Design Notes for Journal CLI

Manjunatha Sarma Majety

November 8, 2025

Abstract

This document serves as a personal log and design record for the `jrnl` CLI tool. It captures the evolution of the project, lessons learned, and insights into the thought process behind each design choice.

Contents

1 Motivation

The goal of this project was to create a simple, fast, and minimalistic command-line journaling tool that felt native to Unix-style workflows. I wanted something I could rely on daily—lightweight, intuitive, and with a design philosophy centered around clarity and control.

2 Design Evolution

2.1 Phase 1: Structure Before Clarity

nitially, the entire program was written in a purely procedural style. It consisted of two main functions—`display()` and `addEntry()`—which handled reading and writing entries respectively. A small helper function took care of timestamp generation and UNIX-to-human time conversion.

While this design gave the program a basic structure, it was fragile. Any edit to the `jrnl` file could easily break `getline()`. More importantly, the procedural nature of the code made adding new functionality difficult, often requiring partial or complete rewrites.

It worked and fulfilled my initial needs, but it felt far too brittle for long-term or even personal use. Still, this phase was a major milestone. It was my first fully coherent, working program—something I could actually use. I also learned a lot: about `CMake`, time libraries (at least superficially), ANSI color codes, and the importance of clean header/implementation separation. These lessons laid the groundwork for smoother development later on.

2.2 Phase 2: OOP Refactor

This was the biggest structural change—converting the program from a procedural to an object-oriented design. Classes were the main hurdle at first, but once I got them working together, the overall design started to feel cleaner and more natural.

During this phase, I also reworked the display functionality to support more flexible arguments and specifiers. One interesting challenge was parsing expressions like `"*3"`—I needed to extract the numeric part while detecting whether the `*` appeared before or after the number.

My first attempt was a naive ASCII-based conversion, simply doing `num - '48'`. That failed for multi-digit numbers. I eventually fixed it using `stoi()`, which taught me more about how string parsing and type conversion actually work in practice.

This phase gave me a much deeper understanding of C++—not just its syntax, but how its pieces interact beneath the surface. It felt like progress in both code and mindset.

2.3 Work in Progress and Future Plans

The next major focus areas are as follows:

- **Error Handling and Edge Cases:** Ensure the program gracefully handles missing directories, permission issues, or corrupt files specified in the configuration.
- **Piping and Compatibility:** Improve input/output behavior so the tool can work seamlessly with other Unix tools like `cat`, `grep`, or `less`.
- **Backup System:** Implement an automatic backup feature with optional timed snapshots to protect against data loss.
- **Search and Filter:** Add the ability to search entries by keywords or dates—potentially merging with the display specifiers for a unified experience.
- **Export Features:** Allow exporting journal entries to different formats (for example, LaTeX) for archival or printing.
- **Parser Refactor:** Introduce subcommand parsing similar to tools like `git` or `ip`, where each subcommand (e.g., `add`, `log`, `search`) has its own handler.

Long-term, I aim to refine configuration handling, improve modularity, and make the project robust enough for daily use. Eventually, I'd like to release it publicly—not as a polished product, but as something honest, useful, and genuinely mine.