Design Notes for Journal CLI

Manjunatha Sarma Majety

December 11, 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 built on a design philosophy centered around clarity and control.

2 Design Evolution

2.1 Phase 1: Structure Before Clarity

Initially, 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 tiny helper function took care of timestamp generation and UNIX-to-human time conversion.

While this design gave the program a basic skeleton, it was fragile. A single malformed line in the journal file could break `getline()`, and adding new features often meant rewriting large portions of the code. It worked, technically, but it felt far too brittle for something I wanted to trust long-term.

Still, this phase mattered. It was my first fully coherent, working program—something I could actually use. I picked up lessons about `CMake`, time libraries (in a very brute-force kind of way), ANSI color codes, and the importance of clean header/implementation separation. Those small lessons ended up laying the groundwork for everything that came next.

2.2 Phase 2: OOP Refactor

This was the first major transformation: rebuilding the tool from a procedural script into an object-oriented system. Classes felt intimidating at first, but once they started connecting in predictable ways, the structure of the program became clearer, more modular, and much easier to extend.

During this refactor, I reworked the display logic to accept a wider range of arguments and specifiers. One unexpectedly tricky problem was parsing expressions like `"*3"` or `"3*"`. My first attempt was a naive ASCII-based trick (`num - '48'`), which broke instantly for multi-digit numbers. I eventually replaced it with `stoi()`, which forced me to actually learn how string parsing and type conversion work in practice.

This phase gave me a deeper sense of how C++ behaves beneath its surface. Not just what the language can express, but how its moving parts fit together. It felt like progress in both code and mindset.

2.3 Post-Refactor Milestones: Atomic Saves and Configuration

After the OOP refactor, the project crossed its biggest stability milestone so far: **atomic file saves** and **robust configuration handling**. These changes fundamentally altered how safe and trustworthy the tool became.

Originally, the save logic was dangerously naive. The journal was cleared as soon as it was read into memory, and then rewritten completely. A crash, a power loss, or even an interrupted write meant the entire journal could evaporate. At the time, I underestimated how catastrophic that was.

Eventually, I realised that correctness is meaningless without *durability*. So I rewrote the entire save pipeline around POSIX guarantees:

- Write all entries to a temporary `.tmp` file.
- Flush it explicitly to disk via `fsync()`.
- Close the file descriptor.
- Atomically replace the original using `rename()`.
- `fsync()` the parent directory for extra safety.

This ensured that the journal could never exist in a partially-written state. Either the new version replaces the old one entirely, or the old version remains. There is no corrupted in-between. This alone elevated the project from "toy" to "tool I can trust."

Around this period, I also implemented full configuration handling. The program now respects XDG base directories when available, falling back to `$HOME` otherwise. If no config exists, one is generated automatically with sane defaults. Directories and journal files are created lazily and safely.

This phase also pushed me into a long-overdue parser refactor. The earlier argument-handling logic had grown into a messy tangle of conditionals, tightly coupled to execution. The new dispatcher-based design—inspired loosely by `git`—cleanly separates *what* the user wants from *how* the program executes it. No new features appeared, but the internal clarity improved dramatically.

I also had to face an uncomfortable truth: it is almost impossible to write fully error-proof code from the beginning. Defensive checks should have been added incrementally, not as a giant cleanup pass. But tackling it all taught me about:

- POSIX file descriptors and low-level I/O
- Command dispatchers and their simplicity
- Atomic filesystem guarantees and their limits
- The difference between logical correctness and durability
- Why safety mechanisms must be designed, not bolted on later

More than any individual feature, this milestone reshaped how I think about reliability. The journal no longer merely *works*—it now fails *responsibly*.

2.4 The Next Big Thing: Date-Based Filters & Expanding the Config File

In all honesty, every time I thought the tool was finally approaching a neat, sensible conclusion—something that would let me breathe for a moment—it managed to grow again. The next target was proper time-based filtering using `--before` and `--after`. On paper, the logic was simple: use `std::lower_bound` and `std::upper_bound` to mark the search boundaries. Reality, of course, had ahem... opinions.

The real challenge was composability. The range-based filters and time-based filters each worked fine alone. But making them coexist without stepping on each other's toes? That took some thinking.

My first attempt was a quick, almost improvised logic. Surprisingly, it worked—for a handful of cases. But it broke on combinations like `--before` with `3*`, or `--after` with `*3`.

Eventually, early one morning, the actual solution arrived through a tiny bit of math. The two filter types didn't have equal priority the way I assumed. Time-based filters had to dominate, with range-based ones slicing *within* those constraints. Once I rewired the logic around that idea, everything fell neatly into place.

Math saves the day again.

Around this time, I also expanded the configuration system. What used to be a simple path variable grew into something bigger: backup paths, default color codes, and more flexible initialization logic. I split the old single-purpose constructor into clearer methods, making the design more modular and future-proof.

This stage felt like opening a door to a room I hadn't noticed before. Suddenly, all the tiny choices I'd postponed started demanding attention. But fixing them wasn't a burden—it felt like the project was maturing, piece by piece, into something I could be genuinely proud of.

2.5 A Quiet, Strange Kind of Ending

When the color configuration finally clicked into place, I felt this unexpected wave of relief wash over me. Not because the project was truly finished—it isn't, and I still have edge cases and error paths scattered around like crumbs—but because it finally felt *complete enough*. The tool had taken on a stable, coherent shape. A shape I could trust. A shape I could step away from without worry.

It was a strange feeling, almost bittersweet. I've watched this little program grow from a clumsy procedural script into something deliberate, reliable, and strangely alive. And in a quiet way, it has watched me grow too—through mistakes, rewrites, midnight bug hunts, and those tiny breakthroughs that feel bigger than they are. Now it feels like we're about to part ways. I'll still use it often, of course; that's why I built it in the first place. But there's a soft, lingering melancholy in knowing that this chapter is closing, and the tool and I are no longer evolving together at the same pace.

2.6 Work in Progress and Future Plans

The next major focus areas are as follows:

- **Backup System:**
  Implement a backup feature with optional timed snapshots to protect against data loss.
- **Export Features:**
  Allow exporting journal entries to different formats (for example, LaTeX) for archival or printing.

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.