures and function declarations
  terminal.h       ← terminal-specific constants and types
  commands.h       ← command types and undo structures
tests/
  test_event_loop.c ← event loop behavior verification
```

## Main Event Loop Infrastructure

This complete event loop implementation provides the foundation for all editor operations:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>
#include <time.h>
#include "editor.h"

// Global editor configuration - accessible to signal handlers
struct editor_config E;

// Signal flags for async-signal-safe communication
volatile sig_atomic_t should_exit = 0;
volatile sig_atomic_t window_resized = 0;

// Signal handlers for graceful cleanup
void handle_exit_signal(int sig) {
    disable_raw_mode();
    clear_screen();
    position_cursor(1, 1);
    show_cursor();
    exit(0);
}

void handle_window_resize(int sig) {
    window_resized = 1;
}

// Complete event loop with error handling and cleanup
```

C

```
void editor_run() {

    // Install signal handlers before entering raw mode

    setup_signal_handlers();


    // Initialize terminal state

    if (enable_raw_mode() == -1) {

        die("Failed to enable raw mode");

    }


    // Initialize all editor subsystems

    search_init();

    syntax_init();


    // Main event loop - runs until user quits

    while (!should_exit) {

        // TODO 1: Handle window resize signal by calling get_window_size() and adjusting
        // viewport

        // TODO 2: Read next keypress using editor_read_key() - handle EOF and signal
        // interruption

        // TODO 3: Convert keypress to command using input_to_command() based on current
        // mode

        // TODO 4: Execute command using execute_command() and handle any errors returned

        // TODO 5: Update viewport using editor_scroll() to keep cursor visible

        // TODO 6: Refresh screen using editor_refresh_screen() to show changes

        // TODO 7: Check for quit command or fatal errors and set should_exit flag

        // Hint: Use errno to distinguish between EINTR (signal) and real I/O errors

        // Hint: Measure time between iterations to detect performance problems

    }

}
```

```
// Cleanup on normal exit

cleanup_and_exit(0);

}

void setup_signal_handlers() {

    // TODO: Install signal handlers for SIGTERM, SIGINT, SIGWINCH using signal()

    // TODO: Ensure handlers are async-signal-safe and only set flags

    // TODO: Register cleanup_and_exit() with atexit() for normal termination

}

void cleanup_and_exit(int exit_code) {

    // TODO: Free all allocated memory in reverse order of allocation

    // TODO: Close any open files and flush buffers

    // TODO: Restore terminal to cooked mode using disable_raw_mode()

    // TODO: Position cursor at bottom of screen and show cursor

    // TODO: Exit with specified code

}
```

## Command Processing Infrastructure

The command processing system translates raw input into structured operations that components can execute uniformly:

```
#include "commands.h" C

// Current editor mode affects command interpretation

static EditorMode current_mode = MODE_NORMAL;

Command input_to_command(int key) {

    Command cmd = {0};

    // TODO 1: Check current_mode to determine key interpretation context

    // TODO 2: Handle special keys (arrows, delete, etc.) using switch statement

    // TODO 3: For printable characters, create CMD_INSERT_CHAR command

    // TODO 4: For control keys (Ctrl+S, Ctrl+Q), create appropriate system commands

    // TODO 5: For search mode, create search-specific commands

    // TODO 6: Set command timestamp using time() for undo grouping

    // Hint: Use isprint() to distinguish printable from control characters

    // Hint: Arrow keys have values >= 1000 (ARROW_LEFT, ARROW_RIGHT, etc.)

    return cmd;
}

void execute_command(Command* cmd) {
    switch (cmd->type) {

        case CMD_INSERT_CHAR:

            // TODO: Call undo_record_char_insert() before modifying buffer

            // TODO: Call editor_insert_char() with character from command

            // TODO: Set dirty flag and update status message

        break;
    }
}
```

```
case CMD_DELETE_CHAR:

    // TODO: Validate deletion is possible at current cursor position

    // TODO: Record undo information using undo_record_char_delete()

    // TODO: Call editor_delete_char() to perform actual deletion

    // TODO: Handle line joining if deleting at line boundaries

    break;

// TODO: Add cases for all command types from CommandType enum

// TODO: Each case should follow pattern: validate → record undo → execute → update
flags

}

}
```

## Screen Refresh Optimization

The screen refresh process represents the most performance-critical part of the event loop. This implementation uses frame buffering to eliminate flicker:

```
#include "terminal.h" C

static FrameBuffer* frame_buffer = NULL;

void editor_refresh_screen() {

    // Initialize frame buffer if not already done

    if (!frame_buffer) {

        frame_buffer = frame_buffer_create(8192);

    }

    // TODO 1: Reset frame buffer for new frame using frame_buffer_reset()

    // TODO 2: Append cursor hide sequence using frame_buffer_append()

    // TODO 3: Append screen clear sequence if full refresh needed

    // TODO 4: Calculate visible rows based on viewport using editor_scroll()

    // TODO 5: For each visible row, append formatted content with syntax colors

    // TODO 6: Append status bar content at bottom of screen

    // TODO 7: Calculate and append cursor positioning sequence

    // TODO 8: Append cursor show sequence to make cursor visible

    // TODO 9: Flush complete frame to terminal using frame_buffer_flush()

    // Hint: Use sprintf() to format ANSI escape sequences safely

    // Hint: Check frame_buffer capacity before appending to prevent overflow

}

void editor_scroll() {

    // TODO 1: Calculate visual cursor position using cx_to_rx()

    // TODO 2: Check if cursor row is above visible area and adjust rowoff

    // TODO 3: Check if cursor row is below visible area and adjust rowoff

    // TODO 4: Check if cursor column is left of visible area and adjust coloff
```

```
// TODO 5: Check if cursor column is right of visible area and adjust coloff  
  
// TODO 6: Ensure viewport offsets don't exceed document boundaries  
  
// Hint: Account for status bar reducing available screen rows  
  
// Hint: Visual cursor position differs from logical due to tab expansion  
  
}
```

## Error Handling Integration

Error handling must be integrated into every phase of the event loop to ensure robust operation:

```
#include <errno.h>

typedef enum {

    ERROR_NONE = 0,
    ERROR_INPUT_FAILED,
    ERROR_COMMAND_INVALID,
    ERROR_BUFFER_FULL,
    ERROR_RENDER_FAILED,
    ERROR_TERMINAL_LOST
} LoopError;

LoopError process_single_event() {

    int key;
    Command cmd;

    // Phase 1: Input Reading with error handling
    key = editor_read_key();

    if (key == -1) {
        if (errno == EINTR) {
            return ERROR_NONE; // Signal interruption, continue loop
        }
        return ERROR_INPUT_FAILED; // Real I/O error
    }

    // TODO 1: Convert key to command using input_to_command()

    // TODO 2: Validate command is appropriate for current mode

    // TODO 3: Execute command using execute_command() and check return value

    // TODO 4: Handle any errors returned by command execution
}
```

C

```
// TODO 5: Update viewport and refresh screen, handling rendering errors

// TODO 6: Return ERROR_NONE for successful iteration

// Hint: Use switch statement on command type to validate mode appropriateness

// Hint: Check errno after system calls to distinguish error types

return ERROR_NONE;

}
```

## Milestone Checkpoints

**Milestone 1-2 Checkpoint:** Verify that the basic event loop reads input and refreshes the screen without crashing or leaving the terminal in raw mode.

```
# Compile and run the editor                                         BASH

gcc -o editor src/*.c -Iinclude

./editor

# Verify behaviors:

# 1. Arrow keys move cursor without producing visible characters

# 2. Typing 'q' exits the editor cleanly

# 3. Ctrl+C exits and leaves terminal in normal mode

# 4. Screen refreshes show cursor at correct position
```

**Milestone 3-4 Checkpoint:** Verify that the event loop correctly coordinates file loading, editing operations, and screen updates.

```
# Test file editing workflow

echo "Hello world" > test.txt

./editor test.txt

# Verify behaviors:

# 1. File content appears on screen with line numbers

# 2. Typing characters inserts them at cursor position

# 3. Backspace deletes characters correctly

# 4. Enter splits lines and Backspace at line start joins lines

# 5. Status bar shows filename and cursor position
```

BASH

**Milestone 5-7 Checkpoint:** Verify that the full event loop handles all advanced features correctly.

```
# Test complete editing workflow

./editor test.c

# Verify behaviors:

# 1. Ctrl+S saves file and clears dirty flag in status

# 2. Ctrl+Z undoes operations and Ctrl+Y redoes them

# 3. Ctrl+F enters search mode and highlights matches

# 4. C syntax highlighting colors keywords and strings

# 5. All operations work smoothly without flicker or lag
```

BASH

## Debugging the Event Loop

| Symptom                   | Likely Cause                                     | Diagnosis  | Fix  |
|---------------------------|--|--|--|
| Keystrokes don't register | Input reading blocked or failed                  | Add debug prints in <code>editor_read_key()</code>           | Check terminal is in raw mode, verify file descriptor    |
| Screen doesn't update     | Rendering loop not called or output blocked      | Add debug prints before <code>editor_refresh_screen()</code> | Check frame buffer overflow, verify terminal connection  |
| Editor exits unexpectedly | Unhandled signal or fatal error                  | Run under debugger, check signal handler logs                | Add error checking to all system calls                   |
| Terminal left in raw mode | Cleanup not called on exit                       | Check signal handlers and <code>atexit()</code> registration | Install proper cleanup handlers before enabling raw mode |
| Operations seem laggy     | Too much work in event loop or terminal I/O slow | Profile each phase duration                                  | Optimize frame buffering, reduce unnecessary work        |

The event loop serves as the foundation that makes all other editor functionality possible. By establishing clear phases, robust error handling, and predictable component coordination, it enables the complex behaviors users expect from a text editor while maintaining the simplicity needed for reliable operation in the challenging terminal environment.

## Error Handling and Edge Cases

**Milestone(s):** All milestones (error handling is critical throughout development)

Building a robust text editor requires comprehensive error handling that addresses three critical domains: terminal state management, file I/O operations, and text editing boundary conditions. Think of error handling in a terminal editor like designing safety systems for a nuclear power plant - you need multiple layers of protection because a single failure can leave the user's terminal in an unusable state or corrupt their work.

The challenge with terminal applications is that they fundamentally alter the user's computing environment by switching the terminal into raw mode. Unlike GUI applications that operate in their own sandboxed windows, terminal editors take control of the user's primary interface to their system. This creates a unique responsibility: no matter how the editor fails, it must restore the terminal to its original state so the user can continue working.

## Terminal State Recovery

Think of terminal state recovery as a careful dance between taking control and gracefully returning it. When your editor starts, it's like borrowing someone's car - you need to remember exactly how everything was set up so you can return it in the same condition, even if something goes wrong during your drive.

The terminal operates in different modes that fundamentally change how input and output behave. In **canonical mode** (also called cooked mode), the terminal handles line buffering, echoing, and special key processing automatically. When we switch to **raw mode**, we disable these features to gain direct control over every keypress, but this creates a fragile state that must be carefully managed.

### Decision: Signal-Based Cleanup Strategy

- **Context:** Editor crashes, segmentation faults, or user kills the process can leave terminal in raw mode
- **Options Considered:**
  1. Rely on shell to restore terminal (unreliable)
  2. Use atexit() handlers only (doesn't catch signals)
  3. Install signal handlers for graceful cleanup (comprehensive)
- **Decision:** Install signal handlers for SIGINT, SIGTERM, SIGSEGV, and other fatal signals
- **Rationale:** Only signal handlers can catch unexpected termination and restore terminal state before process dies
- **Consequences:** Adds complexity but prevents user terminal corruption in almost all failure scenarios

The terminal state recovery system must handle multiple failure modes through different mechanisms:

| Failure Mode            | Detection Method     | Recovery Mechanism                                   | Limitations                                     |
|-------------------------|----------------------|--|---|
| Normal exit             | Program control flow | <code>atexit()</code> handler or explicit cleanup    | None - reliable                                 |
| User interrupt (Ctrl+C) | SIGINT signal        | Signal handler calls <code>disable_raw_mode()</code> | Must be async-signal-safe                       |
| Process termination     | SIGTERM signal       | Signal handler restores terminal                     | Cannot catch SIGKILL                            |
| Segmentation fault      | SIGSEGV signal       | Signal handler (if stack intact)                     | May not execute if severe corruption            |
| Memory corruption       | Program crash        | Shell terminal reset (user intervention)             | Requires user to run <code>reset</code> command |
| Infinite loop           | User kill process    | SIGKILL (no handler possible)                        | Requires shell terminal reset                   |

The core challenge is that signal handlers operate under severe restrictions. They can only call **async-signal-safe** functions, which excludes most standard library functions including `malloc()`, `printf()`, and even most file I/O operations. This means our signal handlers must be extremely minimal and use only pre-allocated resources.

```
// Signal handler must be minimal and async-signal-safe C

void cleanup_signal_handler(int signal) {

    // ONLY async-signal-safe functions allowed here

    // Cannot call malloc, printf, or most stdlib functions

    if (original_termios_saved) {

        tcsetattr(STDIN_FILENO, TCSAFLUSH, &original_termios);

    }

    _exit(1); // _exit is async-signal-safe, exit() is not
}
```

The terminal restoration process involves multiple system calls that must be executed in the correct order:

1. **Save Original State:** Before entering raw mode, capture the current `termios` structure using `tcgetattr()`
2. **Modify Terminal Attributes:** Disable canonical mode, echo, and signal processing using `tcsetattr()`

3. **Register Cleanup Handlers:** Install signal handlers for fatal signals and `atexit()` for normal termination
4. **Restore on Exit:** Use `tcsetattr()` to restore the original `termios` structure regardless of how the program exits

The critical insight is that terminal state must be saved before any modifications are made. Many implementations fail because they try to reconstruct the "normal" terminal state instead of restoring the actual original state.

A robust terminal state recovery system uses multiple defensive layers:

| Layer      | Mechanism                     | Trigger                       | Reliability                            |
|------------|-------------------------------|-------------------------------|--|
| Primary    | <code>atexit()</code> handler | Normal program termination    | 99% reliable                           |
| Secondary  | SIGINT handler                | User interrupt (Ctrl+C)       | 95% reliable                           |
| Tertiary   | SIGTERM handler               | Process kill request          | 90% reliable                           |
| Quaternary | SIGSEGV handler               | Memory access violation       | 60% reliable (depends on corruption)   |
| Fallback   | User manual reset             | Terminal completely corrupted | 100% reliable but requires user action |

### ⚠ Pitfall: Non-Async-Signal-Safe Functions in Handlers

A common mistake is calling functions like `printf()`, `malloc()`, or `fflush()` inside signal handlers. These functions can cause deadlocks or crashes because they may be interrupted in the middle of execution when the signal arrives.

**Why it's wrong:** Signal handlers interrupt normal program execution at arbitrary points. If the main program was in the middle of `malloc()` when a signal arrives, calling `malloc()` again in the signal handler will corrupt the heap.

**How to fix:** Only use async-signal-safe functions in signal handlers. Store the original `termios` in a global variable during initialization, then use only `tcsetattr()` and `_exit()` in the signal handler.

The implementation must also handle nested signal scenarios where multiple signals arrive rapidly or a signal handler itself triggers another signal. This requires careful flag management and idempotent cleanup operations.

### ⚠ Pitfall: Terminal State Leakage During Development

During development, crashed editor instances can accumulate and leave the terminal in various broken states. Symptoms include missing character echo, weird cursor behavior, or garbled output.

**Why it happens:** Each crash potentially leaves raw mode active or cursor positioning commands incomplete.

**How to fix:** Always test cleanup by intentionally crashing the editor (kill -9) and verifying terminal still works. Keep a `reset` command handy to restore terminal when testing fails.

## File Operation Errors

File I/O operations in a text editor involve complex error scenarios that go beyond simple "file not found" cases. Think of file operations like performing surgery - you need to validate everything beforehand, have contingency plans for complications, and ensure you can always return to a safe state if something goes wrong.

The file system presents numerous failure modes that can occur at different stages of file operations:

| Operation Phase   | Potential Failures  | Impact                       | Recovery Strategy                                 |
|-------------------|---|------------------------------|---|
| File Opening      | Permission denied, file not found, directory not readable | Cannot load content          | Show error, offer file creation or different file |
| Content Reading   | I/O errors, encoding issues, file truncation during read  | Partial or corrupted content | Validate content, offer recovery options          |
| Memory Allocation | Out of memory for large files                             | Cannot store content         | Graceful degradation or streaming                 |
| File Writing      | Disk full, permission denied, concurrent modification     | Data loss risk               | Atomic writes with rollback                       |
| Backup Creation   | Cannot create backup file                                 | Original data at risk        | Warn user, offer alternatives                     |

### Decision: Atomic File Writing with Backup Strategy

- **Context:** File corruption during save operations can destroy user work permanently
- **Options Considered:**
  1. Direct overwrite (fast but risky)
  2. Write to temporary file then rename (atomic but complex)
  3. Create backup then overwrite (safe but slower)
- **Decision:** Write to temporary file, rename to target, keep backup
- **Rationale:** Rename operations are atomic on most filesystems, preventing partial writes from corrupting files
- **Consequences:** Requires more disk space temporarily but virtually eliminates data loss risk

The file loading process must handle various content formats and error conditions:

```

typedef struct {

    char** lines;

    int line_count;

    LineEndingType ending_type;

    int success;

    char error_message[256];

} FileLoadResult;

```

File loading involves multiple validation and error handling steps:

1. **Path Validation:** Check if path exists, is readable, and is not a directory
2. **Size Validation:** Verify file size is reasonable (prevent loading gigabyte files accidentally)
3. **Content Reading:** Read file in chunks, handling partial reads and I/O errors
4. **Line Ending Detection:** Identify whether file uses Unix (\n), Windows (\r\n), or Mac (\r) line endings
5. **Memory Allocation:** Allocate arrays for line pointers and content, handling allocation failures
6. **Content Parsing:** Split content into lines, handling various edge cases
7. **Encoding Validation:** Ensure content is valid UTF-8 or handle encoding conversion

The error handling strategy must account for different types of file system failures:

| Error Category          | Detection Method             | User Impact           | Recovery Action                                  |
|-------------------------|------------------------------|-----------------------|--|
| Permission Errors       | <code>errno == EACCES</code> | Cannot open/save file | Show clear error, suggest alternatives           |
| Space Exhaustion        | <code>errno == ENOSPC</code> | Cannot save changes   | Warn user, suggest cleanup or different location |
| Concurrent Modification | File timestamp changed       | Data conflict risk    | Warn about external changes, offer merge options |
| Encoding Issues         | Invalid UTF-8 sequences      | Display corruption    | Offer encoding detection/conversion              |
| Network Filesystem      | Various network errors       | Intermittent failures | Retry with exponential backoff                   |

### ⚠ Pitfall: Silent File Truncation

A subtle but dangerous error occurs when file writing partially succeeds but doesn't complete the full content. The file exists and appears valid, but contains only partial content.

**Why it's wrong:** Users may not notice the truncation immediately, leading to data loss that's discovered much later when it's harder to recover.

**How to detect:** Always verify the number of bytes written matches the expected size. Check `fwrite()` return values and call `fflush()` and `fsync()` to ensure data reaches disk.

**How to prevent:** Use atomic writes where you write to a temporary file, verify its complete contents, then rename it to replace the original.

The atomic file writing process follows a specific sequence to prevent data loss:

- Generate Temporary Filename:** Create unique temp file in same directory as target (ensures same filesystem for atomic rename)
- Write Complete Content:** Write all editor content to temporary file, checking each write operation
- Verify Write Success:** Confirm all bytes written, call `fflush()` and `fsync()` to force to disk
- Create Backup:** If backup enabled, rename original file to `.bak` extension
- Atomic Rename:** Use `rename()` system call to atomically replace original with temporary file
- Update Editor State:** Mark buffer as clean and update internal filename references
- Cleanup:** Remove temporary files only after successful rename

The key insight is that `rename()` is atomic on most filesystems when source and destination are on the same filesystem. This means the file switch happens instantaneously without any moment where a partial file exists.

File permission handling requires careful consideration of different user scenarios:

| Scenario                     | Detection                                    | User Experience                            | Implementation                                   |
|------------------------------|--|--|--|
| Read-only file               | <code>access()</code> check or write failure | Clear indication of read-only status       | Show in status bar, disable save shortcuts       |
| Missing permissions          | <code>errno == EACCES</code> on open         | Clear error message with suggested actions | Offer to save to different location              |
| File locked by other process | Platform-specific lock detection             | Informative error about conflict           | Suggest closing other applications               |
| Directory permissions        | Parent directory not writable                | Cannot create backup or temp files         | Check directory permissions before save attempts |

### ⚠ Pitfall: Cross-Filesystem Atomic Operations

Attempting atomic operations across different filesystems breaks the atomicity guarantee and can lead to data loss.

**Why it fails:** The `rename()` system call is only atomic when source and destination are on the same filesystem. Cross-filesystem renames become copy-and-delete operations that can fail midway.

**How to detect:** Check if temporary file directory is on same filesystem as target file using `stat()` and comparing device numbers.

**How to fix:** Always create temporary files in the same directory as the target file, not in system temp directories like `/tmp`.

## Text Editing Boundary Conditions

Text editing operations involve numerous boundary conditions that can cause crashes, corruption, or unexpected behavior if not handled properly. Think of boundary conditions like the edge pieces of a jigsaw puzzle - they have different rules than the middle pieces and require special handling to maintain the puzzle's integrity.

The text buffer presents several categories of boundary conditions that must be handled consistently:

| Boundary Type          | Conditions                     | Common Issues  | Handling Strategy                                 |
|------------------------|--------------------------------|--|---|
| Empty Buffer           | Zero lines, zero characters    | Cursor positioning, operations on non-existent content | Maintain at least one empty line, bounds checking |
| Beginning/End of Lines | Cursor at column 0 or line end | Backspace/delete behavior, navigation wrapping         | Clear rules for each operation at boundaries      |
| Beginning/End of File  | First/last line operations     | Insert/delete at extremes, scrolling limits            | Bounds checking, viewport management              |
| Memory Limits          | Very large files, long lines   | Allocation failures, performance degradation           | Graceful degradation, chunking strategies         |
| Unicode Boundaries     | Multi-byte characters          | Cursor movement within characters                      | Character-aware positioning                       |

## Decision: Always Maintain One Line Minimum

- **Context:** Empty buffers create special cases throughout the codebase for cursor positioning and operations
- **Options Considered:**
  1. Allow completely empty buffer (complex special cases everywhere)
  2. Maintain minimum of one empty line (simpler invariants)
  3. Distinguish between empty file and empty buffer (complex state management)
- **Decision:** Always maintain at least one line in the buffer, even if it's empty
- **Rationale:** Simplifies cursor positioning, navigation, and editing operations by eliminating empty buffer special cases
- **Consequences:** Slight memory overhead but dramatically reduces code complexity and edge case bugs

The cursor positioning system must handle several boundary scenarios:

**End-of-Line Behavior:** When the cursor is positioned at the end of a line, different operations have different semantics. For example:

- **Arrow Right:** Move to beginning of next line (if exists)
- **Delete Key:** Join current line with next line
- **Character Insert:** Append to current line
- **Enter Key:** Split line at current position (cursor at end means create empty line below)

**Beginning-of-Line Behavior:** When cursor is at column 0, operations must handle line boundaries:

- **Arrow Left:** Move to end of previous line (if exists)
- **Backspace:** Join current line with previous line
- **Home Key:** No movement (already at beginning)
- **Character Insert:** Insert at line beginning

The line manipulation operations present complex boundary conditions:

| Operation           | At First Line            | At Last Line            | At Empty Line              | At Line Boundary            |
|---------------------|--------------------------|-------------------------|----------------------------|-----------------------------|
| Insert Line Above   | Create new first line    | Normal operation        | Normal operation           | Cursor positioning          |
| Insert Line Below   | Normal operation         | Create new last line    | Normal operation           | Cursor positioning          |
| Delete Current Line | Cursor to new first line | Cursor to new last line | Cannot delete if only line | Maintain minimum line count |
| Join with Previous  | Cannot join first line   | Normal operation        | Previous line extended     | Handle empty content        |
| Join with Next      | Normal operation         | Cannot join last line   | Next line content moved up | Handle empty content        |

### ⚠ Pitfall: Off-by-One Errors in Cursor Positioning

Cursor positioning involves multiple coordinate systems that can cause off-by-one errors: logical character positions, visual screen columns, and zero-based vs one-based indexing.

**Why it's wrong:** Mixing coordinate systems leads to cursor appearing one position off from where user expects, or worse, accessing invalid memory locations.

#### Common mistakes:

- Using screen coordinates (1-based) for buffer operations (0-based)
- Not accounting for tab expansion when converting cursor positions
- Forgetting that line length doesn't include null terminator for positioning

**How to fix:** Establish clear conventions for each coordinate system and use conversion functions consistently. Always validate cursor positions after operations.

Memory management for text operations requires careful boundary checking:

```
typedef struct {
    int max_lines;
    int max_line_length;
    size_t total_memory_limit;
    int current_lines;
    size_t current_memory;
} BufferLimits;
```

C

The memory management strategy must handle various resource exhaustion scenarios:

1. **Line Count Limits**: Prevent runaway line creation (e.g., pasting millions of newlines)
2. **Line Length Limits**: Handle extremely long lines that could cause integer overflow
3. **Total Memory Limits**: Monitor overall buffer memory usage to prevent system exhaustion
4. **Reallocation Failures**: Handle cases where line arrays or content cannot be expanded

### ⚠ Pitfall: Integer Overflow in Buffer Operations

Large files or extremely long lines can cause integer overflow in size calculations, leading to buffer overruns and crashes.

**Why it's dangerous:** Line lengths are often stored as `int`, but can theoretically exceed `INT_MAX`. Buffer size calculations can wrap around and allocate insufficient memory.

**How to detect:** Check for overflow before arithmetic operations: `if (a > INT_MAX - b) /* overflow */`

**How to prevent:** Use `size_t` for all buffer size calculations, validate input ranges, and set reasonable limits on line lengths and buffer sizes.

The scroll management system must handle boundary conditions for viewport positioning:

| Scroll Direction | At Boundary          | Cursor Behavior             | Viewport Adjustment        |
|------------------|----------------------|-----------------------------|----------------------------|
| Up               | At first line        | No movement                 | Viewport stays at top      |
| Down             | At last line         | No movement                 | Viewport shows end of file |
| Left             | At beginning of line | Wrap to previous line end   | Adjust horizontal offset   |
| Right            | At end of line       | Wrap to next line beginning | Adjust horizontal offset   |
| Page Up          | Near beginning       | Move to first line          | Viewport to top            |
| Page Down        | Near end             | Move to last line           | Viewport to bottom         |

The undo system adds additional boundary conditions that must be managed:

**Empty Undo Stack:** When no operations have been performed or all operations have been undone:

- Undo commands should have no effect and possibly provide user feedback
- Redo stack may contain operations that can be reapplied

**Undo Stack Limits:** When the undo stack reaches capacity:

- Oldest operations must be discarded to make room for new ones
- Save point tracking must be updated when save point operation is discarded

**Complex Operation Boundaries:** Operations that span multiple buffer modifications:

- Multi-character paste operations must undo as single unit
- Find-and-replace operations across multiple matches need compound undo records

### Pitfall: Cursor Validation After Undo Operations

After undoing operations, the cursor may be positioned at locations that no longer exist in the buffer.

**Why it's wrong:** If undo removes lines or shortens lines, the cursor position may now point beyond the buffer boundaries, causing crashes or unexpected behavior.

**How to detect:** After every undo operation, validate that cursor row is within buffer bounds and cursor column is within line bounds.

**How to fix:** Implement `validate_cursor_position()` that clamps cursor to valid positions and call it after all buffer modifications.

The search system introduces boundary conditions around match positions and wrap-around behavior:

**Search Wrap-Around:** When search reaches end of file:

- Forward search continues from beginning of file
- Backward search continues from end of file
- Must detect when search has completed full cycle to avoid infinite loops

**Match Boundary Validation:** Search matches must be validated:

- Match positions must be within valid line and column bounds
- Multi-line matches need special handling for line boundaries
- Unicode characters require character-boundary-aware matching

## Implementation Guidance

The error handling implementation requires careful coordination between multiple systems and defensive programming techniques throughout the codebase.

### Technology Recommendations:

| Component         | Simple Option                                  | Advanced Option                            |
|-------------------|--|--|
| Signal Handling   | Basic SIGINT/SIGTERM handlers                  | Full signal mask with custom handlers      |
| File I/O          | Standard C file operations with error checking | Memory-mapped files with atomic operations |
| Memory Management | Manual tracking with limits                    | Custom allocator with leak detection       |
| Error Reporting   | Simple status messages                         | Structured error codes with user guidance  |

## **Recommended File Structure:**

```
editor/
src/
    error_handling.c      ← Central error handling utilities
    error_handling.h      ← Error codes and handler prototypes
    terminal_recovery.c   ← Terminal state management
    file_operations.c     ← Safe file I/O operations
    buffer_validation.c   ← Boundary condition checking
    signal_handlers.c     ← Async-signal-safe cleanup
include/
    error_types.h         ← Error enumeration and structures
    safety_macros.h       ← Defensive programming macros
tests/
    error_scenarios/      ← Crash testing and recovery verification
```

## **Error Handling Infrastructure (Complete Implementation):**

```
// error_types.h - Complete error enumeration and structures

#ifndef ERROR_TYPES_H

#define ERROR_TYPES_H


#include <time.h>

typedef enum {

    ERR_NONE = 0,

    ERR_TERMINAL_RAW_MODE_FAILED,

    ERR_TERMINAL_RESTORE_FAILED,

    ERR_FILE_NOT_FOUND,

    ERR_FILE_PERMISSION_DENIED,

    ERR_FILE_TOO_LARGE,

    ERR_OUT_OF_MEMORY,

    ERR_CURSOR_OUT_OF_BOUNDS,

    ERR_LINE_TOO_LONG,

    ERR_BUFFER_CORRUPTED,

    ERR_INVALID_UTF8

} ErrorCode;

typedef struct {

    ErrorCode code;

    char message[256];

    char context[128];

    time_t timestamp;

    const char* file;

    int line;

} EditorError;
```

C

```
// Global error state

extern EditorError last_error;

extern int error_logging_enabled;

// Error reporting macros

#define SET_ERROR(code, msg, ctx) \
    set_error(code, msg, ctx, __FILE__, __LINE__)

#define CLEAR_ERROR() \
    (last_error.code = ERR_NONE)

#define HAS_ERROR() \
    (last_error.code != ERR_NONE)

#endif
```

```
// terminal_recovery.c - Complete terminal state management
```

C

```
#include <termios.h>
```

```
#include <signal.h>
```

```
#include <unistd.h>
```

```
#include "error_types.h"
```

```
static struct termios original_termios;
```

```
static int termios_saved = 0;
```

```
// Async-signal-safe cleanup handler
```

```
void emergency_cleanup(int sig) {
```

```
    if (termios_saved) {
```

```
        tcsetattr(STDIN_FILENO, TCSAFLUSH, &original_termios);
```

```
}
```

```
    _exit(1);
```

```
}
```

```
int enable_raw_mode(void) {
```

```
    // TODO 1: Get current terminal attributes using tcgetattr()
```

```
    // TODO 2: Save original attributes in global variable
```

```
    // TODO 3: Modify attributes - disable ECHO, ICANON, ISIG, IEXTEN
```

```
    // TODO 4: Apply modified attributes with tcsetattr()
```

```
    // TODO 5: Set global flag indicating termios is saved
```

```
    // TODO 6: Install signal handlers for emergency cleanup
```

```
}
```

```
void disable_raw_mode(void) {
```

```
    // TODO 1: Check if original termios was saved
```

```
    // TODO 2: Restore original terminal attributes
```

```
// TODO 3: Clear the saved flag

// Hint: This must be callable from signal handlers - keep it simple

}

void setup_signal_handlers(void) {

    // TODO 1: Install handler for SIGINT (Ctrl+C)

    // TODO 2: Install handler for SIGTERM (kill command)

    // TODO 3: Install handler for SIGSEGV (segmentation fault)

    // TODO 4: Consider handler for SIGPIPE (broken pipe)

    // Hint: Use sigaction() for portable signal handling

}
```

### File Operations Safety (Complete Implementation):

```
// file_operations.h - Safe file I/O interface

#ifndef FILE_OPERATIONS_H
#define FILE_OPERATIONS_H

#include "error_types.h"

typedef struct {

    char* content;

    size_t size;

    int line_count;

    LineEndingType endings;

} FileContent;

// Safe file operations with comprehensive error handling

FileContent* load_file_safely(const char* filename);

int save_file_atomically(const char* filename, const FileContent* content);

int create_backup_file(const char* filename);

int validate_file_permissions(const char* filename);

void free_file_content(FileContent* content);

// File operation result checking

#define CHECK_FILE_OP(op, error_action) \
    if (!(op)) { \
        error_action; \
    }

#endif
```

C

```
// file_operations.c - Atomic file operations with error handling

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include "file_operations.h"

FileContent* load_file_safely(const char* filename) {

    // TODO 1: Validate filename parameter (not NULL, not empty)

    // TODO 2: Check if file exists and is readable using access()

    // TODO 3: Get file size using stat() and validate reasonable size

    // TODO 4: Open file with error checking

    // TODO 5: Allocate memory for content with size validation

    // TODO 6: Read file in chunks, handling partial reads

    // TODO 7: Detect line ending type by scanning content

    // TODO 8: Count lines for buffer initialization

    // TODO 9: Return populated FileContent or NULL on error

}

int save_file_atomically(const char* filename, const FileContent* content) {

    // TODO 1: Generate temporary filename in same directory as target

    // TODO 2: Open temporary file with appropriate permissions

    // TODO 3: Write all content to temporary file with error checking

    // TODO 4: Call fflush() and fsync() to ensure data written to disk

    // TODO 5: Close temporary file and check for errors

    // TODO 6: Create backup of original file if it exists

    // TODO 7: Atomically rename temporary file to target filename
```

C

```
// TODO 8: Clean up temporary files on any failure

// Hint: Use rename() for atomic file replacement

}
```

### Buffer Validation System (Core Logic Skeleton):

```
// buffer_validation.h - Boundary condition checking C

typedef struct {

    int min_row, max_row;

    int min_col, max_col;

    int total_lines;

    size_t memory_used;

    size_t memory_limit;

} BufferBounds;

int validate_cursor_position(int row, int col);

int validate_buffer_operation(int row, int col, int operation_type);

int check_memory_limits(size_t additional_bytes);

BufferBounds get_current_buffer_bounds(void);
```

```

// buffer_validation.c - Implementation of boundary checking

C

int validate_cursor_position(int row, int col) {

    // TODO 1: Check if row is within buffer bounds (0 to numrows-1)

    // TODO 2: Check if col is within current line bounds

    // TODO 3: Handle special case of empty buffer (should have one empty line)

    // TODO 4: Return 1 for valid position, 0 for invalid

    // Hint: Use editor_config global state to check bounds

}

int validate_buffer_operation(int row, int col, int operation_type) {

    // TODO 1: Validate cursor position using validate_cursor_position()

    // TODO 2: Check operation-specific constraints (insert vs delete)

    // TODO 3: Verify memory limits won't be exceeded for insert operations

    // TODO 4: Check line length limits for character insertions

    // TODO 5: Return error code or ERR_NONE for valid operations

}

int check_memory_limits(size_t additional_bytes) {

    // TODO 1: Calculate current memory usage across all lines

    // TODO 2: Check if additional bytes would exceed configured limits

    // TODO 3: Consider fragmentation and reallocation overhead

    // TODO 4: Return 1 if operation is safe, 0 if would exceed limits

    // Hint: Set reasonable defaults like 100MB total buffer limit

}

```

### Milestone Checkpoints:

After implementing terminal recovery:

- **Test Command:** Kill editor with Ctrl+C, verify terminal echo still works
- **Expected Behavior:** Terminal should respond normally to typing after editor exits

- **Error Signs:** Missing character echo, garbled output, cursor positioning issues
- **Fix:** Run `reset` command to restore terminal, check signal handler installation

After implementing file operations:

- **Test Command:** Try saving to read-only directory, loading non-existent files
- **Expected Behavior:** Clear error messages, no data corruption, graceful fallbacks
- **Error Signs:** Crashes on file errors, silent data loss, permission errors ignored
- **Fix:** Add comprehensive error checking around all file operations

After implementing boundary validation:

- **Test Command:** Navigate to extremes (beginning/end of file), try operations at boundaries
- **Expected Behavior:** Consistent cursor behavior, no crashes at edges
- **Error Signs:** Crashes when cursor at line end, wrong cursor position after undo
- **Fix:** Add bounds checking to all cursor movement and buffer operations

### Debugging Tips:

| Symptom                           | Likely Cause                 | Diagnosis                                  | Fix  |
|-----------------------------------|------------------------------|--|--|
| Terminal broken after crash       | Signal handler not installed | Check if cleanup runs on Ctrl+C            | Install SIGINT handler calling <code>disable_raw_mode()</code> |
| File appears corrupted after save | Non-atomic write operation   | Check if partial content written           | Implement atomic write with temporary file                     |
| Cursor jumps to wrong position    | Coordinate system mismatch   | Print cursor coordinates during operations | Use consistent 0-based indexing, convert for display           |
| Editor crashes at file boundaries | Missing bounds checking      | Run under debugger, check stack trace      | Add validation before all buffer operations                    |
| Memory usage grows continuously   | Memory leak in error paths   | Use valgrind or similar memory checker     | Ensure cleanup on all error return paths                       |

## Testing Strategy

**Milestone(s):** All milestones (testing approaches span the entire development process)

Testing a terminal-based text editor presents unique challenges that differ significantly from testing traditional applications. Think of testing a terminal editor like quality-assuring a complex theatrical performance - you need to verify not just that the actors (components) know their lines, but that the lighting (screen rendering), sound (terminal I/O), and stage direction (user interactions) all work together seamlessly. Unlike web

applications where you can easily inspect DOM elements or desktop applications with accessible UI controls, terminal applications operate through a stream of bytes and ANSI escape sequences that require specialized verification approaches.

The complexity of terminal testing stems from the stateful nature of terminal interactions. Each keypress potentially changes the editor's internal state, the visual display, and the cursor position simultaneously. A single character insertion involves coordinating between the input handler, text buffer, undo system, and screen renderer. Testing must verify not just that the character appears on screen, but that it's stored correctly in the buffer, recorded properly for undo, and renders at the correct visual position with appropriate syntax highlighting.

## Architecture Decision: Testing Strategy Approach

### Decision: Multi-layered Testing with Component Isolation

- **Context:** Terminal editors have complex interdependencies between components, making bugs difficult to isolate. Pure integration testing would be slow and unreliable, while pure unit testing would miss critical interaction bugs.
- **Options Considered:**
  1. Pure integration testing with scripted terminal sessions
  2. Pure unit testing with mocked dependencies
  3. Multi-layered approach with unit tests for components plus milestone verification checkpoints
- **Decision:** Multi-layered approach combining isolated unit tests with milestone-based integration verification
- **Rationale:** Unit tests catch component-level logic errors quickly during development. Milestone checkpoints verify that components integrate correctly and produce expected user-visible behavior. This combination provides fast feedback for development while ensuring the system works end-to-end.
- **Consequences:** Requires more test infrastructure setup but provides comprehensive coverage with fast feedback loops. Test maintenance overhead is higher but bug detection is more thorough.

| Testing Approach | Coverage                   | Speed | Isolation                                     | Maintenance | Chosen |
|------------------|----------------------------|-------|---|-------------|--------|
| Pure Integration | End-to-end behavior        | Slow  | Poor (hard to isolate failures)               | Low         | No     |
| Pure Unit        | Component logic            | Fast  | Excellent                                     | Medium      | No     |
| Multi-layered    | Both logic and integration | Mixed | Good for components, adequate for integration | High        | Yes    |

The multi-layered approach recognizes that different types of bugs require different detection strategies. Logic errors in text buffer operations are best caught with fast unit tests that can run hundreds of scenarios in seconds. Integration issues like cursor positioning after scrolling are best caught with milestone verification that tests the complete user workflow.

## Milestone Verification Checkpoints

Think of milestone verification checkpoints as quality gates in a manufacturing pipeline - each milestone must meet specific behavioral requirements before proceeding to the next level of complexity. These checkpoints focus on externally observable behavior rather than internal implementation details, ensuring that the editor works correctly from the user's perspective while allowing flexibility in how features are implemented internally.

Each checkpoint defines three levels of verification: **automated behavior tests** that can be scripted, **manual interaction tests** that require human verification, and **stress tests** that push the implementation to its limits. This comprehensive approach catches both obvious functional bugs and subtle edge cases that emerge under real usage conditions.

### Milestone 1: Raw Mode and Input Verification

The first milestone establishes the foundation of terminal control and input processing. Verification focuses on ensuring that the terminal enters raw mode correctly, processes individual keypresses without waiting for line endings, and restores terminal state reliably under all exit conditions.

#### Automated Verification Steps:

1. **Raw Mode Activation Test:** Create a test program that calls `enable_raw_mode()` and verifies that terminal echo is disabled by sending characters to stdin and confirming they don't appear in stdout without explicit writing.
2. **Keypress Reading Test:** Verify that `editor_read_key()` returns individual characters immediately without waiting for Enter key. Test with regular characters ('a', 'b', '1', ' ') and confirm each returns the correct ASCII value.
3. **Escape Sequence Processing Test:** Send multi-byte escape sequences for arrow keys and verify they map to correct `EditorKey` constants. Test sequence: `\x1b[A` should return `ARROW_UP`, `\x1b[B` should return `ARROW_DOWN`, etc.
4. **Terminal Restoration Test:** Call `disable_raw_mode()` and verify that echo is restored by sending characters and confirming they appear in output.

#### Manual Interaction Tests:

1. **Interactive Key Response:** Run the editor and type characters - they should not echo automatically to the terminal. Only characters explicitly written by the editor should appear.
2. **Arrow Key Navigation:** Press arrow keys and verify they are recognized as navigation commands rather than printing escape sequences.

3. **Ctrl+C Handling:** Press Ctrl+C and verify the terminal is restored to canonical mode before the program exits.

#### Stress Tests:

- Rapid Keypress Handling:** Send burst of 1000 keypresses and verify all are processed correctly without buffer overflow or lost input.
- Signal Interruption Recovery:** Send SIGTERM or SIGINT during operation and verify terminal state is restored properly.

| Test Category           | Expected Behavior                                 | Failure Symptoms                                     | Recovery Steps  |
|-------------------------|---|--|---|
| Raw Mode Setup          | Characters don't echo, immediate input processing | Characters appear twice, or input waits for Enter    | Check <code>termios</code> flags, verify <code>tcsetattr()</code> success |
| Escape Sequence Parsing | Arrow keys navigate, don't print sequences        | <code>^[[A</code> appears instead of cursor movement | Debug sequence detection logic, check timeout values                      |
| Terminal Restoration    | Terminal works normally after editor exit         | Terminal stuck in raw mode, no echo                  | Ensure <code>disable_raw_mode()</code> called in all exit paths           |

## Milestone 2: Screen Refresh Verification

Screen refresh verification focuses on ensuring that the editor can control the terminal display through ANSI escape sequences, position the cursor accurately, and avoid visual artifacts like flickering during updates.

#### Automated Verification Steps:

- Screen Clearing Test:** Call `clear_screen()` and verify that the terminal receives the `ANSI_CLEAR_SCREEN` sequence (`\x1b[2J`).
- Cursor Positioning Test:** Call `position_cursor(5, 10)` and verify the terminal receives the correct positioning sequence (`\x1b[5;10H`).
- Status Bar Rendering Test:** Create an `editor_config` with known values and call `editor_refresh_screen()`. Verify that status bar shows correct filename, line count, and cursor position.
- Frame Buffer Consistency Test:** Verify that screen updates are written as complete frames rather than individual character writes to prevent flickering.

#### Manual Interaction Tests:

- Visual Refresh Verification:** Resize terminal window and verify that editor detects new dimensions and redraws correctly.

2. **Cursor Synchronization:** Move cursor with arrow keys and verify that visual cursor position matches logical cursor position in status bar.
3. **Screen Content Stability:** Verify that screen content doesn't flicker or show partial updates during refresh operations.

#### **Stress Tests:**

1. **Rapid Refresh Handling:** Trigger 100 consecutive screen refreshes and verify no visual artifacts or performance degradation.
2. **Terminal Resize Handling:** Rapidly resize terminal window and verify editor adapts correctly without crashes or display corruption.

| Test Category            | Expected Behavior                  | Failure Symptoms                         | Recovery Steps  |
|--------------------------|------------------------------------|--|---|
| ANSI Sequence Generation | Cursor moves to specified position | Cursor appears in wrong location         | Check escape sequence format, verify 1-based indexing     |
| Screen Clearing          | Entire screen clears before redraw | Old content remains visible              | Ensure <code>ANSI_CLEAR_SCREEN</code> sent before content |
| Frame Buffer Writing     | No flicker during updates          | Screen flickers or shows partial content | Buffer complete frame before writing to terminal          |

### **Milestone 3: File Viewing Verification**

File viewing verification ensures that the editor can load files into memory, display content correctly with line numbers, and handle scrolling both vertically and horizontally for files that exceed terminal dimensions.

#### **Automated Verification Steps:**

1. **File Loading Test:** Create test files with known content and verify that `editor_open()` loads all lines correctly into the `erow` array structure.
2. **Line Number Display Test:** Load a file with 50 lines and verify that line numbers appear in the left gutter with consistent width padding.
3. **Vertical Scrolling Test:** Create a file with more lines than `screenrows` and verify that `editor_scroll()` adjusts `rowoff` to keep cursor visible.
4. **Horizontal Scrolling Test:** Create a line longer than `screencols` and verify that horizontal scrolling reveals hidden content when cursor moves beyond visible area.

#### **Manual Interaction Tests:**

1. **File Content Verification:** Open a known text file and verify all content displays correctly with proper line breaks and character encoding.

2. **Scrolling Navigation:** Use arrow keys to navigate through a large file and verify that scrolling keeps the cursor visible and displays correct content.
3. **Long Line Handling:** Open a file with lines longer than terminal width and verify horizontal scrolling works correctly.

#### **Stress Tests:**

1. **Large File Handling:** Open a file with 10,000 lines and verify that loading, display, and navigation remain responsive.
2. **Wide Line Handling:** Open a file with lines containing 1000 characters and verify horizontal scrolling works without performance issues.

| Test Category       | Expected Behavior                                | Failure Symptoms                       | Recovery Steps   |
|---------------------|--|--|--|
| File I/O            | All file content loaded into buffer              | Missing lines, garbled content         | Check file reading logic, verify line ending handling                |
| Viewport Management | Cursor always visible, content scrolls correctly | Cursor disappears off screen           | Debug <code>editor_scroll()</code> logic, verify offset calculations |
| Line Number Display | Consistent gutter width, correct numbering       | Misaligned content, wrong line numbers | Check gutter width calculation, verify 1-based numbering             |

## **Milestone 4: Text Editing Verification**

Text editing verification focuses on the core editing operations: character insertion, deletion, line breaks, and line joining. These operations must maintain cursor consistency, preserve undo information, and handle edge cases at line boundaries.

#### **Automated Verification Steps:**

1. **Character Insertion Test:** Position cursor at known location, insert character with `editor_insert_char()`, and verify character appears in correct position in both buffer and display.
2. **Character Deletion Test:** Position cursor after known character, call `editor_delete_char()`, and verify character is removed from buffer and display.
3. **Line Breaking Test:** Position cursor in middle of line, call `editor_insert_newline()`, and verify line splits correctly with cursor on new line.
4. **Line Joining Test:** Position cursor at beginning of line, press backspace, and verify current line joins with previous line.

#### **Manual Interaction Tests:**

1. **Typing Flow:** Type several sentences with punctuation and verify all characters appear correctly at cursor position.

- Editing Operations:** Insert characters in middle of existing text, delete characters, and verify surrounding text shifts appropriately.
- Line Manipulation:** Create new lines with Enter and join lines with backspace at line boundaries.

#### Stress Tests:

- Rapid Editing:** Type 1000 characters rapidly and verify all appear correctly without lost input or buffer corruption.
- Boundary Editing:** Perform editing operations at beginning of file, end of file, and line boundaries to verify edge case handling.

| Test Category        | Expected Behavior                           | Failure Symptoms                                  | Recovery Steps  |
|----------------------|---|---|---|
| Character Operations | Characters insert/delete at cursor position | Text appears in wrong location, buffer corruption | Debug cursor position tracking, verify buffer operations        |
| Line Operations      | Lines split and join correctly              | Content lost during line operations               | Check line boundary logic, verify memory management             |
| Cursor Consistency   | Cursor position matches displayed location  | Visual cursor doesn't match logical position      | Debug <code>cx_to_rx()</code> conversion, verify cursor updates |

## Milestone 5: Save and Undo Verification

Save and undo verification ensures that file operations preserve data correctly and that the undo system maintains operation history reliably. This milestone is critical for data integrity and user confidence.

#### Automated Verification Steps:

- File Saving Test:** Make edits to buffer, call `editor_save()`, and verify that saved file contains exactly the buffer content with correct line endings.
- Dirty Flag Test:** Make edit and verify dirty flag is set. Save file and verify dirty flag is cleared. Make another edit and verify flag is set again.
- Undo Operation Test:** Perform sequence of edits, call `undo_execute()`, and verify that operations are reversed in correct order with proper buffer state.
- Redo Operation Test:** Undo several operations, then call `redo_execute()` and verify operations are re-applied correctly.

#### Manual Interaction Tests:

- Save Confirmation:** Make edits, save file, and verify that saved content matches what's displayed in editor.
- Undo Workflow:** Type text, undo changes, and verify that buffer returns to previous state with correct cursor position.

3. **Redo Workflow:** Undo several operations, then redo them and verify buffer returns to intermediate states correctly.

#### Stress Tests:

1. **Large File Saving:** Edit file with 10,000 lines, save, and verify save completes without data loss or corruption.
2. **Undo History Limits:** Perform 1000 edit operations and verify undo system maintains reasonable history without memory exhaustion.

| Test Category     | Expected Behavior                 | Failure Symptoms                                   | Recovery Steps                                  |
|-------------------|-----------------------------------|--|---|
| File Operations   | Saved file matches buffer exactly | Data loss, corruption, wrong line endings          | Debug file writing logic, check error handling  |
| Undo Consistency  | Operations reverse correctly      | Wrong content after undo, cursor in wrong position | Verify undo recording, check operation reversal |
| Memory Management | Undo stack doesn't grow unbounded | Memory usage increases without limit               | Implement undo history limits, verify cleanup   |

#### Milestone 6: Search Verification

Search verification focuses on incremental search functionality, match highlighting, and navigation between search results. The search system must handle edge cases like wrapping around the document and canceling searches.

#### Automated Verification Steps:

1. **Incremental Search Test:** Create buffer with known content, activate search with `search_activate()`, add characters to query, and verify matches are found and highlighted correctly.
2. **Search Navigation Test:** Find multiple matches for a query, call `search_next_match()` and `search_prev_match()`, and verify cursor moves to correct match positions.
3. **Search Wrap Test:** Search for pattern that appears at beginning and end of file, verify search wraps around correctly in both directions.
4. **Search Cancel Test:** Start search, move cursor to match, then call `search_deactivate(1)` and verify cursor returns to original position.

#### Manual Interaction Tests:

1. **Interactive Search:** Press search key, type query, and verify that matches highlight as each character is typed.
2. **Match Navigation:** Use next/previous search commands and verify cursor jumps between matches with proper highlighting.

- Search Cancellation:** Start search, press Escape, and verify search mode exits with cursor at original position.

#### Stress Tests:

- Pattern Matching Performance:** Search for common patterns in large files and verify search completes within reasonable time.
- Complex Query Handling:** Search for patterns with special characters and verify correct matching behavior.

| Test Category    | Expected Behavior                       | Failure Symptoms                          | Recovery Steps  |
|------------------|---|---|---|
| Match Detection  | All occurrences found and highlighted   | Missing matches, incorrect highlighting   | Debug search algorithm, verify pattern matching         |
| Navigation       | Cursor jumps to correct match positions | Cursor moves to wrong location            | Check match position calculation, verify cursor updates |
| State Management | Search state transitions correctly      | Search mode stuck, incorrect cancellation | Debug search state machine, verify cleanup              |

## Milestone 7: Syntax Highlighting Verification

Syntax highlighting verification ensures that code elements are colored correctly based on programming language rules and that highlighting doesn't interfere with editing operations or performance.

#### Automated Verification Steps:

- Language Detection Test:** Open files with different extensions (.c, .py, .js) and verify that `detect_language()` returns correct `LanguageDefinition` for each.
- Keyword Highlighting Test:** Create source code with known keywords and verify that `tokenize_row()` identifies keywords correctly and `apply_syntax_colors()` applies appropriate color codes.
- String Highlighting Test:** Create code with string literals and verify that strings are tokenized correctly including proper handling of escape sequences.
- Comment Highlighting Test:** Create code with both single-line and multi-line comments and verify that comment spans are identified and colored correctly.

#### Manual Interaction Tests:

- Visual Highlighting:** Open source code files and verify that syntax elements are colored appropriately with readable color scheme.
- Editing Consistency:** Edit highlighted code and verify that syntax highlighting updates correctly after modifications.

- Performance Impact:** Edit large source files and verify that syntax highlighting doesn't cause noticeable delays in editing responsiveness.

#### Stress Tests:

- Large File Highlighting:** Open source file with 5,000 lines and verify highlighting completes without performance degradation.
- Complex Syntax Handling:** Open files with nested structures (strings in comments, etc.) and verify correct highlighting behavior.

| Test Category     | Expected Behavior                                | Failure Symptoms                          | Recovery Steps  |
|-------------------|--|---|---|
| Token Recognition | Keywords, strings, comments identified correctly | Wrong elements highlighted, missed tokens | Debug tokenization rules, verify language definitions     |
| Color Application | Appropriate colors applied to syntax elements    | No colors, wrong colors, garbled output   | Check ANSI color code generation, verify terminal support |
| Performance       | Highlighting doesn't slow editing operations     | Delays during typing, slow scrolling      | Optimize tokenization, consider incremental highlighting  |

## Automated Testing Approaches

Automated testing for terminal applications requires specialized techniques that account for the unique challenges of testing byte streams, ANSI escape sequences, and stateful terminal interactions. Think of automated testing for a terminal editor like testing a complex state machine where each input can potentially change multiple aspects of the system simultaneously.

The key insight for terminal testing is that while you cannot easily test the visual appearance of the terminal output, you can test the **data transformations** that produce that output. Instead of verifying that text appears blue on the screen, you verify that the correct ANSI color codes are generated. Instead of checking cursor position visually, you verify that the correct positioning escape sequences are sent to the terminal.

### Component-Level Unit Testing

Component-level testing focuses on individual modules in isolation, allowing rapid testing of complex logic without the overhead of terminal setup and teardown. Each component has specific testing requirements based on its responsibilities and interfaces.

#### Text Buffer Testing Strategy:

The text buffer component handles the core data transformations for text editing. Unit tests should focus on the correctness of buffer operations under various conditions, including edge cases that are difficult to trigger through manual testing.

| Test Category        | Test Cases                                 | Expected Results   | Common Failures                          |
|----------------------|--|--|--|
| Character Operations | Insert at beginning, middle, end of line   | Character appears at correct position, buffer size updates | Off-by-one errors, buffer overflow       |
| Line Operations      | Split line, join lines, insert/delete rows | Line structure maintained, cursor position updated         | Memory leaks, incorrect line boundaries  |
| File I/O             | Load files with various line endings       | All content preserved, correct line ending detection       | Encoding issues, truncated files         |
| Buffer Bounds        | Operations at file boundaries              | Graceful handling, no crashes                              | Segmentation faults, array bounds errors |

Test implementation should use **dependency injection** to isolate the text buffer from terminal I/O and rendering components. Create test buffers with known content and verify that operations produce expected results without requiring terminal interaction.

#### Terminal Manager Testing Strategy:

Testing the terminal manager requires careful handling of terminal state to avoid interfering with the test environment. The key approach is to test the **ANSI escape sequence generation** rather than the actual terminal state changes.

| Test Category              | Test Cases                          | Expected Results                       | Common Failures                             |
|----------------------------|-------------------------------------|--|---|
| Escape Sequence Generation | Cursor positioning, screen clearing | Correct ANSI codes generated           | Wrong sequence format, incorrect parameters |
| Raw Mode Setup             | Enable/disable raw mode             | Correct <code>termios</code> flags set | Terminal left in wrong state                |
| Signal Handling            | Process termination, interruption   | Terminal restored correctly            | Terminal corrupted after crash              |
| Window Size Detection      | Terminal resize                     | Correct dimensions detected            | Incorrect screen size calculations          |

Use **mock terminals** or **captured output streams** to verify escape sequence generation without affecting the actual terminal. Create wrapper functions that capture terminal output for verification:

```

// Test helper that captures terminal output

typedef struct {

    char* captured_output;

    size_t output_length;

    size_t output_capacity;

} MockTerminal;

MockTerminal* create_mock_terminal();

void mock_terminal_write(MockTerminal* mock, const char* data, size_t len);

int verify_escape_sequence(MockTerminal* mock, const char* expected_sequence);

```

### Input Handler Testing Strategy:

Input handler testing focuses on the **keypress parsing** and **command generation** logic. The challenge is testing multi-byte escape sequences that represent special keys without requiring actual keyboard input.

| Test Category               | Test Cases                                   | Expected Results                           | Common Failures                      |
|-----------------------------|--|--|--------------------------------------|
| Character Parsing           | Regular keys, special keys, escape sequences | Correct key codes returned                 | Wrong key mapping, timing issues     |
| Command Generation          | Key combinations, mode-specific bindings     | Appropriate commands generated             | Wrong commands, missing bindings     |
| Escape Sequence Recognition | Arrow keys, function keys                    | Multi-byte sequences parsed correctly      | Incomplete sequences, timeout errors |
| Mode Handling               | Different editor modes                       | Context-appropriate command interpretation | Commands active in wrong mode        |

Create **input simulation** functions that feed byte sequences to the input handler and verify the resulting commands:

```

// Test helper for simulating keyboard input

typedef struct {

    unsigned char* input_buffer;

    size_t buffer_size;

    size_t current_position;

} InputSimulator;

InputSimulator* create_input_simulator(const char* input_sequence);

int simulate_keypress(InputSimulator* sim);

void verify_command_sequence(InputSimulator* sim, CommandType* expected_commands, int count);

```

### **Undo System Testing Strategy:**

Undo system testing requires verification of complex state transitions and memory management. Tests must ensure that operations are recorded correctly, reversed properly, and that memory is managed efficiently throughout the undo/redo cycle.

| Test Category       | Test Cases                 | Expected Results                | Common Failures                            |
|---------------------|----------------------------|---------------------------------|--|
| Operation Recording | Character/line operations  | Correct undo data captured      | Incomplete recording, wrong operation type |
| Undo Execution      | Single/multiple operations | Operations reversed correctly   | Wrong order, incomplete reversal           |
| Redo Execution      | After undo operations      | Operations re-applied correctly | State corruption, memory issues            |
| Memory Management   | Large operation sequences  | No memory leaks, bounded usage  | Memory growth, dangling pointers           |

Use **operation builders** to create controlled test scenarios without requiring actual text editing:

```

// Test helper for undo system verification

typedef struct {

    UndoStack* stack;

    char* test_buffer;

    size_t buffer_size;

} UndoTestContext;

UndoTestContext* create_undo_test_context();

void simulate_char_insert(UndoTestContext* ctx, int row, int col, int ch);

void simulate_char_delete(UndoTestContext* ctx, int row, int col);

int verify_undo_reverses_operation(UndoTestContext* ctx);

```

## Integration Testing Framework

Integration testing for terminal editors requires a framework that can coordinate multiple components while capturing and verifying the interactions between them. The key challenge is testing the **complete workflow** from input to screen output without human intervention.

### Pseudo-Terminal Testing:

The most effective approach for integration testing uses **pseudo-terminals** (ptys) that allow programmatic control of terminal input and output. This enables automated testing of the complete editor workflow including terminal state management.

| Framework Component | Responsibility                  | Implementation Strategy                            |
|---------------------|---------------------------------|--|
| PTY Manager         | Create/destroy pseudo-terminals | Use <code>openpty()</code> or similar system calls |
| Input Generator     | Send keystroke sequences        | Write to PTY master file descriptor                |
| Output Capture      | Record terminal output          | Read from PTY master, parse ANSI sequences         |
| State Verification  | Check editor state              | Compare expected vs actual buffer/cursor state     |

A pseudo-terminal framework allows testing scenarios like:

1. Send arrow key sequence to editor
2. Capture ANSI cursor positioning output
3. Verify cursor moved to expected location
4. Check internal editor state matches display state

## Test Scenario Scripting:

Integration tests benefit from a **scenario scripting** approach where test cases are defined as sequences of input actions and expected results. This makes tests readable and maintainable while covering complex user workflows.

```
// Example test scenario structure C

typedef struct {

    char* scenario_name;

    char* initial_file_content;

    KeystrokeSequence* input_sequence;

    ExpectedState* final_state;

    char* expected_display_content;

} TestScenario;

// Example scenario: "Insert character in middle of line"

TestScenario insert_char_scenario = {

    .scenario_name = "insert_char_middle_line",

    .initial_file_content = "Hello World\n",

    .input_sequence = create_keystroke_sequence("6G", "i", "Beautiful ", "ESC"),

    .final_state = create_expected_state(1, 16, "Hello Beautiful World\n"),

    .expected_display_content = "Hello Beautiful World"

};
```

## Component Mock Framework:

For testing individual components in integration scenarios, a **mock framework** allows replacement of dependencies with controllable test doubles. This enables testing of error conditions and edge cases that are difficult to reproduce with real components.

| Component        | Mock Strategy                | Verification Points              |
|------------------|------------------------------|----------------------------------|
| Terminal Manager | Capture ANSI sequences       | Verify correct escape codes sent |
| File System      | Simulate I/O errors          | Test error handling paths        |
| Memory Allocator | Simulate allocation failures | Verify graceful degradation      |
| System Calls     | Control return values        | Test edge cases and errors       |

## Performance and Stress Testing

Performance testing for terminal editors focuses on **responsiveness** during interactive use rather than absolute throughput. Users expect immediate visual feedback for keypresses and smooth scrolling through large documents. Stress testing verifies that the editor remains stable under extreme conditions.

### Responsiveness Benchmarks:

| Operation           | Target Response Time | Measurement Method               | Acceptable Limits    |
|---------------------|----------------------|----------------------------------|----------------------|
| Keypress to Display | < 50ms               | Time from input to screen update | < 100ms maximum      |
| Vertical Scrolling  | < 20ms               | Time to redraw after scroll      | < 50ms maximum       |
| File Loading        | < 500ms for 1MB      | Time from open to first display  | < 2s for 10MB        |
| Search              | < 100ms              | Time to highlight first match    | < 500ms for 1MB file |

### Memory Usage Verification:

Terminal editors must manage memory efficiently since they may edit large files for extended periods. Memory testing should verify both **peak usage** and **memory stability** over time.

| Test Type        | Measurement                         | Expected Behavior         | Failure Indicators                   |
|------------------|-------------------------------------|---------------------------|--------------------------------------|
| Peak Memory      | Maximum heap usage during operation | Proportional to file size | Excessive overhead, memory spikes    |
| Memory Stability | Usage over extended session         | Stable or bounded growth  | Memory leaks, unbounded growth       |
| Recovery Testing | Memory usage after operations       | Returns to baseline       | Memory not released after operations |

### Stress Test Scenarios:

| Scenario            | Test Parameters        | Success Criteria                  | Common Failures                  |
|---------------------|------------------------|-----------------------------------|----------------------------------|
| Large File Editing  | 10MB file, 100K lines  | Responsive editing, stable memory | Slow response, crashes           |
| Rapid Input         | 1000 keystrokes/second | All input processed               | Lost keystrokes, buffer overflow |
| Extended Session    | 8 hour editing session | Stable performance                | Memory leaks, degraded response  |
| Resource Exhaustion | Limited memory/disk    | Graceful degradation              | Crashes, data corruption         |

## Continuous Integration Testing

Continuous integration for terminal editors must account for the **headless environment** limitations of CI systems while ensuring comprehensive test coverage across different terminal environments and operating systems.

### CI-Friendly Test Categories:

| Test Category         | CI Suitability | Implementation Strategy                           |
|-----------------------|----------------|---|
| Unit Tests            | Excellent      | Run in headless environment with mocked terminals |
| Component Integration | Good           | Use pseudo-terminals for terminal I/O simulation  |
| End-to-End            | Limited        | Script-driven testing with output capture         |
| Performance           | Good           | Automated benchmarks with pass/fail thresholds    |

### Cross-Platform Testing Matrix:

| Platform       | Terminal Types              | Special Considerations          |
|----------------|-----------------------------|---------------------------------|
| Linux          | xterm, GNOME Terminal, tmux | ANSI sequence compatibility     |
| macOS          | Terminal.app, iTerm2        | BSD terminal differences        |
| Windows        | Windows Terminal, WSL       | Windows console API differences |
| CI Environment | Pseudo-terminals            | Limited terminal emulation      |

# Implementation Guidance

## Technology Recommendations

| Testing Component      | Simple Option                                | Advanced Option                               |
|------------------------|--|---|
| Unit Testing Framework | Simple assert macros with custom test runner | Unity test framework or CTest                 |
| Mock Framework         | Manual mocks with function pointers          | CMock or similar mocking library              |
| PTY Testing            | Basic openpty() usage                        | Expect-style scripting framework              |
| Performance Testing    | Simple timing with clock_gettime()           | Profiling tools like valgrind, perf           |
| CI Integration         | Shell scripts with exit codes                | GitHub Actions or Jenkins with test reporting |

## Recommended File Structure

```
project-root/
  tests/
    unit/
      test_text_buffer.c      ← Unit tests for text buffer operations
      test_terminal_manager.c ← Unit tests for terminal management
      test_input_handler.c    ← Unit tests for input processing
      test_undo_system.c     ← Unit tests for undo functionality
      test_search_system.c   ← Unit tests for search functionality
    integration/
      test_editing_workflow.c ← Integration tests for complete editing scenarios
      test_file_operations.c  ← Integration tests for file I/O
      test_display_rendering.c ← Integration tests for screen rendering
    milestones/
      milestone_1_verification.c ← Automated verification for raw mode
      milestone_2_verification.c ← Automated verification for screen refresh
      milestone_3_verification.c ← Automated verification for file viewing
      milestone_4_verification.c ← Automated verification for text editing
      milestone_5_verification.c ← Automated verification for save/undo
      milestone_6_verification.c ← Automated verification for search
      milestone_7_verification.c ← Automated verification for syntax highlighting
    framework/
      test_framework.h         ← Common test utilities and macros
      mock_terminal.c          ← Mock terminal for testing
      input_simulator.c        ← Input sequence simulation
      pty_helper.c              ← Pseudo-terminal testing utilities
    scripts/
      run_all_tests.sh          ← Script to execute all test suites
      milestone_check.sh        ← Script to verify milestone completion
  Makefile                  ← Build configuration including test targets
```

## **Test Framework Infrastructure**

**Complete Test Framework Implementation:**

```
// tests/framework/test_framework.h

#ifndef TEST_FRAMEWORK_H

#define TEST_FRAMEWORK_H


#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <assert.h>

#include <time.h>

// Test result tracking

typedef struct {

    int tests_run;

    int tests_passed;

    int tests_failed;

    char current_test_name[256];

    clock_t start_time;

} TestResults;

extern TestResults global_test_results;

// Core test macros

#define TEST_START(name) \
do { \
    strcpy(global_test_results.current_test_name, name); \
    global_test_results.start_time = clock(); \
    printf("Running test: %s... ", name); \
    fflush(stdout); \
} while(0)
```

C

```
#define TEST_END() \  
do { \  
    double elapsed = ((double)(clock() - global_test_results.start_time)) /  
CLOCKS_PER_SEC; \  
  
    printf("PASSED (%.3fs)\n", elapsed); \  
    global_test_results.tests_passed++; \  
  
    global_test_results.tests_run++; \  
} while(0)  
  
#define TEST_FAIL(msg) \  
do { \  
    printf("FAILED: %s\n", msg); \  
    global_test_results.tests_failed++; \  
    global_test_results.tests_run++; \  
    return 0; \  
} while(0)  
  
#define ASSERT_EQ(expected, actual) \  
do { \  
    if ((expected) != (actual)) { \  
        char error_msg[512]; \  
        snprintf(error_msg, sizeof(error_msg), "Expected %d, got %d at %s:%d", \  
            (int)(expected), (int)(actual), __FILE__, __LINE__); \  
        TEST_FAIL(error_msg); \  
    } \  
} while(0)  
  
#define ASSERT_STR_EQ(expected, actual) \  
do { \  
    if (strcmp(expected, actual) != 0) { \  
        char error_msg[512]; \  
        snprintf(error_msg, sizeof(error_msg), "Expected '%s', got '%s' at %s:%d", \  
            expected, actual, __FILE__, __LINE__); \  
        TEST_FAIL(error_msg); \  
    } \  
} while(0)
```

```
do { \
    if (strcmp(expected, actual) != 0) { \
        char error_msg[512]; \
        sprintf(error_msg, sizeof(error_msg), "Expected '%s', got '%s' at %s:%d", \
            expected, actual, __FILE__, __LINE__); \
        TEST_FAIL(error_msg); \
    } \
} while(0)

#define ASSERT_NULL(ptr) \
do { \
    if ((ptr) != NULL) { \
        char error_msg[512]; \
        sprintf(error_msg, sizeof(error_msg), "Expected NULL, got %p at %s:%d", \
            ptr, __FILE__, __LINE__); \
        TEST_FAIL(error_msg); \
    } \
} while(0)

#define ASSERT_NOT_NULL(ptr) \
do { \
    if ((ptr) == NULL) { \
        char error_msg[512]; \
        sprintf(error_msg, sizeof(error_msg), "Expected non-NUL pointer at %s:%d", \
            __FILE__, __LINE__); \
        TEST_FAIL(error_msg); \
    } \
} while(0)
```

```
// Test suite management

#define RUN_TEST(test_func) \
do { \
    if (test_func()) { \
        /* Test passed, already counted */ \
    } else { \
        /* Test failed, already counted */ \
    } \
} while(0)

#define TEST_SUMMARY() \
do { \
    printf("\n==== Test Results ====\n"); \
    printf("Tests run: %d\n", global_test_results.tests_run); \
    printf("Tests passed: %d\n", global_test_results.tests_passed); \
    printf("Tests failed: %d\n", global_test_results.tests_failed); \
    if (global_test_results.tests_failed == 0) { \
        printf("All tests PASSED!\n"); \
    } else { \
        printf("Some tests FAILED!\n"); \
    } \
} while(0)

// Initialize test framework

void test_framework_init(void);

// Memory leak detection helpers

void* test_malloc(size_t size);
```

```
void test_free(void* ptr);

void check_memory_leaks(void);

#endif // TEST_FRAMEWORK_H
```

**Mock Terminal Implementation:**

```
// tests/framework/mock_terminal.c

#include "mock_terminal.h"

#include <stdlib.h>

#include <string.h>

MockTerminal* mock_terminal_create(int rows, int cols) {

    MockTerminal* mock = malloc(sizeof(MockTerminal));

    if (!mock) return NULL;

    mock->output_buffer = malloc(4096);

    mock->output_capacity = 4096;

    mock->output_length = 0;

    mock->cursor_row = 1;

    mock->cursor_col = 1;

    mock->screen_rows = rows;

    mock->screen_cols = cols;

    mock->raw_mode_enabled = 0;

    return mock;
}

void mock_terminal_destroy(MockTerminal* mock) {

    if (mock) {

        free(mock->output_buffer);

        free(mock);
    }
}
```

C

```
int mock_terminal_write(MockTerminal* mock, const char* data, size_t len) {

    if (!mock || !data) return -1;

    // Expand buffer if needed

    if (mock->output_length + len >= mock->output_capacity) {

        size_t new_capacity = mock->output_capacity * 2;

        while (new_capacity < mock->output_length + len + 1) {

            new_capacity *= 2;

        }

        char* new_buffer = realloc(mock->output_buffer, new_capacity);

        if (!new_buffer) return -1;

        mock->output_buffer = new_buffer;

        mock->output_capacity = new_capacity;

    }

    memcpy(mock->output_buffer + mock->output_length, data, len);

    mock->output_length += len;

    mock->output_buffer[mock->output_length] = '\0';

    // Parse ANSI sequences to track cursor position

    mock_terminal_parse_output(mock, data, len);

    return len;

}
```

```
void mock_terminal_parse_output(MockTerminal* mock, const char* data, size_t len) {

    // TODO: Parse ANSI escape sequences and update mock terminal state

    // This should handle cursor positioning, screen clearing, etc.

    // Example: \x1b[5;10H should set cursor to row 5, column 10

}

int mock_terminal_verify_sequence(MockTerminal* mock, const char* expected) {

    if (!mock || !expected) return 0;

    size_t expected_len = strlen(expected);

    if (mock->output_length < expected_len) return 0;

    // Check if expected sequence appears in output

    for (size_t i = 0; i <= mock->output_length - expected_len; i++) {

        if (memcmp(mock->output_buffer + i, expected, expected_len) == 0) {

            return 1; // Found

        }

    }

    return 0; // Not found

}

void mock_terminal_clear_output(MockTerminal* mock) {

    if (mock) {

        mock->output_length = 0;

        mock->output_buffer[0] = '\0';

    }

}
```

}

## Core Logic Skeleton for Milestone Tests

### Milestone 1 Verification Skeleton:

```
// tests/milestones/milestone_1_verification.c

#include "../framework/test_framework.h"

#include "../framework/mock_terminal.h"

#include "../../src/terminal_manager.h"

#include "../../src/input_handler.h"

int test_raw_mode_setup() {

    TEST_START("raw_mode_setup");

    // TODO 1: Call enable_raw_mode() and verify it returns success

    // TODO 2: Check that terminal echo is disabled using mock terminal

    // TODO 3: Verify that canonical mode is disabled

    // TODO 4: Call disable_raw_mode() and verify terminal is restored

    // Hint: Use mock terminal to capture termios changes

    TEST_END();

    return 1;
}

int test_keypress_reading() {

    TEST_START("keypress_reading");

    // TODO 1: Set up input simulation with known characters

    // TODO 2: Call editor_read_key() for each character

    // TODO 3: Verify that correct key codes are returned

    // TODO 4: Test with special characters like ESC, arrow keys

    // Hint: Create input simulator that feeds bytes to stdin
```

```
TEST_END();

return 1;

}

int test_escape_sequence_parsing() {

TEST_START("escape_sequence_parsing");

// TODO 1: Simulate arrow key escape sequences

// TODO 2: Call editor_read_key() and verify ARROW_* constants returned

// TODO 3: Test other special keys (HOME, END, DELETE, PAGE_UP, PAGE_DOWN)

// TODO 4: Test incomplete sequences and timeout behavior

// Hint: Use timeout-based parsing to distinguish ESC from sequences


TEST_END();

return 1;

}

int test_terminal_cleanup() {

TEST_START("terminal_cleanup");

// TODO 1: Enable raw mode and simulate program crash

// TODO 2: Verify signal handlers restore terminal state

// TODO 3: Test normal exit path restoration

// TODO 4: Test multiple enable/disable cycles

// Hint: Use signal simulation to test cleanup paths


TEST_END();

return 1;
```

```

    }

int main() {
    test_framework_init();

    printf("==== Milestone 1 Verification ====\n");
    printf("Testing raw mode and input handling...\n\n");

    RUN_TEST(test_raw_mode_setup);
    RUN_TEST(test_keypress_reading);
    RUN_TEST(test_escape_sequence_parsing);
    RUN_TEST(test_terminal_cleanup);

    TEST_SUMMARY();
    check_memory_leaks();

    return global_test_results.tests_failed > 0 ? 1 : 0;
}

```

## Language-Specific Testing Hints

### C-Specific Testing Considerations:

- Use `valgrind --tool=memcheck` to detect memory leaks and buffer overflows during testing
- Test signal handling with `kill()` function to simulate crashes and interruptions
- Use `pipe()` or `socketpair()` to create controlled input/output streams for testing
- Compile tests with `-g -O0` for better debugging and `-fsanitize=address` for runtime error detection
- Use `tcgetattr()` and `tcsetattr()` to verify terminal state changes in tests

### Debugging Integration:

```
// Debugging helper for test development C

#ifndef DEBUG_TESTS

#define DEBUG_PRINT(fmt, ...) \
    printf("[DEBUG] %s:%d: " fmt "\n", __FILE__, __LINE__, ##__VA_ARGS__)

#else

#define DEBUG_PRINT(fmt, ...)

#endif


// Memory debugging integration

void test_framework_init() {

    global_test_results.tests_run = 0;

    global_test_results.tests_passed = 0;

    global_test_results.tests_failed = 0;

    if (ENABLE_MEMORY_TRACKING) {
        init_memory_tracking();
    }
}
```

## Milestone Checkpoint Scripts

Complete Milestone Verification Script:

BASH

```
#!/bin/bash

# tests/scripts/milestone_check.sh

set -e # Exit on any error

MILESTONE=$1

if [ -z "$MILESTONE" ]; then
    echo "Usage: $0 <milestone_number>"
    echo "Example: $0 1"
    exit 1
fi

echo "==== Milestone $MILESTONE Verification ==="
echo "Compiling and running tests..."

# Build test for specified milestone
make clean
make test_milestone_$MILESTONE

if [ $? -ne 0 ]; then
    echo "✖ Compilation failed for milestone $MILESTONE"
    exit 1
fi

# Run automated tests
echo "Running automated tests..."
./tests/bin/milestone_${MILESTONE}_verification

if [ $? -ne 0 ]; then
    echo "✖ Automated tests failed for milestone $MILESTONE"
```

```
exit 1

fi

# Run manual verification prompts

case $MILESTONE in

1)

echo "✅ Automated tests passed!"

echo ""

echo "Manual verification checklist:"

echo "1. Run: ./editor"

echo "2. Type characters - they should NOT echo automatically"

echo "3. Press arrow keys - they should be recognized (no ^[[A output)"

echo "4. Press Ctrl+C - terminal should restore to normal mode"

echo ""

echo "Does the editor behave correctly? (y/n)"

read -r response

if [[ ! "$response" =~ ^[Yy]$ ]]; then

    echo "✗ Manual verification failed"

    exit 1

fi

;

2)

echo "✅ Automated tests passed!"

echo ""

echo "Manual verification checklist:"

echo "1. Run: ./editor"

echo "2. Screen should clear and show status bar"
```

```

echo "3. Resize terminal - editor should adapt"

echo "4. No flickering during screen updates"

echo ""

echo "Does the screen rendering work correctly? (y/n)"

read -r response

if [[ ! "$response" =~ ^[Yy]$ ]]; then

    echo "✗ Manual verification failed"

    exit 1

fi

;;

# Add cases for other milestones...

esac

echo "✓ Milestone $MILESTONE verification complete!"

```

## Debugging Tips for Test Failures

| Test Failure Symptom                  | Likely Cause                     | Diagnosis Steps                        | Fix Strategy   |
|---------------------------------------|----------------------------------|--|--|
| Tests hang indefinitely               | Blocking on terminal input       | Check for unread input in buffers      | Use non-blocking reads or input simulation             |
| Segmentation faults in tests          | Memory management errors         | Run with valgrind, check buffer bounds | Add bounds checking, fix memory leaks                  |
| ANSI sequence verification fails      | Incorrect escape code generation | Print actual vs expected sequences     | Debug sequence formatting, check parameters            |
| Terminal left in wrong state          | Cleanup not called on test exit  | Check signal handling and exit paths   | Ensure cleanup in all exit scenarios                   |
| Performance tests fail intermittently | System load affects timing       | Use relative performance metrics       | Set reasonable thresholds, test on consistent hardware |

# Debugging Guide

**Milestone(s):** All milestones (debugging techniques are essential throughout the entire development process)

Think of debugging a terminal text editor as being a detective investigating a crime scene where the evidence keeps disappearing. Unlike GUI applications where you can inspect the interface state visually, terminal applications manipulate invisible state—terminal modes, escape sequences, and raw input streams—that can only be observed through their effects. The challenge is compounded by the fact that debugging tools themselves often interfere with the very terminal state you're trying to investigate.

Terminal text editors fail in spectacular ways that can leave your terminal in an unusable state, with echo disabled, cursor hidden, or screen clearing broken. The key to successful debugging is understanding the layered nature of failures: terminal state corruption at the bottom, rendering issues in the middle, and text manipulation bugs at the top. Each layer requires different diagnostic approaches and recovery strategies.

This debugging guide provides systematic approaches to the most common failure modes you'll encounter while building your text editor. Each category includes concrete symptoms you can observe, systematic diagnostic procedures, and specific fixes rather than generic advice.

## Terminal State Issues

Terminal state problems are the most dangerous category of bugs because they can render your development environment unusable. These issues stem from the fundamental challenge of managing the transition between canonical mode (normal terminal behavior) and raw mode (character-by-character input processing).

### Raw Mode Setup and Restoration Failures

The most common terminal state issue occurs when raw mode is enabled but never properly restored, leaving your terminal in an unusable state where keystrokes aren't echoed and line editing doesn't work.

**Understanding the Problem:** When you enable raw mode using `tcsetattr` to modify the `termios` structure, you're fundamentally changing how the terminal driver processes input. The terminal stops buffering input until newline characters, stops echoing typed characters to the screen, and disables signal generation for control characters like Ctrl+C. If your program exits without calling `disable_raw_mode()`, these changes persist, leaving you with a terminal that appears broken.

### Common Failure Scenarios:

| Symptom                           | Cause  | Immediate Recovery                              | Prevention   |
|-----------------------------------|--|---|--|
| No echo after editor crash        | <code>disable_raw_mode()</code> never called | Type <code>reset</code> and press Enter blindly | Install signal handlers before enabling raw mode             |
| Ctrl+C doesn't work in editor     | <code>ISIG</code> flag disabled in raw mode  | Kill process from another terminal              | Handle <code>SIGTERM</code> and <code>SIGINT</code> properly |
| Terminal shows garbage after exit | Screen not cleared on exit                   | <code>printf("\x1b[2J\x1b[H")</code>            | Always clear screen in cleanup                               |
| Can't see typed characters        | Echo disabled but raw mode not enabled       | <code>stty echo</code> command                  | Check return value of <code>tcsetattr</code>                 |

The most insidious version of this problem occurs when your program crashes during raw mode setup—after disabling echo but before installing signal handlers. This creates a situation where Ctrl+C doesn't work to kill the program, and you can't see what you're typing to fix it.

**Diagnostic Approach:** To debug raw mode issues systematically, create a minimal test program that only handles terminal mode switching:

1. Add logging to both `enable_raw_mode()` and `disable_raw_mode()` functions that writes to a separate log file, not stdout
2. Use `tcgetattr()` to save the original terminal attributes and log their key field values (`c_lflag`, `c_iflag`)
3. After each `tcsetattr()` call, immediately call `tcgetattr()` again to verify the changes took effect
4. Test signal handler installation by sending `SIGTERM` from another shell and verifying cleanup occurs
5. Use `strace` or similar tools to observe actual system calls and their return values

**⚠ Pitfall: Testing Raw Mode in Development Environment** Many developers test their terminal code in their primary development terminal, which means a crash can lock them out of their development environment. Instead, always test terminal mode changes in a separate terminal window or use a virtual terminal. Even better, develop a wrapper script that runs your editor in a timeout, automatically killing it if it hangs.

**Signal Handler Implementation:** The critical insight for preventing terminal state corruption is that signal handlers must be installed *before* enabling raw mode, and they must be async-signal-safe. This means you can't use standard library functions like `printf()` or `malloc()` inside signal handlers.

Here's the correct sequence for robust terminal state management:

1. Save original terminal attributes using `tcgetattr()` into a global variable

2. Install signal handlers for `SIGTERM`, `SIGINT`, `SIGSEGV`, and `SIGABRT` that call an async-signal-safe cleanup function
3. Enable raw mode by modifying the saved attributes and calling `tcsetattr()`
4. Set up an `atexit()` handler as a backup cleanup mechanism
5. In your main cleanup function, first restore terminal attributes, then clear the screen, then show the cursor

**Async-Signal-Safe Cleanup:** Your `emergency_cleanup()` function must only use async-signal-safe functions. This severely limits what you can do, but the essential operations are still possible:

| Safe Functions           | Unsafe Functions      | Safe Alternative   |
|--------------------------|-----------------------|--|
| <code>write()</code>     | <code>printf()</code> | <code>write(STDOUT_FILENO, msg, strlen(msg))</code>            |
| <code>tcsetattr()</code> | <code>fflush()</code> | Not needed for direct file descriptor writes                   |
| <code>_exit()</code>     | <code>exit()</code>   | <code>_exit()</code> doesn't call atexit handlers but is safe  |
| <code>kill()</code>      | <code>abort()</code>  | <code>kill(getpid(), SIGKILL)</code> for immediate termination |

## Escape Sequence Parsing Problems

ANSI escape sequences are multi-byte sequences that begin with the escape character (ASCII 27) followed by additional characters that specify the action. The challenge is distinguishing between a user pressing the Escape key and the beginning of an escape sequence for special keys like arrows.

**The Timing Problem:** When a user presses the Escape key, your program receives byte 27. When they press an arrow key, your program receives bytes 27, 91, followed by A/B/C/D for up/down/left/right. The only way to distinguish these cases is by timing—if additional bytes arrive quickly after the escape character, it's likely an escape sequence.

### Common Escape Sequence Issues:

| Symptom                       | Root Cause                         | Debug Approach                                | Solution   |
|-------------------------------|------------------------------------|---|--|
| Arrow keys print letters      | Escape sequences not parsed        | Log raw bytes received                        | Implement escape sequence state machine                      |
| Escape key doesn't work       | Timeout too short for detection    | Increase timeout, test on different terminals | Use 100ms timeout minimum                                    |
| Arrow keys work sometimes     | Race condition in parsing          | Log timing between bytes                      | Use <code>select()</code> or <code>poll()</code> for timeout |
| Function keys produce garbage | Terminal sends different sequences | Log actual sequences per terminal             | Build lookup table of terminal variations                    |

The most robust approach to escape sequence parsing is implementing a state machine that tracks the current parsing state and handles partial sequences gracefully.

### Escape Sequence State Machine:

| Current State | Input Byte | Next State   | Action                  |
|---------------|------------|--------------|-------------------------|
| NORMAL        | 27 (ESC)   | ESCAPE_START | Start timeout timer     |
| ESCAPE_START  | 91 ('[')   | CSI_SEQUENCE | Continue parsing        |
| ESCAPE_START  | Any other  | NORMAL       | Return ESC_KEY          |
| CSI_SEQUENCE  | 'A'        | NORMAL       | Return ARROW_UP         |
| CSI_SEQUENCE  | 'B'        | NORMAL       | Return ARROW_DOWN       |
| Any state     | Timeout    | NORMAL       | Return accumulated keys |

**⚠ Pitfall: Terminal Emulator Variations** Different terminal emulators send different escape sequences for the same keys. For example, the Home key might send `\x1b[H`, `\x1b[1~`, or `\x1b0H` depending on the terminal. Don't assume all terminals behave identically—build a comprehensive mapping table or use a library like ncurses that handles these variations.

### Screen Size Detection Issues

Terminal applications must adapt to different screen sizes and handle resize events. The `get_window_size()` function typically uses the `TIOCGWINSZ` ioctl to query the terminal dimensions, but this can fail in certain environments.

### Window Size Detection Failure Modes:

| Environment       | TIOCGWINSZ Result         | Fallback Approach                                     | Implementation Notes                        |
|-------------------|---------------------------|---|---|
| SSH without PTY   | Returns 0x0               | Query using cursor positioning                        | Send <code>\x1b[999C\x1b[999B\x1b[6n</code> |
| Screen/tmux       | May return incorrect size | Check <code>\$LINES</code> and <code>\$COLUMNS</code> | Environment variables override ioctl        |
| IDE terminals     | Size changes not reported | Handle <code>SIGWINCH</code> signal                   | Resize events come as signals               |
| Docker containers | No terminal attached      | Default to 80x24                                      | Graceful degradation                        |

The cursor positioning fallback technique works by sending escape sequences to move the cursor to position 999,999 (far beyond any reasonable screen size), then querying the cursor position to determine the actual screen boundaries.

**Handling SIGWINCH:** Terminal resize events are delivered as `SIGWINCH` signals. Your editor needs to handle these by re-querying the screen size and triggering a full screen refresh. However, signal handlers have the same async-signal-safe restrictions mentioned earlier.

The proper approach is to set a flag in the signal handler and check that flag in your main event loop:

1. Install `SIGWINCH` handler that sets `volatile sig_atomic_t window_size_changed = 1`
2. In main event loop, check flag before processing each key
3. If flag is set, call `get_window_size()` and `editor_refresh_screen()`
4. Reset flag to zero after handling resize

## Rendering and Display Problems

Rendering problems manifest as visual artifacts on the screen—flickering, incorrect cursor positioning, missing text, or garbled output. These issues stem from the complex interaction between your editor's internal state, the viewport calculations, and the ANSI escape sequences sent to the terminal.

### Screen Flicker and Update Issues

Screen flicker occurs when the terminal displays partial updates during the rendering process. The human eye perceives this as rapid flashing or text that appears to jump around. Understanding the cause requires thinking about how terminal display works at the byte level.

**The Root Cause of Flicker:** Every call to `write()` or `printf()` that outputs to the terminal can cause immediate visual changes. If you clear the screen, then write line 1, then write line 2, and so on, the user sees each intermediate state. Modern terminals can display thousands of updates per second, so even fast code can produce visible flicker.

### Anti-Flicker Strategies:

| Technique        | How It Works                      | Implementation  | Trade-offs                   |
|------------------|-----------------------------------|---|------------------------------|
| Frame buffering  | Accumulate all output, write once | <code>FrameBuffer</code> structure                          | Uses more memory             |
| Cursor hiding    | Hide cursor during updates        | <code>\x1b[?25l</code> before, <code>\x1b[?25h</code> after | Prevents cursor flicker only |
| Minimal updates  | Only redraw changed regions       | Track dirty regions   | Complex change detection     |
| Double buffering | Compare old vs new screen state   | Keep previous frame copy                                    | Memory intensive             |

The frame buffering approach is the most reliable anti-flicker technique. Instead of writing directly to the terminal, you accumulate all output in a buffer, then send the entire frame in a single `write()` call.

**Frame Buffer Implementation Strategy:** Your `FrameBuffer` structure should handle dynamic growth efficiently since you don't know the final output size in advance. The key insight is that most terminal output is predictable in size—you know roughly how many characters per line and how many lines you're rendering.

Start with a buffer sized for the full screen plus overhead for escape sequences. If the buffer fills up, resize it exponentially (double the size) to avoid frequent reallocations.

**⚠ Pitfall: Escape Sequence Overhead** When calculating buffer size, don't forget that ANSI escape sequences add significant overhead. A simple line of text might require cursor positioning (`\x1b[row;colH]`), color codes (`\x1b[31m`), and reset sequences (`\x1b[0m`). Budget at least 20-30 extra bytes per line for escape sequences.

**Cursor Positioning Errors:** Terminal cursor positioning uses 1-based coordinates (row 1, column 1 is the top-left), while your editor's internal coordinates are likely 0-based. This off-by-one error is extremely common and causes text to appear shifted by one position.

| Common Mistake                           | Symptom                           | Correct Approach                                  |
|--|-----------------------------------|---|
| Using 0-based coords in escape sequences | Text appears one row/col off      | Add 1 when generating <code>\x1b[row;colH]</code> |
| Mixing coordinate systems                | Cursor jumps around randomly      | Use conversion functions consistently             |
| Forgetting screen boundaries             | Cursor moves outside visible area | Clamp coordinates to valid ranges                 |

## Viewport and Scrolling Issues

The viewport represents the portion of your text buffer that's currently visible on screen. Scrolling bugs manifest as text that jumps around unexpectedly, cursor disappearing off-screen, or content that doesn't scroll when it should.

**Understanding Viewport Coordinates:** Your editor maintains several coordinate systems simultaneously: logical coordinates (position in the full document), screen coordinates (position on the terminal screen), and render coordinates (accounting for tab expansion and line wrapping). Bugs occur when these coordinate systems get out of sync.

### Viewport State Variables:

| Variable            | Purpose                | Valid Range                            | Common Bug                 |
|---------------------|------------------------|--|----------------------------|
| <code>rowoff</code> | First visible row      | 0 to <code>numrows - screenrows</code> | Goes negative              |
| <code>coloff</code> | First visible column   | 0 to max line length                   | Not updated on cursor move |
| <code>cx</code>     | Cursor column in file  | 0 to line length                       | Exceeds line length        |
| <code>cy</code>     | Cursor row in file     | 0 to <code>numrows - 1</code>          | Not clamped to valid range |
| <code>rx</code>     | Rendered cursor column | 0 to rendered line length              | Tabs not handled           |

The most common scrolling bug is failing to update the viewport offsets when the cursor moves outside the visible area. The `editor_scroll()` function must run before every screen refresh to ensure the cursor remains visible.

**Scrolling Logic Algorithm:** The scrolling algorithm has four distinct cases to handle:

1. **Cursor above visible area:** Set `rowoff = cy` to bring cursor to top of screen
2. **Cursor below visible area:** Set `rowoff = cy - screenrows + 1` to bring cursor to bottom of screen
3. **Cursor left of visible area:** Set `coloff = rx` to bring cursor to left edge
4. **Cursor right of visible area:** Set `coloff = rx - screencols + 1` to bring cursor to right edge

Note that horizontal scrolling uses `rx` (rendered position) rather than `cx` (logical position) because tabs expand to multiple columns.

**⚠ Pitfall: Scrolling Boundary Conditions** When the file has fewer lines than the screen height, or lines shorter than screen width, the scrolling offsets can become invalid. Always clamp offsets to valid ranges: `rowoff` cannot exceed `numrows - screenrows` and cannot go below 0. Similarly, `coloff` must be clamped based on the current line's rendered length.

## Status Bar and UI Element Display

The status bar provides crucial feedback about editor state, but it's also a common source of rendering bugs. Status bar issues typically involve incorrect positioning, content overflow, or conflicts with the main text area.

**Status Bar Positioning:** The status bar appears at the bottom of the screen, typically at row `screenrows`. However, coordinate calculations must account for the fact that the main text area ends at row `screenrows - 1`, leaving the last row for the status bar.

## Status Bar Content Management:

| Element        | Content             | Max Length    | Overflow Handling   |
|----------------|---------------------|---------------|---------------------|
| Filename       | Current file path   | 20 characters | Truncate with "..." |
| Position       | "row,col"           | 10 characters | Always fits         |
| Line count     | "X lines"           | 15 characters | Abbreviate as "Xln" |
| Dirty flag     | "**" if modified    | 1 character   | Always show         |
| Mode indicator | Search, insert, etc | 10 characters | Truncate mode name  |

The challenge is fitting all status information into the available screen width while ensuring the most important information (filename and position) is always visible.

**Dynamic Status Messages:** Some status messages are temporary (like "File saved" or error messages) while others are permanent (filename and cursor position). Temporary messages should override permanent ones for a few seconds, then revert.

Use `statusmsg_time` to track when temporary messages expire. In your render function, check if `time(NULL) - statusmsg_time > 5` to determine whether to show the temporary message or revert to the standard status bar.

## Text Editing Logic Bugs

Text editing bugs are the most subtle category because they involve the complex interaction between cursor movement, buffer modification, and coordinate system management. These bugs often manifest as data corruption, cursor jumping to wrong positions, or operations affecting the wrong part of the text.

### Buffer Manipulation Edge Cases

Text buffer operations must handle numerous edge cases that don't occur in normal usage but can cause crashes or data loss when they do occur.

### Character Insertion Edge Cases:

| Scenario                       | Expected Behavior                     | Common Bug             | Correct Handling               |
|--------------------------------|---------------------------------------|------------------------|--------------------------------|
| Insert at end of line          | Character appends normally            | Buffer overflow        | Check/resize before insertion  |
| Insert in empty file           | Creates first character of first line | Null pointer access    | Ensure at least one row exists |
| Insert beyond line end         | Cursor at end, character appends      | Cursor jumps           | Clamp cursor to line length    |
| Insert with cursor at column 0 | Character inserted at beginning       | Off-by-one in position | Handle 0-index correctly       |

The most dangerous buffer manipulation bug is failing to resize the `chars` array when inserting characters. If your `erow.chars` array was allocated with `malloc(10)` and already contains 9 characters, inserting one more character requires calling `realloc()` to expand the array.

**Memory Management in Buffer Operations:** Every text modification operation potentially requires memory reallocation. The key insight is that these operations come in patterns—users typically insert or delete multiple characters in sequence. Optimize for these patterns by growing buffers in chunks rather than one character at a time.

### Buffer Resizing Strategy:

| Operation   | Current Size         | New Size                  | Rationale                         |
|-------------|----------------------|---------------------------|-----------------------------------|
| Insert char | <code>size</code>    | <code>size * 1.5</code>   | Amortized O(1) insertion          |
| Insert line | <code>numrows</code> | <code>numrows + 10</code> | Lines added less frequently       |
| Delete char | Any                  | No resize                 | Keep memory for future insertions |
| Delete line | Any                  | No resize until 50% waste | Avoid thrashing                   |

**⚠ Pitfall: Cursor Consistency After Buffer Changes** After any buffer modification, the cursor position might become invalid. For example, if you delete a line that contained the cursor, the cursor row must be adjusted. Similarly, if you delete characters from the current line, the cursor column might extend beyond the new line length.

Always validate and correct cursor position after buffer modifications:

1. Ensure `cy < numrows` (cursor row within file)
2. Ensure `cx <= row[cy].size` (cursor column within line)
3. Recalculate `rx` based on new `cx` and line content
4. Update viewport offsets if cursor moved outside visible area

## Line Operations and Cursor Management

Line-level operations (inserting newlines, joining lines, deleting lines) are more complex than character operations because they affect the overall structure of the text buffer.

**Newline Insertion (Line Splitting):** When the user presses Enter, the current line must be split at the cursor position. This involves:

1. Create a new `erow` structure for the second half of the line
2. Copy characters from cursor position to end of line into the new row
3. Truncate the original line at the cursor position
4. Insert the new row into the `editor_config.row` array at position `cy + 1`
5. Update `numrows` and move cursor to beginning of new line

**Line Deletion and Joining:** Backspace at the beginning of a line should join the current line with the previous line. This operation is essentially the reverse of line splitting:

1. Save content of current line
2. Append current line content to previous line (requires resizing previous line's `chars` array)
3. Remove current line from the `row` array
4. Update `numrows` and move cursor to join position in previous line

**Array Management for Line Operations:** Since lines are stored in a dynamic array (`erow* row`), inserting or deleting lines requires shifting other elements. For insertion at position `at`:

1. Resize the `row` array if necessary
2. Use `memmove()` to shift elements `[at..numrows-1]` to positions `[at+1..numrows]`
3. Initialize the new element at position `at`

For deletion, reverse the process: save/free the deleted element, then shift remaining elements down.

**⚠ Pitfall: Using `memcpy` Instead of `memmove`** When shifting array elements, the source and destination ranges overlap. `memcpy()` is undefined for overlapping ranges and may corrupt data. Always use `memmove()` for overlapping memory operations.

## Coordinate System Synchronization

Text editors maintain multiple coordinate systems that must stay synchronized: logical coordinates (position in file), visual coordinates (position on screen accounting for tabs), and screen coordinates (terminal row/column for cursor positioning).

**Coordinate Conversion Functions:** The `cx_to_rx()` function converts logical cursor position to visual position by expanding tabs. The inverse function `rx_to(cx)` converts visual position back to logical position. These functions are critical for horizontal scrolling and mouse support.

**Tab Expansion Logic:** Tabs expand to the next multiple of `TAB_SIZE`. If `TAB_SIZE` is 8, a tab at column 0 expands to column 8, a tab at column 1 expands to column 8, and a tab at column 7 expands to column 8. The formula is:

```
next_tab_stop = (current_column / TAB_SIZE + 1) * TAB_SIZE
tab_width = next_tab_stop - current_column
```

## Common Coordinate System Bugs:

| Bug Type                            | Symptom                                    | Cause  | Solution   |
|-------------------------------------|--|--|--|
| Cursor jumps on tab navigation      | Cursor moves wrong distance                | Using <code>cx</code> instead of <code>rx</code> for display | Use <code>rx</code> for horizontal scrolling                           |
| Home/End keys go to wrong position  | Cursor at column 8 instead of 0            | Confusing logical vs visual coordinates                      | Home sets <code>cx=0</code> , End sets <code>cx=line.size</code>       |
| Mouse clicks select wrong character | Click selects different char than expected | Mouse coordinates not converted                              | Convert mouse <code>rx</code> to <code>cx</code> before setting cursor |
| Horizontal scroll broken            | Text scrolls by wrong amount               | Scroll offset using wrong coordinate system                  | Use <code>rx</code> for <code>coloff</code> calculations               |

**Debugging Coordinate Issues:** When debugging coordinate problems, log all three coordinate systems simultaneously. Create a debug function that prints current values of `cx`, `cy`, `rx`, `rowoff`, `coloff`, and screen cursor position. Call this function after every cursor movement operation to track when coordinates get out of sync.

The most effective debugging approach is to add assertions that validate coordinate consistency:

1. Assert that `cx` is within the current line bounds: `assert(cx >= 0 && cx <= row[cy].size)`
2. Assert that `rx` corresponds to `cx`: `assert(rx == cx_to_rx(&row[cy], cx))`
3. Assert that screen coordinates are valid: `assert(cy - rowoff >= 0 && cy - rowoff < screenrows)`

## Implementation Guidance

The debugging strategies described above require specific tools and techniques that work effectively in terminal environments. This section provides concrete implementations and debugging workflows.

## Technology Recommendations

| Debugging Category | Simple Approach                             | Advanced Approach                                      |
|--------------------|---|--|
| Terminal State     | Manual testing + <code>stty</code> commands | Automated PTY testing with <code>script</code> command |
| Rendering Issues   | Log file debugging                          | Screen capture comparison tools                        |
| Memory Problems    | <code>valgrind</code> with suppressions     | AddressSanitizer with custom allocators                |
| Timing Issues      | <code>strace</code> system call tracing     | Custom timing instrumentation                          |
| Input Parsing      | Hex dump logging                            | State machine visualization                            |

## Recommended File Structure for Debugging

```
project-root/  
  C  
  src/  
    editor.c          ← main editor logic  
    terminal.c       ← terminal state management  
    debug.c          ← debugging utilities (this is new)  
    debug.h          ← debugging macros and types  
  debug/  
    test_terminal.c  ← terminal state test harness  
    mock_terminal.c  ← mock terminal for unit tests  
    debug_log_parser.py  ← script to analyze debug logs  
  logs/  
    editor_debug.log   ← main debug output  
    terminal_state.log  ← terminal state transitions  
    memory_trace.log    ← memory allocation tracking
```

The `debug.c` file should contain utilities that can be compiled out in release builds but provide detailed instrumentation during development.

## Debug Infrastructure Starter Code

```
// debug.h - Complete debugging infrastructure C

#ifndef DEBUG_H

#define DEBUG_H


#include <stdio.h>

#include <time.h>

#include <stdarg.h>

// Debug levels - can be set via environment variable DEBUG_LEVEL

typedef enum {

    DEBUG_OFF = 0,

    DEBUG_ERROR = 1,

    DEBUG_WARN = 2,

    DEBUG_INFO = 3,

    DEBUG_TRACE = 4

} debug_level_t;

// Debug categories - allows filtering debug output by component

#define DEBUG_TERMINAL 0x01

#define DEBUG_INPUT 0x02

#define DEBUG_RENDER 0x04

#define DEBUG_BUFFER 0x08

#define DEBUG_COORDS 0x10

// Global debug state

extern debug_level_t g_debug_level;

extern unsigned int g_debug_categories;

extern FILE* g_debug_file;
```

```
// Initialize debugging system - call this early in main()

void debug_init(const char* log_file);

void debug_cleanup(void);

// Core debugging macros

#define DEBUG_LOG(level, category, format, ...) \
do { \
    if (g_debug_level >= level && (g_debug_categories & category)) { \
        debug_printf(__FILE__, __LINE__, #level, #category, format, ##__VA_ARGS__); \
    } \
} while(0)

// Convenience macros for common debug levels

#define DEBUG_ERROR_LOG(cat, fmt, ...) DEBUG_LOG(DEBUG_ERROR, cat, fmt, ##__VA_ARGS__)

#define DEBUG_INFO_LOG(cat, fmt, ...) DEBUG_LOG(DEBUG_INFO, cat, fmt, ##__VA_ARGS__)

#define DEBUG_TRACE_LOG(cat, fmt, ...) DEBUG_LOG(DEBUG_TRACE, cat, fmt, ##__VA_ARGS__)

// Special macros for coordinate debugging

#define DEBUG_COORDS_STATE() \
DEBUG_INFO_LOG(DEBUG_COORDS, "Coords: cx=%d cy=%d rx=%d rowoff=%d coloff=%d", \
    E.cx, E.cy, E.rx, E.rowoff, E.coloff)

// Terminal state debugging

#define DEBUG_TERMINAL_STATE(msg) \
do { \
    struct termios current; \
    tcgetattr(STDIN_FILENO, &current); \
    DEBUG_INFO_LOG(DEBUG_TERMINAL, "%s: lflag=0x%x iflag=0x%x", \

```

```
    msg, current.c_lflag, current.c_iflag); \n\n} while(0)\n\n// Function prototypes\n\nvoid debug_printf(const char* file, int line, const char* level,\n                  const char* category, const char* format, ...);\n\nvoid debug_dump_buffer_state(void);\n\nvoid debug_dump_screen_contents(void);\n\nvoid debug_hex_dump(const char* label, const unsigned char* data, size_t len);\n\n#endif // DEBUG_H
```

```
// debug.c - Implementation of debugging utilities

#include "debug.h"

#include "editor.h"

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

// Global debug state

debug_level_t g_debug_level = DEBUG_OFF;

unsigned int g_debug_categories = 0xFF; // All categories enabled by default

FILE* g_debug_file = NULL;

void debug_init(const char* log_file) {

    // Check environment variables for debug configuration

    char* level_str = getenv("DEBUG_LEVEL");

    if (level_str) {

        g_debug_level = atoi(level_str);

    }

    char* categories_str = getenv("DEBUG_CATEGORIES");

    if (categories_str) {

        g_debug_categories = strtoul(categories_str, NULL, 16);

    }

    // Open debug log file

    g_debug_file = fopen(log_file, "a");

    if (!g_debug_file) {

        g_debug_file = stderr; // Fallback to stderr
```

C

```
}

DEBUG_INFO_LOG(DEBUG_TERMINAL, "Debug system initialized");

}

void debug_cleanup(void) {

    if (g_debug_file && g_debug_file != stderr) {

        fclose(g_debug_file);

    }

    g_debug_file = NULL;

}

void debug_printf(const char* file, int line, const char* level,
                  const char* category, const char* format, ...) {

    if (!g_debug_file) return;

    // Get current timestamp

    struct timespec ts;

    clock_gettime(CLOCK_MONOTONIC, &ts);

    // Print debug header with timestamp, file, line

    fprintf(g_debug_file, "[%ld.%03ld] %s:%d [%s/%s] ",
            ts.tv_sec, ts.tv_nsec / 1000000, file, line, level, category);

    // Print the actual debug message

    va_list args;

    va_start(args, format);

    vfprintf(g_debug_file, format, args);
```

```
va_end(args);

fprintf(g_debug_file, "\n");

fflush(g_debug_file); // Ensure immediate write for crash debugging

}

void debug_dump_buffer_state(void) {

DEBUG_INFO_LOG(DEBUG_BUFFER, "Buffer state: numrows=%d, dirty=%d",
E.numrows, E.dirty);

for (int i = 0; i < E.numrows && i < 5; i++) { // Only dump first 5 rows
DEBUG_TRACE_LOG(DEBUG_BUFFER, "Row %d: size=%d, chars='%.20s%s'",
i, E.row[i].size, E.row[i].chars,
E.row[i].size > 20 ? "..." : "");
}

}

void debug_hex_dump(const char* label, const unsigned char* data, size_t len) {

DEBUG_INFO_LOG(DEBUG_INPUT, "Hex dump of %s (%zu bytes):", label, len);

char hex_line[80];

for (size_t i = 0; i < len; i += 16) {

int pos = 0;

pos += sprintf(hex_line + pos, "%04zx: ", i);

// Print hex bytes

for (size_t j = 0; j < 16 && i + j < len; j++) {

pos += sprintf(hex_line + pos, "%02x ", data[i + j]);
}
}
}
```

```
    }

    // Pad with spaces if needed

    while (pos < 54) hex_line[pos++] = ' ';

    // Print ASCII representation

    hex_line[pos++] = '|';

    for (size_t j = 0; j < 16 && i + j < len; j++) {

        unsigned char c = data[i + j];

        hex_line[pos++] = (c >= 32 && c < 127) ? c : '.';

    }

    hex_line[pos++] = '|';

    hex_line[pos] = '\0';

    DEBUG_TRACE_LOG(DEBUG_INPUT, "%s", hex_line);

}

}
```

## Core Debug Integration Points

The following skeleton shows where to integrate debug calls into your main editor functions:

```
// In terminal.c - Enable raw mode with debugging C

int enable_raw_mode() {

    DEBUG_TERMINAL_STATE("before enable_raw_mode");

    // TODO: Save original terminal attributes to global variable

    // TODO: Modify attributes to disable ECHO, ICANON, ISIG, IXON

    // TODO: Set VMIN=0, VTIME=1 for timeout-based reading

    // TODO: Apply changes with tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw)

    if /* tcsetattr failed */ {

        DEBUG_ERROR_LOG(DEBUG_TERMINAL, "Failed to enable raw mode: %s", strerror(errno));

        return -1;

    }

    DEBUG_TERMINAL_STATE("after enable_raw_mode");

    DEBUG_INFO_LOG(DEBUG_TERMINAL, "Raw mode enabled successfully");

    return 0;

}

// In editor.c - Character insertion with debugging

void editor_insert_char(int c) {

    DEBUG_TRACE_LOG(DEBUG_BUFFER, "Inserting char %d ('%c') at cx=%d, cy=%d",
                    c, isprint(c) ? c : '?', E.cx, E.cy);

    DEBUG_COORDS_STATE();

    // TODO: Handle case where cy == numrows (cursor past end of file)

    // TODO: Call editor_row_insert_char() to insert character
```

```
// TODO: Update cursor position (E.cx++)

// TODO: Set dirty flag

// TODO: Record undo operation


DEBUG_COORDS_STATE();

DEBUG_TRACE_LOG(DEBUG_BUFFER, "Character insertion complete");

}

// In input.c - Key reading with escape sequence debugging

int editor_read_key() {

    int nread;

    char c;

    while ((nread = read(STDIN_FILENO, &c, 1)) != 1) {

        if (nread == -1 && errno != EAGAIN) {

            DEBUG_ERROR_LOG(DEBUG_INPUT, "Read error: %s", strerror(errno));

            die("read");

        }

    }

    DEBUG_TRACE_LOG(DEBUG_INPUT, "Read byte: %d (0x%02x) '%c'",

                    c, (unsigned char)c, isprint(c) ? c : '?');




    if (c == '\x1b') {

        // TODO: Handle escape sequences with timeout-based parsing

        // TODO: Read additional bytes with timeout

        // TODO: Parse arrow keys, function keys, etc.

    }

}
```

```
// TODO: Log escape sequence parsing results  
}  
  
return c;  
}
```

## Milestone Debugging Checkpoints

After completing each milestone, use these verification procedures to ensure your implementation is solid:

### Milestone 1 Checkpoint - Raw Mode and Input:

```
# Test raw mode setup/cleanup  
#  
./editor /dev/null  
  
# Press Ctrl+C - should exit cleanly  
  
# Check terminal works normally after exit  
  
# Test signal handling  
  
./editor /dev/null &  
  
kill -TERM $!  
  
# Terminal should be restored properly  
  
# Test escape sequence parsing  
  
./editor /dev/null  
  
# Press arrow keys - should not print letters  
  
# Press Escape key - should register as ESC_KEY
```

### Milestone 2 Checkpoint - Screen Refresh:

```
# Test screen clearing and cursor positioning

DEBUG_LEVEL=3 DEBUG_CATEGORIES=0x04 ./editor /dev/null

# Check logs/editor_debug.log for ANSI sequences

# Verify cursor position matches internal coordinates

# Test window resize handling

./editor /dev/null

# Resize terminal window - screen should adapt

# Status bar should show correct dimensions
```

BASH

#### Milestone 4 Checkpoint - Text Editing:

```
# Test character insertion edge cases

echo "" > test_empty.txt

./editor test_empty.txt

# Type characters - should work in empty file

# Press Enter - should create new line

# Backspace at line start - should join lines

# Debug coordinate consistency

DEBUG_LEVEL=4 DEBUG_CATEGORIES=0x10 ./editor test.txt

# Move cursor around and verify coordinates in log
```

BASH

#### Common Debugging Scenarios

**Symptom: Terminal left in raw mode after crash**

```
# Immediate recovery

reset

# Diagnosis approach

strace -o trace.log ./editor test.txt

# Look for tcsetattr calls in trace.log

# Verify signal handlers are installed before raw mode

# Prevention

gdb ./editor

(gdb) break enable_raw_mode

(gdb) break disable_raw_mode

(gdb) run test.txt

# Step through and verify cleanup occurs
```

BASH

### Symptom: Screen flickering during editing

```
# Capture screen output for analysis

script -c "./editor test.txt" terminal_output.txt

# Examine terminal_output.txt for excessive cursor movements

# Enable frame buffer debugging

DEBUG_LEVEL=3 DEBUG_CATEGORIES=0x04 ./editor test.txt

# Verify single write() call per screen refresh

# Test frame buffer implementation

valgrind --tool=memcheck ./editor test.txt

# Check for buffer overruns in frame buffer code
```

BASH

### Symptom: Cursor jumps to wrong position

```
# Enable coordinate debugging  
  
DEBUG_LEVEL=4 DEBUG_CATEGORIES=0x10 ./editor test.txt  
  
# Move cursor and observe coordinate changes in log  
  
# Test coordinate conversion functions  
  
echo -e "hello\tworld\ttab" > test_tabs.txt  
  
DEBUG_LEVEL=4 ./editor test_tabs.txt  
  
# Navigate through tabs and verify rx vs cx values
```

BASH

The key to successful debugging is systematic instrumentation from the beginning of development, not after problems occur. Build debugging infrastructure first, then implement editor features with comprehensive logging at each step.

## Future Extensions

**Milestone(s):** All milestones (building on the completed foundation to identify next development opportunities)

Building a text editor is like constructing a house - once you have the solid foundation of basic editing, file I/O, search, and syntax highlighting, you can add countless rooms and amenities to make it more comfortable and powerful. The core architecture we've established provides stable ground for ambitious extensions that can transform our simple editor into a full-featured development environment.

Think of future extensions as evolutionary branches from our main trunk. Each extension builds upon the existing component architecture, leveraging the Terminal Manager's robust I/O handling, the Text Buffer's flexible data structures, and the Renderer's efficient screen management. The modular design we've created makes it possible to add sophisticated features without destabilizing the core editing experience.

The extensions we'll explore fall into several categories: workspace management (multiple buffers and split views), user experience enhancements (configuration and themes), developer productivity features (advanced language support and project integration), and system integration capabilities (plugin architecture and external tool integration). Each category represents a different dimension of editor evolution, from basic usability improvements to power-user features that rival professional IDEs.

### Multiple Buffer Management

The most natural extension is supporting multiple open files simultaneously. Think of this as expanding our editor from a single notepad to a complete desk with multiple documents spread across it. Users need to see

what files are open, switch between them efficiently, and manage the memory and screen real estate required for multiple active buffers.

The current `editor_config` structure assumes a single active buffer, but multi-buffer support requires a collection of buffer states with one designated as active. Each buffer needs its own `erow` array, cursor position, scroll offset, filename, dirty flag, and undo stack. The challenge lies in memory management - keeping frequently accessed buffers in memory while potentially swapping inactive buffers to disk for large projects.

### Decision: Buffer Pool Architecture

- **Context:** Supporting multiple files requires managing memory for potentially dozens of open buffers while maintaining responsive editing performance
- **Options Considered:**
  1. Keep all buffers in memory always
  2. Implement buffer pool with LRU eviction to disk
  3. Lazy loading with background buffer management
- **Decision:** Buffer pool with LRU eviction and configurable memory limits
- **Rationale:** Provides predictable memory usage while keeping active buffers responsive, with background loading for smooth user experience
- **Consequences:** Enables large project support but requires complex buffer lifecycle management and serialization logic

The buffer management system needs several key data structures to track multiple files effectively:

| Structure                 | Type                       | Description  |
|---------------------------|----------------------------|--|
| <code>BufferPool</code>   | <code>buffer_pool</code>   | Manages collection of open buffers with LRU eviction policy  |
| <code>BufferEntry</code>  | <code>buffer_entry</code>  | Wrapper around text buffer with metadata and access tracking |
| <code>BufferList</code>   | <code>buffer_list</code>   | Ordered list of buffers for tab bar display and navigation   |
| <code>ActiveBuffer</code> | <code>active_buffer</code> | Reference to currently visible and editable buffer           |
| <code>BufferCache</code>  | <code>buffer_cache</code>  | In-memory cache of frequently accessed buffer content        |

The buffer switching interface requires keyboard shortcuts for rapid navigation between open files. Common patterns include Ctrl+Tab for next buffer, Ctrl+Shift+Tab for previous buffer, and Ctrl+Number for direct buffer selection. The status bar must show the current buffer name and position in the buffer list, while a tab bar or buffer list command shows all open files with their dirty status.

Buffer lifecycle management becomes critical with multiple files. The editor needs policies for when to automatically close buffers, how to handle unsaved changes across multiple buffers, and efficient algorithms

for finding buffers by filename or content. Memory pressure requires evicting inactive buffers while preserving their undo history and cursor positions for seamless restoration.

**⚠ Pitfall: Buffer State Synchronization** Many implementers assume they can simply keep multiple copies of the current single-buffer data structures. This leads to synchronization nightmares where cursor positions, undo stacks, and dirty flags get out of sync between the active buffer display and the stored buffer state. Instead, design a clear separation between buffer storage (permanent state) and buffer view (display state), with explicit synchronization points when switching buffers.

## Split View and Window Management

Split view functionality transforms the editor from a single pane into a multi-pane workspace, like having multiple windows open on your desk simultaneously. This requires sophisticated screen real estate management, coordinating multiple viewports within the terminal's limited screen space, and handling input routing to the currently active pane.

The terminal screen becomes a canvas divided into rectangular regions, each displaying a different buffer or different views of the same buffer. The renderer must calculate pane boundaries, draw separator lines, and manage cursor visibility across multiple panes. Each pane needs its own viewport, scroll position, and display state while sharing the underlying buffer data.

Window management introduces concepts familiar from traditional windowing systems: active pane highlighting, pane resizing, pane creation and destruction, and keyboard shortcuts for navigating between panes. The terminal's limited screen space requires efficient algorithms for pane layout and minimum size constraints to ensure usability.

| Component     | Responsibility                        | Key Operations   |
|---------------|---------------------------------------|--|
| WindowManager | Coordinates multiple panes and layout | Split pane, close pane, resize pane, switch active pane          |
| PaneLayout    | Calculates pane positions and sizes   | Compute boundaries, handle resize events, validate minimum sizes |
| PaneRenderer  | Draws individual pane content         | Render buffer content, draw pane borders, highlight active pane  |
| InputRouter   | Routes input to appropriate pane      | Determine target pane, handle pane switching shortcuts           |

The split view state machine manages transitions between different layout configurations:

| Current Layout | User Action      | New Layout            | Actions Taken  |
|----------------|------------------|-----------------------|--|
| Single pane    | Split horizontal | Two panes horizontal  | Create new pane, adjust heights, update renderer     |
| Two panes      | Split vertical   | Three panes mixed     | Create pane, recalculate layout, redraw all panes    |
| Multiple panes | Close pane       | Fewer panes           | Remove pane, redistribute space, update focus        |
| Multiple panes | Resize pane      | Same count, new sizes | Adjust boundaries, enforce minimums, redraw affected |

Pane navigation requires intuitive keyboard shortcuts that don't conflict with existing editor commands. Common patterns include Ctrl+W for window operations followed by directional keys (h/j/k/l or arrow keys) for movement, plus/minus for resizing, and s/v for horizontal/vertical splitting. The active pane needs clear visual indication through border highlighting or status bar updates.

## Configuration System and User Preferences

A configuration system transforms the editor from a fixed-behavior tool into a customizable environment that adapts to individual user preferences and workflows. Think of it as providing a control panel where users can adjust everything from key bindings and display colors to editing behavior and file associations.

The configuration architecture must balance flexibility with maintainability, providing comprehensive customization without creating a maintenance nightmare. Users need both a simple configuration file format for common settings and programmatic configuration for advanced customizations. The system must handle configuration loading, validation, runtime updates, and graceful fallback for invalid or missing settings.

Configuration data encompasses multiple domains: editor behavior (tab size, line endings, auto-save), display preferences (colors, fonts, line numbers), key bindings (shortcuts, modal mappings), language settings (syntax rules, file associations), and system integration (external commands, shell settings). Each domain requires different validation rules and update mechanisms.

## Decision: Hierarchical Configuration Architecture

- **Context:** Users need both simple text-based configuration and complex programmatic customization without creating maintenance complexity
- **Options Considered:**
  1. Single monolithic configuration file with all settings
  2. Multiple specialized configuration files per feature domain
  3. Hierarchical system with defaults, user config, and runtime overrides
- **Decision:** Hierarchical configuration with domain-specific modules and override precedence
- **Rationale:** Provides clean separation of concerns, supports both simple and advanced use cases, enables partial configuration updates
- **Consequences:** More complex configuration loading logic but much better maintainability and extensibility

The configuration system data structures support hierarchical organization and type safety:

| Structure    | Fields   | Description   |
|--------------|--|---|
| ConfigSystem | <code>domains Config*[], override_stack ConfigOverride*[], current_profile char*</code>        | Root configuration manager with domain handlers and override tracking     |
| ConfigDomain | <code>name char*, schema ConfigSchema*, values ConfigValue*[], validation_fn ValidateFn</code> | Individual configuration domain (editor, display, keys) with typed values |
| ConfigValue  | <code>key char*, type ConfigType, value ConfigData, source ConfigSource, priority int</code>   | Single configuration setting with metadata and precedence tracking        |
| ConfigSchema | <code>entries ConfigSchemaEntry*[], entry_count int, validation_rules ValidationRule*[]</code> | Schema definition for configuration validation and documentation          |

Configuration loading follows a precedence hierarchy: built-in defaults, system-wide configuration, user configuration directory, project-specific settings, and runtime overrides. Each level can override settings from lower precedence levels, with clear rules for merging complex settings like key binding maps or syntax highlighting rules.

The configuration format balances human readability with parsing efficiency. A simple key-value format works for basic settings, while structured formats like JSON or TOML support complex configurations. The parser must provide clear error messages for syntax errors and type mismatches, guiding users toward correct configuration syntax.

Runtime configuration updates enable users to experiment with settings without restarting the editor. Changes to display settings take effect immediately, while changes to core behavior like key bindings require careful state management to avoid inconsistencies. The configuration system must track which settings require restart versus runtime updates.

## Plugin Architecture and Extensibility

A plugin architecture transforms the editor from a fixed application into an extensible platform where users and developers can add functionality without modifying the core codebase. Think of plugins as specialized tools in a workshop - each plugin adds specific capabilities while using the shared infrastructure of the main editor.

The plugin system must balance power with safety, enabling rich extensions while protecting the core editor from plugin bugs or malicious code. This requires defining stable APIs for plugin interaction, sandboxing mechanisms for plugin isolation, and lifecycle management for plugin loading, initialization, and cleanup.

Plugin capabilities span multiple domains: new editing modes (vim emulation, specialized input methods), language support (new syntax highlighting, intelligent code completion), user interface extensions (custom status bars, new color schemes), and external tool integration (version control, build systems, debuggers). Each domain requires different API surfaces and capabilities.

The plugin architecture introduces several key abstractions for safe and powerful extensibility:

| Component      | Responsibility                                 | Plugin Interface   |
|----------------|--|--|
| PluginManager  | Load, initialize, and coordinate plugins       | Registration API, lifecycle hooks, dependency resolution |
| PluginSandbox  | Isolate plugin execution from core editor      | Memory limits, API access control, resource monitoring   |
| PluginRegistry | Track available plugins and their capabilities | Discovery, versioning, conflict detection                |
| PluginAPI      | Provide stable interface to core functionality | Buffer access, rendering hooks, event system             |

The plugin lifecycle manages the complex process of discovering, loading, initializing, and coordinating multiple plugins:

1. Plugin discovery scans configured directories for plugin manifests and validates plugin metadata and compatibility requirements
2. Dependency resolution analyzes plugin requirements and determines loading order to satisfy dependencies
3. Plugin loading reads plugin code into isolated execution contexts with controlled access to editor APIs
4. Initialization phase calls plugin setup routines and registers plugin capabilities with the core editor
5. Runtime coordination routes events and API calls between plugins and core editor components

## 6. Cleanup phase ensures proper resource disposal when plugins are disabled or the editor exits

Plugin API design requires careful consideration of stability and capability. The API must provide access to core editor functionality while remaining stable across editor versions. Key API surfaces include buffer manipulation (read/write text content, cursor control), event system (key presses, buffer changes, file operations), rendering hooks (custom display elements, syntax highlighting), and configuration integration (plugin-specific settings).

 **Pitfall: Plugin API Stability** Exposing too much internal editor state to plugins creates tight coupling that makes editor evolution difficult. Plugin authors start depending on internal implementation details, making it impossible to refactor the core editor without breaking plugins. Instead, design the plugin API around high-level operations and stable abstractions, even if this means some plugin use cases become more complex to implement.

Security considerations become paramount with executable plugins. The plugin system must prevent plugins from accessing unauthorized file system locations, consuming excessive memory or CPU resources, or interfering with other plugins. This requires sandboxing mechanisms appropriate to the implementation language - process isolation for native plugins, virtual machine constraints for interpreted plugins, or capability-based security for any plugin architecture.

## Advanced Language Support

Advanced language support elevates the editor from basic syntax highlighting to intelligent language-aware assistance that understands code structure, provides contextual information, and assists with common programming tasks. Think of this as having a knowledgeable programming partner who understands multiple languages and can provide real-time guidance and assistance.

This extension builds upon the basic syntax highlighting foundation but adds semantic understanding through language servers, code completion, error detection, and code navigation features. The editor becomes a language-aware development environment that can parse code structure, understand symbol relationships, and provide intelligent assistance.

Language Server Protocol (LSP) integration represents the most powerful approach to advanced language support. LSP allows the editor to communicate with language-specific server processes that provide semantic analysis, completion suggestions, error diagnostics, and code navigation. This architecture separates language intelligence from the editor implementation, enabling support for dozens of programming languages through standard protocol interactions.

The LSP integration architecture introduces several new components that coordinate with existing editor systems:

| Component          | Responsibility  | LSP Integration   |
|--------------------|---|---|
| LanguageClient     | Communicate with language servers via JSON-RPC            | Send document changes, receive diagnostics and completion data  |
| CompletionProvider | Generate and display code completion suggestions          | Request completions based on cursor position and context        |
| DiagnosticsManager | Show errors, warnings, and hints from language analysis   | Display inline errors, populate error list, provide quick fixes |
| SymbolNavigator    | Enable go-to-definition and find-references functionality | Request symbol information, navigate to definitions and usages  |

The LSP workflow coordinates between the editor and external language server processes:

1. Server initialization establishes connection with language server and negotiates supported capabilities
2. Document synchronization sends file content and changes to language server to maintain consistent state
3. Completion requests query server for code completion suggestions based on cursor position and local context
4. Diagnostic updates receive error and warning information from server and display inline annotations
5. Symbol queries support navigation features like go-to-definition and find-all-references
6. Shutdown coordination ensures clean disconnection and resource cleanup when editor exits

Code completion transforms the editing experience by providing intelligent suggestions based on language semantics rather than simple word matching. The completion system must integrate with the existing input handling to trigger completion on appropriate characters (dot, colon, space depending on language), display completion options without disrupting the editing flow, and handle completion selection and insertion smoothly.

Error diagnostics provide real-time feedback about code problems without requiring manual compilation or external tool invocation. The diagnostics system overlays error information on the rendered text through colored underlines, margin markers, or inline annotations. The editor must efficiently update diagnostic display as users edit and provide mechanisms for navigating between errors.

## Decision: Asynchronous Language Server Architecture

- **Context:** Language servers can take significant time to analyze code, especially for large projects, but editor responsiveness must remain high
- **Options Considered:**
  1. Synchronous requests that block editor during language server operations
  2. Asynchronous requests with callback-based result handling
  3. Asynchronous requests with polling for completion status
- **Decision:** Fully asynchronous architecture with event-driven result processing
- **Rationale:** Maintains editor responsiveness while providing rich language features, handles slow language servers gracefully
- **Consequences:** More complex state management but much better user experience, especially for large codebases

## Version Control Integration

Version control integration connects the editor with source control systems like Git, providing visual indicators of file status, inline change indicators, and streamlined commit workflows. Think of this as having a version control assistant that keeps you constantly informed about your project's change state and makes common version control operations accessible without leaving the editor.

The integration displays file modification status in the buffer list, shows added/modified/deleted line indicators in the editor margin, and provides commands for common Git operations like staging changes, viewing diffs, and creating commits. This requires monitoring file system changes, executing Git commands, and parsing Git output to extract relevant status information.

Version control status tracking involves several layers of information that must be efficiently synchronized with the editor state:

| Information Layer       | Data Source         | Display Location                   | Update Frequency                  |
|-------------------------|---------------------|------------------------------------|-----------------------------------|
| Repository status       | Git status command  | Buffer list, status bar            | On file save, periodic refresh    |
| File modification state | Git diff per file   | Editor margin, syntax highlighting | On buffer change, file save       |
| Branch information      | Git branch commands | Status bar, window title           | On editor startup, manual refresh |
| Commit history          | Git log commands    | Dedicated history view             | On demand, periodic background    |

The version control workflow integrates with existing editor operations to provide seamless change tracking:

1. Repository detection automatically identifies Git repositories and enables version control features for relevant files
2. Change monitoring tracks modifications to version-controlled files and updates visual indicators accordingly
3. Diff calculation compares current buffer content with Git HEAD to identify added, modified, and deleted lines
4. Status updates synchronize version control information when files are saved, loaded, or externally modified
5. Command integration provides editor shortcuts for common Git operations like add, commit, and push

Visual change indicators enhance the editing experience by showing version control status directly in the editor interface. Added lines receive green highlighting or plus indicators in the margin, modified lines show yellow indicators, and deleted lines display red markers indicating content removal. The indicator system must efficiently update as users edit and avoid performance degradation for large files with many changes.

## **Project Management and Workspace Features**

Project management capabilities transform the editor from a file-focused tool into a project-aware development environment that understands project structure, maintains project-specific settings, and provides project-wide operations like search, navigation, and build integration.

A project workspace encompasses multiple related files with shared configuration, build settings, and development tools. The editor must detect project boundaries (through project files, version control roots, or user configuration), maintain project-specific state, and provide project-wide operations that span multiple files and directories.

Project detection algorithms identify project roots through various heuristics and explicit markers:

1. Version control detection looks for `.git`, `.hg`, or `.svn` directories indicating repository roots
2. Build system detection searches for `Makefile`, `package.json`, `Cargo.toml`, or similar build configuration files
3. IDE markers check for `.vscode`, `.idea`, or similar editor-specific project configuration directories
4. Manual configuration allows users to explicitly define project boundaries and settings

The project data model captures project-wide information and settings:

| Structure        | Fields  | Description  |
|------------------|---|--|
| ProjectWorkspace | root_path char*, config ProjectConfig*, open_files BufferList*, search_index SearchIndex* | Complete project state with configuration and active files |
| ProjectConfig    | build_command char*, test_command char*, file_patterns char*[], exclusions char*[]        | Project-specific settings and commands                     |
| ProjectIndex     | files FileEntry*[], symbols SymbolEntry*[], last_update time_t                            | Cached project structure for fast navigation and search    |
| BuildIntegration | build_system BuildSystemType, commands Command*[], output_parser OutputParser*            | Build system detection and output processing               |

Project-wide search capabilities enable finding text patterns, symbols, or file names across the entire project structure. This requires building and maintaining search indices, handling large result sets efficiently, and providing intuitive interfaces for refining and navigating search results. The search system must respect project exclusion patterns (ignoring build artifacts, dependencies, temporary files) and provide reasonable performance for large projects.

File navigation within projects benefits from fuzzy finding algorithms that help users quickly locate files by typing partial names or paths. The file picker must index project files, support incremental filtering as users type, and provide preview capabilities for selected files before opening them.

## Performance Optimizations and Large File Handling

Performance optimizations become critical as the editor evolves to handle large files, complex projects, and resource-intensive features like language servers and syntax highlighting. Think of performance optimization as tuning a race car - every component must be efficient, and the overall system must remain responsive under demanding conditions.

Large file handling presents unique challenges that the current line-based buffer architecture may not handle efficiently. Files with millions of lines or very long lines can overwhelm memory and cause unacceptable delays in basic operations. The editor needs strategies for lazy loading, viewport-based rendering, and efficient text operations on large datasets.

The performance optimization strategy addresses multiple bottlenecks and resource constraints:

| Performance Domain  | Current Limitation       | Optimization Strategy                | Implementation Approach                           |
|---------------------|--------------------------|--------------------------------------|---|
| Memory usage        | All file content loaded  | Lazy loading with viewport buffering | Load visible sections plus lookahead buffer       |
| Rendering speed     | Full screen redraw       | Incremental updates, dirty regions   | Track changed screen regions, update selectively  |
| Search performance  | Linear text scanning     | Indexed search with caching          | Build search indices for large files and projects |
| Syntax highlighting | Per-character processing | Incremental parsing, state caching   | Parse visible regions, cache highlighting state   |

Large file architecture requires fundamental changes to the text buffer representation. Instead of loading entire files into memory, the editor can implement a paged buffer system that loads file sections on demand. The viewport-based approach keeps several screens of content in memory while reading additional content from disk as needed.

The paged buffer system introduces new data structures for efficient large file management:

| Component       | Responsibility                          | Key Operations  |
|-----------------|---|---|
| PagedBuffer     | Manage file content in fixed-size pages | Load page, evict page, find page containing offset            |
| PageCache       | LRU cache of loaded file pages          | Cache lookup, eviction policy, dirty page management          |
| FileMapping     | Map logical positions to file offsets   | Translate line numbers to file positions, handle line endings |
| ViewportManager | Coordinate visible content loading      | Predict needed pages, preload adjacent content                |

Incremental rendering optimization reduces screen update overhead by tracking which portions of the display have changed and updating only those regions. This requires maintaining dirty region tracking, calculating minimal update sets, and generating efficient ANSI escape sequences for selective screen updates.

**⚠ Pitfall: Premature Optimization** Many developers assume they need complex performance optimizations from the beginning, leading to complicated architectures that don't address actual bottlenecks. Instead, implement the straightforward approach first, measure performance with realistic workloads, and optimize only the components that prove to be bottlenecks. Profiling real usage patterns often reveals surprising performance characteristics that don't match assumptions.

## Implementation Guidance

The future extensions represent significant architectural evolution beyond the core editor foundation. Each extension requires careful planning and incremental implementation to maintain system stability while adding powerful new capabilities.

### Technology Recommendations for Extensions:

| Extension Domain | Simple Approach         | Advanced Approach                |
|------------------|-------------------------|----------------------------------|
| Multiple Buffers | Array of buffer structs | Hash table with LRU eviction     |
| Configuration    | Simple key-value file   | JSON/TOML with schema validation |
| Plugin System    | Shared library loading  | Process isolation with IPC       |
| Language Support | Basic completion lists  | Full LSP client implementation   |
| Version Control  | Shell command execution | Native Git library integration   |
| Large Files      | Memory-mapped files     | Custom paging system             |

### Extension Implementation Priority:

The extensions should be implemented in dependency order to build capabilities incrementally:

1. **Configuration System** - Foundational for all other extensions
2. **Multiple Buffer Management** - Essential for productivity features
3. **Split View** - Natural progression from multiple buffers
4. **Basic Plugin Architecture** - Enables third-party contributions
5. **Version Control Integration** - High-value developer productivity
6. **Advanced Language Support** - Transforms editor into development environment
7. **Project Management** - Coordinates multiple features for project-wide operations
8. **Performance Optimizations** - Address bottlenecks as they arise

### Core Extension Skeleton Code:

For the multiple buffer management foundation:

```
// Multi-buffer support structures
```

C

```
typedef struct {

    erow *rows;

    int numrows;

    int cx, cy, rx;

    int rowoff, coloff;

    char *filename;

    int dirty;

    UndoStack *undo_stack;

    time_t last_access;

} BufferState;
```

```
typedef struct {
```

```
    BufferState *buffers;

    int buffer_count;

    int buffer_capacity;

    int active_buffer;

    int max_memory_mb;

    int current_memory_kb;

} BufferPool;
```

```
// Initialize multi-buffer system
```

```
BufferPool* buffer_pool_create(int max_buffers, int memory_limit_mb) {

    // TODO: Allocate buffer pool structure

    // TODO: Initialize buffer array with initial capacity

    // TODO: Set memory tracking variables

    // TODO: Initialize active buffer index to -1 (no buffers open)

    return NULL; // Replace with actual implementation
```

```
}

// Open file in new buffer or switch to existing buffer

int buffer_pool_open_file(BufferPool *pool, const char *filename) {

    // TODO: Check if file is already open in existing buffer

    // TODO: If found, switch to existing buffer and return buffer index

    // TODO: If not found, create new buffer and load file content

    // TODO: Check memory limits and evict old buffers if necessary

    // TODO: Set new buffer as active and return buffer index

    return -1; // Replace with actual implementation

}

// Switch to buffer by index

int buffer_pool_switch_to(BufferPool *pool, int buffer_index) {

    // TODO: Validate buffer index is within valid range

    // TODO: Save current buffer cursor position and scroll state

    // TODO: Update active_buffer_index to new buffer

    // TODO: Restore cursor position and scroll state for new buffer

    // TODO: Update editor_config to reflect new active buffer

    return 0; // Replace with success/failure code

}
```

For configuration system foundation:

```
// Configuration system structures

typedef enum {

    CONFIG_INT,
    CONFIG_STRING,
    CONFIG_BOOL,
    CONFIG_COLOR

} ConfigType;
```

```
typedef struct {

    char *key;
    ConfigType type;
    union {

        int int_value;
        char *string_value;
        int bool_value;
        int color_value;

    } value;
    char *source_file;
    int line_number;

} ConfigEntry;
```

```
typedef struct {

    ConfigEntry *entries;
    int entry_count;
    int entry_capacity;
    char *config_file_path;
    time_t last_modified;

} ConfigSystem;
```

C

```

// Load configuration from file

int config_load_from_file(ConfigSystem *config, const char *filename) {

    // TODO: Open configuration file and check if readable

    // TODO: Parse each line for key=value pairs

    // TODO: Determine value type and convert appropriately

    // TODO: Store parsed entries in config entries array

    // TODO: Record source file and line numbers for error reporting

    return 0; // Replace with success/failure code

}

// Get configuration value with default fallback

int config_get_int(ConfigSystem *config, const char *key, int default_value) {

    // TODO: Search entries array for matching key

    // TODO: Verify entry type matches expected type (int)

    // TODO: Return entry value if found, default_value if not found

    // TODO: Log warning if key exists but type doesn't match

    return default_value; // Replace with actual lookup

}

```

### Milestone Checkpoints for Extensions:

After implementing multiple buffer support:

- Test: Open 3-4 files and verify each maintains separate cursor position
- Test: Switch between buffers using Ctrl+Tab and verify content is preserved
- Test: Modify different buffers and verify dirty flags are tracked independently
- Test: Close editor and verify unsaved changes prompt appears for all dirty buffers

After implementing basic configuration:

- Create `.kilorc` file with `tab_size=4` and `line_numbers=true`
- Start editor and verify tab size displays as 4 spaces
- Verify line numbers appear in margin if configured

- Test invalid configuration values are ignored with warning messages

#### **Debugging Tips for Extensions:**

| Extension Area   | Symptom                        | Likely Cause                          | Diagnosis                               | Fix                                  |
|------------------|--------------------------------|---------------------------------------|---|--------------------------------------|
| Multiple Buffers | Cursor jumps to wrong position | Buffer state not saved/restored       | Add logging to switch operations        | Save/restore all cursor state fields |
| Configuration    | Settings ignored               | File parsing fails silently           | Check file permissions and parse errors | Add verbose error reporting          |
| Plugin Loading   | Editor crashes on plugin load  | Memory corruption in plugin interface | Use memory debugging tools              | Validate all plugin API parameters   |
| Large Files      | Editor becomes unresponsive    | Loading entire file into memory       | Monitor memory usage during load        | Implement paging or streaming        |

#### **Performance Monitoring Integration:**

```
// Performance monitoring for extensions C

typedef struct {

    clock_t start_time;

    clock_t end_time;

    char operation[64];

    int buffer_id;

    size_t memory_used;

} PerformanceMetric;

// Start performance measurement

void perf_start(PerformanceMetric *metric, const char *operation) {

    // TODO: Record start timestamp

    // TODO: Copy operation name to metric structure

    // TODO: Sample current memory usage if available

}

// End performance measurement and log if over threshold

void perf_end(PerformanceMetric *metric, int threshold_ms) {

    // TODO: Record end timestamp

    // TODO: Calculate elapsed time in milliseconds

    // TODO: If over threshold, log performance warning with details

    // TODO: Add to performance history for trend analysis

}
```

The extensions represent the editor's evolution from a simple text manipulation tool into a comprehensive development environment. Each extension builds upon the solid foundation of the core components while adding new dimensions of functionality and user experience. The key to successful extension implementation is maintaining the architectural principles established in the core system: clean component separation, robust error handling, and efficient resource management.

# Glossary

**Milestone(s):** All milestones (terminology and concepts span the entire development process)

Think of this glossary as a comprehensive field guide for exploring the intricate world of terminal-based text editor development. Just as a naturalist's field guide helps identify and understand different species in their ecosystem, this glossary serves as your reference companion throughout the journey of building a text editor. Each term represents a crucial concept, technique, or component that you'll encounter while navigating the complex intersection of terminal programming, systems-level development, and user interface design.

The terminology in terminal programming has evolved over decades, drawing from the early days of mainframe computing, through the era of dedicated terminal hardware, to modern terminal emulators running on contemporary operating systems. Understanding these terms isn't just about vocabulary—it's about grasping the historical context, technical constraints, and design patterns that shape how terminal applications work today.

## Terminal Programming Fundamentals

**Raw mode** represents a fundamental shift in how the terminal handles input processing. In the analogy of a restaurant, canonical mode is like having a waiter who takes your complete order before bringing it to the kitchen, while raw mode is like having direct access to the kitchen where each ingredient you request is immediately processed. When a terminal operates in raw mode, it disables the normal line buffering, character echoing, and signal processing that users expect in command-line shells. Every keystroke is immediately available to the application without waiting for the user to press Enter.

| Term           | Definition  | Context  |
|----------------|---|--|
| raw mode       | Terminal input mode where characters are available immediately without line buffering or echo | Essential for real-time keystroke processing in interactive applications |
| canonical mode | Default terminal mode with line buffering, echo, and signal processing enabled                | Also called cooked mode; standard for command-line interfaces            |
| cooked mode    | Synonym for canonical mode, representing fully processed terminal input                       | Historical term from early Unix systems                                  |
| line buffering | Input accumulation until newline character before making data available                       | Provides editing capabilities like backspace at shell prompt             |
| character echo | Automatic display of typed characters back to the terminal                                    | Disabled in raw mode for custom display control                          |

The transition between these modes requires careful management of the `termios` structure, which serves as the control panel for all terminal behavior. This POSIX-standard structure contains flags that govern input

processing (`c_iflag`), output processing (`c_oflag`), control modes (`c_cflag`), local modes (`c_lflag`), and special characters (`c_cc`). Understanding termios is crucial because incorrect configuration can leave the terminal in an unusable state.

**ANSI escape sequences** form the command language for controlling terminal display and cursor behavior. These sequences originated with the VT100 terminal standard in the late 1970s and remain the foundation of modern terminal control. Think of escape sequences as a remote control for your terminal—they provide commands for moving the cursor, changing colors, clearing the screen, and manipulating display attributes.

| Sequence                       | Name                  | Purpose                                 | Usage Context                            |
|--------------------------------|-----------------------|---|--|
| <code>ANSI_CLEAR_SCREEN</code> | Screen clear command  | Erases entire terminal display          | Frame preparation before rendering       |
| <code>ANSI_CURSOR_HIDE</code>  | Cursor visibility off | Makes cursor invisible during updates   | Prevents cursor flicker during rendering |
| <code>ANSI_CURSOR_SHOW</code>  | Cursor visibility on  | Makes cursor visible after updates      | Restores cursor after screen refresh     |
| <code>ANSI_COLOR_RESET</code>  | Color reset sequence  | Returns text to default terminal colors | Cleanup after syntax highlighting        |

The parsing of escape sequences presents one of the most challenging aspects of terminal input handling. A single escape character (ASCII 27) can indicate either the user pressing the Escape key or the beginning of a multi-byte sequence representing arrow keys, function keys, or other special inputs. This ambiguity requires sophisticated parsing logic that often relies on timeout-based disambiguation—if additional characters don't arrive within a short window, the single escape is treated as the Escape key.

## Text Editor Architecture Terms

**Immediate mode rendering** and **retained mode rendering** represent two fundamental approaches to screen display management. Immediate mode rendering is like painting a picture from scratch each time—every frame begins with a blank canvas and explicitly draws each element. Retained mode rendering is like having a display list that the system maintains and updates—you modify the list, and the framework handles the actual drawing.

Text editors typically use immediate mode rendering because it provides precise control over every character displayed on screen. Each refresh cycle explicitly constructs the entire screen content, from line numbers through text content to status bar information. This approach ensures consistency and allows for complex display logic like syntax highlighting and search match visualization.

| Architecture Pattern | Description  | Advantages  | Trade-offs                          |
|----------------------|--|---|-------------------------------------|
| Immediate mode       | Explicit construction of each display frame          | Full control, predictable behavior                | Higher CPU usage, more complex code |
| Retained mode        | Framework maintains display state automatically      | Lower cognitive load, automatic optimization      | Less control, potential state bugs  |
| Modal architecture   | Different input interpretation based on current mode | Powerful command sets, context-sensitive behavior | Mode confusion, state complexity    |
| Event-driven         | Response to discrete input events                    | Clear separation of concerns, testable            | Complex event orchestration         |

The **text buffer** serves as the authoritative representation of document content in memory. Unlike simple string storage, sophisticated text buffers use data structures optimized for insertion, deletion, and navigation operations. Common implementations include arrays of strings (one per line), gap buffers that maintain a moveable gap for efficient insertion, or piece tables that track document changes as a sequence of operations on immutable base text.

**Viewport** management coordinates between the logical document space and the physical terminal display area. The viewport acts like a window sliding over the document—it determines which lines and columns are visible and how cursor movements should trigger scrolling. Viewport calculations must account for terminal dimensions, cursor position, and user preferences for scroll margins.

## Input Processing and Command Systems

**Keypress parsing** transforms raw byte sequences from terminal input into logical key events that the editor can process. This involves distinguishing between regular character input, control key combinations, and multi-byte escape sequences. The complexity arises from the variable-length nature of input—a single logical keypress might generate anywhere from one to several bytes of input data.

| Input Type          | Byte Pattern               | Parsing Strategy      | Example          |
|---------------------|----------------------------|-----------------------|------------------|
| Regular character   | Single byte                | Direct character code | 'a' → 97         |
| Control combination | Single byte with high bit  | Mask and interpret    | Ctrl+C → 3       |
| Arrow keys          | Multi-byte escape sequence | State machine parsing | Up arrow → ESC[A |
| Function keys       | Complex escape sequence    | Sequence table lookup | F1 → ESC[OP      |

**Command dispatching** routes parsed input events to appropriate handler functions within the editor. This system must account for the current editor mode—normal editing, search mode, or command mode—and

translate the same physical keypress into different logical operations depending on context. A well-designed command dispatcher uses lookup tables or pattern matching to avoid complex conditional logic.

The **modal architecture** pattern allows the same physical keys to perform different functions depending on the editor's current state. This approach, popularized by vi and emacs, enables powerful command vocabularies without requiring complex key combinations. However, modal interfaces require careful state management and clear visual feedback to prevent user confusion about the current mode.

## Data Structures and Memory Management

**Dynamic arrays** provide the fundamental storage mechanism for text lines and other variable-length data. In C, these require manual memory management with careful attention to capacity growth, reallocation strategies, and cleanup procedures. The typical pattern involves tracking both current size and allocated capacity, growing the allocation by a multiplicative factor when additional space is needed.

| Structure                  | Purpose                      | Key Fields  | Management Strategy                      |
|----------------------------|------------------------------|---|--|
| <code>erow</code>          | Individual text line storage | <code>chars</code> , <code>size</code> , <code>render</code> ,<br><code>rsize</code>  | Dynamic string with render cache         |
| <code>editor_config</code> | Global editor state          | <code>cx</code> , <code>cy</code> , <code>screenrows</code> ,<br><code>numrows</code> | Static allocation with dynamic row array |
| <code>FrameBuffer</code>   | Screen output accumulation   | <code>buffer</code> , <code>capacity</code> , <code>length</code>                     | Exponential growth with reset capability |
| <code>UndoStack</code>     | Operation history tracking   | <code>commands</code> , <code>capacity</code> ,<br><code>current_position</code>      | Ring buffer with position tracking       |

The distinction between **logical coordinates** and **visual coordinates** becomes crucial when handling tab characters and other display-affecting content. Logical coordinates represent position in terms of actual characters in the file, while visual coordinates account for tab expansion, line wrapping, and other display transformations. The functions `cx_to_rx` and `rx_to_cx` handle conversion between these coordinate systems.

**Gap buffers** offer an alternative to simple string arrays for text storage, maintaining a moveable gap at the cursor position to enable efficient insertion and deletion. While more complex than string arrays, gap buffers provide O(1) insertion and deletion at the cursor position, making them suitable for large files with frequent editing.

## File Operations and Persistence

**Atomic write operations** ensure that file saves either complete successfully or leave the original file unchanged. This prevents data loss from interrupted save operations due to system crashes, disk full

conditions, or other failures. The standard approach involves writing to a temporary file, verifying the write succeeded, then atomically renaming the temporary file over the original.

| Operation         | Risk                      | Mitigation Strategy               | Implementation                                       |
|-------------------|---------------------------|-----------------------------------|--|
| Direct file write | Partial writes on failure | Temporary file with atomic rename | Write to <code>.tmp</code> suffix, rename on success |
| Memory exhaustion | Editor crash during save  | Pre-flight memory check           | Validate available memory before operation           |
| Permission errors | Silent save failure       | Explicit error checking           | Test file writability before attempting              |
| Concurrent access | File corruption           | File locking                      | Advisory locks during save operations                |

**Line ending detection** handles the different conventions for marking line boundaries across operating systems. Unix systems use line feed (LF, `\n`), Windows uses carriage return + line feed (CRLF, `\r\n`), and older Mac systems used carriage return (CR, `\r`). Text editors must detect the existing format and preserve it unless the user explicitly requests conversion.

The **dirty flag** tracks whether the current buffer contains unsaved modifications. This flag integrates with the undo system's save point mechanism to provide accurate indication of document state. The flag is set by any editing operation and cleared when the file is saved, with special consideration for undo/redo operations that might return the buffer to a previously saved state.

## Screen Rendering and Display

**Flicker prevention** requires careful coordination of screen updates to avoid partial display states that create visual artifacts. The fundamental principle involves accumulating all screen changes in memory before sending them to the terminal in a single write operation. This eliminates the visible progression of individual updates that causes screen flicker.

| Technique           | Mechanism                            | Benefits                           | Requirements                      |
|---------------------|--------------------------------------|------------------------------------|-----------------------------------|
| Frame buffering     | Accumulate complete screen in memory | Single atomic update               | Memory allocation for full screen |
| Cursor hiding       | Disable cursor during updates        | Eliminates cursor artifacts        | Proper cursor state restoration   |
| Minimal updates     | Send only changed screen regions     | Reduced bandwidth and processing   | Change detection and tracking     |
| Synchronized output | Coordinate with terminal refresh     | Hardware-level flicker elimination | Modern terminal support           |

**Viewport scrolling** maintains the relationship between cursor position and visible screen area. The scrolling logic must handle both vertical movement (line scrolling) and horizontal movement (column scrolling for long lines). Effective scrolling keeps the cursor visible while minimizing jarring jumps that disrupt the user's spatial orientation.

The **render representation** of text lines differs from the storage representation to handle display-specific formatting. While the storage representation preserves the original file content exactly, the render representation expands tabs to spaces, handles Unicode character width, and applies syntax highlighting color codes. This dual representation allows efficient editing operations on the storage form while providing optimized display rendering.

## Undo and Command Systems

**Command pattern** implementation encapsulates edit operations as discrete objects that can be executed, undone, and redone. Each command object contains sufficient information to both perform the operation and reverse it. This pattern enables sophisticated undo functionality while maintaining clean separation between the user interface and text manipulation logic.

| Command Type       | Stored Data                 | Forward Operation            | Reverse Operation                         |
|--------------------|-----------------------------|------------------------------|---|
| CMD_INSERT_CHAR    | Character, position         | Insert character at position | Delete character at position              |
| CMD_DELETE_CHAR    | Character, position         | Delete character at position | Insert stored character at position       |
| CMD_INSERT_NEWLINE | Line number, split position | Split line at position       | Join lines at position                    |
| CMD_COMPOUND       | Sub-command array           | Execute all sub-commands     | Reverse all sub-commands in reverse order |

**Time-based grouping** merges rapid sequences of similar operations into single undo units. Without grouping, users must undo character-by-character when they typically expect to undo word-by-word or sentence-by-sentence. The grouping algorithm considers operation type, timing, and cursor position to determine when to start new undo groups.

**Linear history** represents one approach to undo behavior where new edits discard any previously undone operations. This contrasts with branching undo systems that preserve multiple edit paths. Linear history is simpler to implement and understand, though it can lead to frustration when users accidentally lose undone changes.

## Search and Navigation

**Incremental search** updates match highlighting and cursor position as the user types each character of the search query. This immediate feedback helps users refine their queries and quickly locate desired text. The challenge lies in efficiently updating the match list and display highlighting after each character addition or deletion.

| Search State    | User Actions       | System Response                                   | Display Changes                           |
|-----------------|--------------------|---|---|
| SEARCH_INACTIVE | Trigger search key | Enter search mode, save cursor position           | Show search prompt in status bar          |
| SEARCH_ACTIVE   | Type character     | Update query, find matches, highlight first match | Update match highlighting, move cursor    |
| SEARCH_ACTIVE   | Navigation key     | Move to next/previous match                       | Update current match highlighting         |
| SEARCH_ACTIVE   | Escape key         | Cancel search, restore cursor position            | Clear highlighting, return to normal mode |

**Match highlighting** uses ANSI color codes to visually distinguish search matches from regular text. The system must track match positions throughout the visible text and apply appropriate color sequences during rendering. Special consideration is needed for the current match, which typically receives different highlighting to indicate the cursor's destination.

**Wrap around** behavior determines what happens when search reaches the end of the document. Most users expect search to continue from the beginning, but this can be confusing if not clearly indicated. Effective implementations provide visual feedback when wrapping occurs and allow users to disable this behavior if desired.

## Syntax Highlighting and Language Processing

**Tokenization** breaks text into meaningful units based on programming language syntax rules. The tokenizer must recognize keywords, operators, string literals, comments, and other language elements. This process

typically uses state machine logic to handle multi-line constructs like block comments and string literals that span multiple lines.

| Token Type    | Recognition Pattern                 | Highlighting Color       | State Considerations                   |
|---------------|-------------------------------------|--------------------------|--|
| TOKEN_KEYWORD | Reserved word list lookup           | COLOR_KEYWORD<br>(blue)  | Context-sensitive in some languages    |
| TOKEN_STRING  | Quote-delimited sequences           | COLOR_STRING<br>(green)  | Handle escape sequences and multi-line |
| TOKEN_COMMENT | Language-specific comment markers   | COLOR_COMMENT<br>(gray)  | Block vs line comments                 |
| TOKEN_NUMBER  | Digit sequences with decimal points | COLOR_NUMBER<br>(yellow) | Integer vs floating point vs hex       |

**Language detection** maps file characteristics to appropriate syntax highlighting rules. The most common approach uses file extensions, but sophisticated editors also examine file content for shebang lines, XML declarations, or other identifying markers. The detection system must handle cases where multiple languages might apply or where detection is ambiguous.

**State machines** track context while parsing multi-line language constructs. For example, when the parser encounters the start of a block comment, it must remember this state while processing subsequent lines until it finds the closing marker. This state must persist across line boundaries and screen refresh operations.

## Error Handling and Recovery

**Async-signal-safe** functions can be safely called from signal handlers without risking deadlock or memory corruption. This restriction is crucial for cleanup code that runs when the program receives signals like SIGINT or SIGTERM. Only a limited set of system calls and library functions meet this safety requirement.

| Error Category            | Detection Method                       | Recovery Strategy                       | Prevention Approach                     |
|---------------------------|--|---|---|
| Terminal state corruption | Function return codes, signal handling | Emergency cleanup, terminal reset       | Robust mode management, signal handlers |
| Memory exhaustion         | Allocation failure checks              | Graceful degradation, user notification | Memory limits, early detection          |
| File operation failures   | System call error codes                | User retry prompts, fallback locations  | Permission checks, space verification   |
| Buffer corruption         | Consistency checks, bounds validation  | Data recovery, user notification        | Defensive programming, validation       |

**Terminal state recovery** ensures that the terminal returns to usable condition even when the editor exits abnormally. This requires signal handlers that can perform minimal cleanup operations safely, restoring terminal modes and cursor visibility. The challenge lies in performing this cleanup using only async-signal-safe functions.

**Boundary conditions** occur at the limits of data structures and operations—empty files, cursor at document boundaries, maximum line lengths, and memory limits. Robust editors anticipate these conditions and handle them gracefully rather than crashing or corrupting data.

## Testing and Debugging Terminology

**Mock terminals** simulate terminal behavior for automated testing without requiring actual terminal devices. These test doubles capture output sequences and allow verification of expected ANSI commands, cursor positioning, and screen content. Mock terminals enable comprehensive testing of rendering logic in continuous integration environments.

| Testing Approach       | Scope                               | Benefits                        | Limitations                  |
|------------------------|-------------------------------------|---------------------------------|------------------------------|
| Component unit testing | Individual modules in isolation     | Fast execution, focused testing | Doesn't verify integration   |
| Integration testing    | Full workflows with mock terminals  | End-to-end verification         | Complex test setup           |
| Milestone verification | Feature completion checkpoints      | Progress validation             | Manual verification required |
| Stress testing         | Extreme conditions and large inputs | Reliability validation          | Resource-intensive execution |

**Input simulation** generates programmatic keystroke sequences for automated testing. This allows verification of complex user interactions like multi-step editing operations, search workflows, and undo/redo sequences. Input simulators must accurately reproduce timing-sensitive operations like escape sequence parsing.

**Pseudo-terminals** (ptys) provide programmatic interfaces that appear as real terminals to the application under test. They enable testing of terminal-specific behavior like raw mode operations and escape sequence handling in controlled environments.

## Performance and Optimization

**Incremental rendering** optimizes screen updates by tracking which portions of the display have changed and updating only those regions. This approach reduces both computation and terminal I/O, especially important for large files or slow terminal connections. Implementation requires careful change tracking and region calculation.

| Optimization           | Target                | Technique                           | Trade-off                        |
|------------------------|-----------------------|-------------------------------------|----------------------------------|
| Minimal screen updates | Rendering performance | Change detection and region updates | Complexity in change tracking    |
| Memory pooling         | Allocation overhead   | Reuse freed memory blocks           | Memory usage overhead            |
| Lazy line rendering    | Large file handling   | Render only visible lines           | More complex viewport management |
| Caching compiled regex | Search performance    | Pre-compile common patterns         | Memory usage for cache storage   |

**Buffer pools** manage multiple open files efficiently by sharing memory resources and implementing eviction policies when memory pressure occurs. The least-recently-used (LRU) algorithm typically determines which buffers to remove from memory while preserving user expectations about recently accessed files remaining available.

**Backpressure** provides natural flow control that prevents system overload by allowing processing queues to fill and slow down input acceptance. In text editors, this might manifest as slightly delayed response to rapid typing when complex operations like syntax highlighting can't keep up with input speed.

## Advanced Features and Extensions

**Language Server Protocol (LSP)** standardizes communication between text editors and language intelligence services. This protocol enables features like code completion, error highlighting, and symbol navigation by delegating language-specific logic to dedicated server processes. LSP integration requires sophisticated message handling and state synchronization.

| LSP Feature       | Protocol Message                             | Editor Integration           | User Benefit              |
|-------------------|--|------------------------------|---------------------------|
| Code completion   | <code>textDocument/completion</code>         | Popup suggestion list        | Faster code writing       |
| Error diagnostics | <code>textDocument/publishDiagnostics</code> | Inline error highlighting    | Real-time error detection |
| Go to definition  | <code>textDocument/definition</code>         | Navigation command           | Code exploration          |
| Symbol search     | <code>workspace/symbol</code>                | Search interface integration | Project navigation        |

**Plugin architecture** enables third-party extensions through well-defined APIs and loading mechanisms. Successful plugin systems balance flexibility with security, providing sufficient access to editor internals while preventing plugins from corrupting editor state or compromising system security.

**Split view** and **window management** allow simultaneous display of multiple files or different sections of the same file. This requires sophisticated layout management, input focus tracking, and screen space allocation among active panes.

## Historical Context and Evolution

**VT100** terminal standard, introduced by Digital Equipment Corporation in 1978, established many of the escape sequences and behaviors that modern terminal emulators still support. Understanding this historical foundation helps explain why certain sequences exist and why terminal programming follows particular patterns.

**Termios** represents the POSIX standardization of Unix terminal control, providing portable interfaces for terminal configuration across different Unix-like systems. The structure and flag definitions reflect decades of evolution in terminal hardware and software requirements.

**Key Design Insight:** Terminal programming terminology reflects the evolution from physical hardware devices to software emulation. Many seemingly arbitrary conventions make sense when understood in their historical context of serial communication protocols and hardware limitations.

The progression from hardware terminals through software emulation to modern terminal applications has preserved compatibility while adding new capabilities. This evolution explains why terminal programming often feels like working with legacy systems—because it is, but legacy systems that remain relevant and powerful for many applications.

Understanding these terms and concepts provides the foundation for successful text editor development. Each term represents accumulated knowledge from decades of terminal programming experience, distilled into practical concepts that guide implementation decisions and help avoid common pitfalls.

## Implementation Guidance

The implementation of terminal programming concepts requires careful attention to platform-specific details and system-level programming techniques. Modern C development provides standardized approaches to most terminal operations, though some platform differences persist.

### Technology Selection for Terminal Operations:

| Component         | Standard Approach                           | Platform Considerations                             |
|-------------------|---|---|
| Terminal control  | POSIX termios                               | Windows requires additional compatibility layers    |
| Signal handling   | POSIX sigaction                             | Signal mask management for cleanup reliability      |
| File I/O          | Standard C FILE* with POSIX extensions      | Atomic operations require platform-specific calls   |
| Memory management | Standard malloc/free with custom allocators | Consider arena allocators for undo stack management |

### Essential Headers and Dependencies:

```
#include <termios.h>           // Terminal I/O control          C

#include <unistd.h>            // POSIX operating system API

#include <sys/ioctl.h>          // Device control operations

#include <signal.h>             // Signal handling for cleanup

#include <errno.h>              // Error code definitions

#include <string.h>              // String manipulation

#include <stdlib.h>              // Memory allocation

#include <stdio.h>               // Standard I/O operations

#include <time.h>                // Time operations for undo grouping

#include <ctype.h>                // Character classification
```

### Terminal State Management Skeleton:

```
// Global terminal state for cleanup

static struct termios orig_termios;

static int terminal_initialized = 0;

// Enable raw mode with comprehensive error handling

// Returns 0 on success, -1 on failure

int enable_raw_mode(void) {

    // TODO 1: Get current terminal attributes using tcgetattr

    // TODO 2: Save original attributes to orig_termios global

    // TODO 3: Copy original attributes to raw structure

    // TODO 4: Disable ECHO flag in c_lflag (don't echo characters)

    // TODO 5: Disable ICANON flag in c_lflag (disable canonical mode)

    // TODO 6: Disable ISIG flag in c_lflag (disable interrupt signals)

    // TODO 7: Disable IEXTEN flag in c_lflag (disable extended processing)

    // TODO 8: Disable IXON flag in c_iflag (disable software flow control)

    // TODO 9: Disable OPOST flag in c_oflag (disable output processing)

    // TODO 10: Set VMIN to 0 and VTIME to 1 for non-blocking reads

    // TODO 11: Apply new attributes using tcsetattr with TCSAFLUSH

    // TODO 12: Set terminal_initialized flag and return success

    // Hint: Always check return values from tcgetattr and tcsetattr

}

// Restore terminal to original state

void disable_raw_mode(void) {

    // TODO 1: Check if terminal was properly initialized

    // TODO 2: Restore original attributes using tcsetattr

    // TODO 3: Clear terminal_initialized flag

    // Note: This must be async-signal-safe for signal handlers
```

```
}
```

### ANSI Sequence Generation:

```
// Centralized escape sequence definitions C

#define ESC "\x1b"

#define CLEAR_SCREEN ESC "[2J"

#define CURSOR_HIDE ESC "[?25l"

#define CURSOR_SHOW ESC "[?25h"

#define CURSOR_HOME ESC "[H"

// Generate cursor position sequence

// Returns number of characters written to buffer

int format_cursor_position(char *buffer, int row, int col) {

    // TODO 1: Use snprintf to format ESC[row;colH sequence

    // TODO 2: Add 1 to row and col (terminal coordinates are 1-based)

    // TODO 3: Return number of characters written

    // Hint: Terminal coordinates start at 1,1 not 0,0

}

// Append string to frame buffer with bounds checking

int frame_buffer_append(FrameBuffer *fb, const char *str, size_t len) {

    // TODO 1: Check if current length + len exceeds capacity

    // TODO 2: If needed, double capacity until sufficient space available

    // TODO 3: Use realloc to grow buffer, handle allocation failure

    // TODO 4: Copy new content using memcpy

    // TODO 5: Update length field and return success

}
```

### Input Processing Infrastructure:

```
// Special key code definitions matching enum values C

enum EditorKey {

    BACKSPACE = 127,
    ARROW_LEFT = 1000,
    ARROW_RIGHT,
    ARROW_UP,
    ARROW_DOWN,
    DELETE_KEY,
    HOME_KEY,
    END_KEY,
    PAGE_UP,
    PAGE_DOWN

};

// Read and parse keyboard input including escape sequences

int editor_read_key(void) {

    // TODO 1: Read single character using read() system call

    // TODO 2: Handle read errors and EAGAIN for non-blocking I/O

    // TODO 3: If character is ESC, check for additional characters

    // TODO 4: Implement timeout-based parsing (100ms timeout)

    // TODO 5: Parse common escape sequences: [A, [B, [C, [D for arrows

    // TODO 6: Parse extended sequences: [3~ for delete, [H for home

    // TODO 7: Return appropriate EditorKey constant or character code

    // Hint: Use select() or poll() for timeout implementation

}
```

## Memory Management Patterns:

```
// Safe memory allocation with error handling C

void* safe_malloc(size_t size) {

    void *ptr = malloc(size);

    if (!ptr && size > 0) {

        // Emergency cleanup before exit

        disable_raw_mode();

        fprintf(stderr, "Memory allocation failed: %zu bytes\n", size);

        exit(1);
    }

    return ptr;
}

// Safe string duplication

char* safe_strdup(const char *str) {

    if (!str) return NULL;

    size_t len = strlen(str);

    char *copy = safe_malloc(len + 1);

    memcpy(copy, str, len + 1);

    return copy;
}

// Dynamic array growth pattern

void ensure_capacity(void **array, size_t *capacity, size_t needed, size_t element_size) {

    // TODO 1: Check if current capacity is sufficient

    // TODO 2: Calculate new capacity (double until sufficient)

    // TODO 3: Use realloc to grow array

    // TODO 4: Handle realloc failure gracefully

    // TODO 5: Update capacity parameter
}
```

```
}
```

## Signal Handler Setup:

```
// Emergency cleanup for signal handlers (async-signal-safe) C

void emergency_cleanup(int sig) {

    // TODO 1: Write cursor show sequence to STDOUT_FILENO

    // TODO 2: Write clear screen sequence to STDOUT_FILENO

    // TODO 3: Restore terminal using tcsetattr (if async-signal-safe)

    // TODO 4: Call _exit() instead of exit() (async-signal-safe)

    // Note: Avoid malloc, printf, or other non-async-signal-safe functions

}

// Install signal handlers for graceful cleanup

void setup_signal_handlers(void) {

    // TODO 1: Install handler for SIGINT (Ctrl+C)

    // TODO 2: Install handler for SIGTERM (termination request)

    // TODO 3: Install handler for SIGSEGV (segmentation fault)

    // TODO 4: Use sigaction() instead of signal() for reliability

    // TODO 5: Set SA_RESTART flag for interrupted system calls

}
```

## File Operation Templates:

```
// Safe file reading with comprehensive error handling

FileLoadResult* load_file_safely(const char *filename) {

    FileLoadResult *result = safe_malloc(sizeof(FileLoadResult));

    // TODO 1: Initialize result structure with default values

    // TODO 2: Check file existence and readability

    // TODO 3: Get file size using fstat() for memory planning

    // TODO 4: Allocate buffer for entire file content

    // TODO 5: Read file in chunks, handling partial reads

    // TODO 6: Detect line ending format during read

    // TODO 7: Split content into lines array

    // TODO 8: Set success flag and return result

    // TODO 9: On any error, set error message and return partial result

}

// Atomic file saving with backup

int save_file_atomically(const char *filename, const FileContent *content) {

    // TODO 1: Generate temporary filename (add .tmp suffix)

    // TODO 2: Open temporary file for writing

    // TODO 3: Write all content to temporary file

    // TODO 4: Flush buffers and sync to disk using fsync()

    // TODO 5: Close temporary file

    // TODO 6: Atomically rename temporary file to target filename

    // TODO 7: Return success, or cleanup and return failure

    // Hint: rename() is atomic on POSIX systems

}
```

## Debugging Integration Points:

```

// Debug logging with conditional compilation

#ifndef DEBUG

#define DEBUG_LOG(level, ...) debug_printf(__FILE__, __LINE__, __func__, level,
__VA_ARGS__)

#else

#define DEBUG_LOG(level, ...) ((void)0)

#endif


// Debug state dumping

void debug_dump_editor_state(void) {

    #ifdef DEBUG

        // TODO 1: Dump cursor position and viewport offset

        // TODO 2: Dump buffer statistics (line count, memory usage)

        // TODO 3: Dump terminal dimensions and raw mode status

        // TODO 4: Dump undo stack state and save point

        // TODO 5: Write debug output to separate log file

    #endif

}

}

```

### Milestone Checkpoints:

After implementing terminal management (Milestone 1):

- Run the editor and verify it enters raw mode without displaying typed characters
- Test that Ctrl+C properly restores the terminal before exiting
- Verify arrow keys are recognized and mapped to movement commands
- Confirm terminal restoration works even when editor crashes

After implementing basic rendering (Milestone 2):

- Verify screen clears and redraws without flicker
- Test cursor positioning matches logical position in buffer
- Confirm status bar displays correct file information
- Test terminal resize handling updates screen dimensions

## Common Implementation Pitfalls:

**⚠️ Pitfall: Terminal Not Restored on Crash** Signal handlers must use only async-signal-safe functions. Using `printf()` or `malloc()` in signal handlers can cause deadlock. Instead, use `write()` with pre-formatted strings and avoid dynamic allocation.

**⚠️ Pitfall: Race Conditions in Escape Sequence Parsing** Distinguishing between ESC key presses and escape sequences requires careful timeout handling. Use `select()` or `poll()` with short timeouts (50-100ms) rather than blocking reads that might hang indefinitely.

**⚠️ Pitfall: Memory Leaks in Dynamic Arrays** Track both individual element cleanup and array cleanup. When growing arrays, ensure old memory is freed and pointers are updated atomically to avoid use-after-free bugs.

**⚠️ Pitfall: Off-by-One Errors in Coordinates** Terminal coordinates are 1-based while C arrays are 0-based. Always add 1 when converting from internal coordinates to terminal positioning commands, and subtract 1 when converting back.

This implementation guidance provides the foundation for building a robust terminal-based text editor while avoiding the most common pitfalls that can lead to terminal corruption, memory leaks, or unstable behavior.