
📚Simple Key-Value Store (File-based Database) – Project Specification

Course Context: This project integrates nearly every key concept from *Effective C*: types, memory, pointers, I/O, modularity, debugging/testing, and advanced features like concurrency or indexing if you strive for stretch goals.

---

1. 🧹Objective

Build a command-line key-value store in C that:

- Loads records from a file named `store.db` on startup.
- Supports commands:
- `put <key> <value>` – inserts or updates an entry.
- `get <key>` – retrieves and prints the value.
- `delete <key>` – removes an entry.
- `exit` – saves and quits.
- Maintains an in-memory data structure for speed.
- Persists changes safely to disk (atomic writes + optional locking).

---

2. 🔍Requirements (Mandatory)

A. Persistent Storage and I/O (`stdio.h`, file functions)

- File format: one record per line, `"key=value\n"`. Disallow `\n` or `=` in keys/values (validate input).
- **On start**: open/create `store.db`, read lines, parse into (key, value).
- **On change (put/delete)**: immediately update file or rewrite entire file atomically:
- Write to a temporary file (e.g. `store.db.tmp`), then `rename()`.
- Handle I/O errors gracefully (e.g., cannot write, bad permissions).

B. In-Memory Data Structure

- Choose a container:

- **Hash table** or **linked list** (acceptable for simplicity).

- Use **struct** for entries:

```
typedef struct entry {
    char *key;
    char *value;
```

```
    struct entry *next; // for linked list or chaining
} entry_t;
```

- Use dynamic memory (`malloc`, `free`). Be careful with string allocation (`strdup()` or `malloc(strlen+1)`).
- Avoid memory leaks: ensure every `malloc`/`strdup` has a corresponding `free`.

C. String Parsing and Manipulation (`string.h`, `strtok()` or manual)

- Split input lines on `=` to extract key/value.
- Parse user commands: separate tokens for command, key, value.
- Reject invalid inputs and print helpful error messages.

D. Defensive Programming

- Validate all inputs — no buffer overflows (e.g. `fgets` + size limit).
- Handle missing keys, malformed lines during load, I/O failures.
- Use `assert()` (module-level) where invariant assumptions exist, but don't use asserts for user input.

E. Modularity > Structure into headers/source files:

- `main.c` – CLI loop.
- `store.h` / `.c` – load, save, put/get/delete, free functions.
- `parse.h` / `.c` – input parsing.
- Use proper forward declarations, include guards.

F. Testing and Debugging

- Compile with debugging flags: `-Wall -Wextra`, enable warnings.
- Use `assert()` for internal checks.
- Optionally integrate a unit-testing approach: e.g., test store operations in a separate `test_store.c`.

---

3. 🕑Stretch Goals (Optional, go beyond minimum)

1. **Batch commands / transactions**

2. `begin`, multiple `put`/`delete`, then `commit` or `rollback`, applying only on commit.

3. **Simple B-tree index**

4. Use sorted array or tree structure to maintain keys in sorted order or allow range queries.

5. **Multiple tables/namespaces**

6. Command syntax: `use <table>`, each table is its own `.db` file.

7. **Concurrency / file locking**

8. Use `fcntl()` or `flock()` to lock `store.db` during writes.

9. Support multi-process safety: readers wait or fail gracefully.

---

4. 📊Scoring & Evaluation

Area	Requirement	Points
Command parsing & CLI	Correct separation of command, key, value	10%
File I/O & atomic writes	Temp file + rename, error handling	20%
In-memory structure	Correct struct design, memory-safe implementation	20%
Modularity	Clear headers/sources, clean interfaces	15%
Defensive coding	Input validation, error messages, no leaks	15%
Debugging/testing	Warnings-free compile, test suite or assertions	10%
Stretch features	At least one advanced feature	10%

---

5. 👜C Concepts Mapping

- **Data structures:** `struct`, pointers, dynamic allocation, pointer arithmetic.
- **Memory management:** `malloc`, `free`, avoid leaks/undefined behavior.
- **Strings & parsing:** `string.h`, `strtok()`, buffer handling.
- **File handling:** `fopen`, `fgets`, `fwrite`, error checking.
- **Error handling:** Return value checks, `assert`, descriptive errors.
- **Modular design:** `extern`, header guards, `static`, compile flags.
- **Debugging/testing:** compiler warnings, assertions, optional unit tests.
- **Concurrency/atomicity:** `rename()`, temp files, optional file locking primitives.

---

6. 🔧Setup, Build & Submission

A. Repo Structure:

```
kvstore/
├── src/
│   ├── main.c
│   ├── store.c
│   ├── store.h
│   ├── parse.c
│   └── parse.h
├── tests/
│   └── test_store.c
```

```
├── CMakeLists.txt
├── README.md
└── build/        # created during build
```

B. CMake Integration

Use CMake for managing the build system.

- Root `CMakeLists.txt` should:
- Set project info (`project(kvstore C)`)
- Specify C standard (`set(CMAKE_C_STANDARD 11)`)
- Add executable (`add_executable(kvstore src/main.c src/store.c src/parse.c)`)
- Add optional test targets
- Compile with:

```
mkdir -p build && cd build
cmake ..
make
./kvstore
```

C. README.md:

- Project overview
- Build instructions (CMake + Make)
- Example usage
- Testing notes (if implemented)

D. Submission:

- Submit Git repo URL via LMS by deadline.
- Ensure repo includes: `src/`, `tests/`, `CMakeLists.txt`, `README.md`

🔗Final Notes

- All behavior must strictly conform to spec—no creative shortcuts like using SQLite.
- Focus on clear modular code, robust error handling, and correct memory use.
- Stretch goals are not required, but well-implemented ones can boost your grade.

Let me know if any part needs clarification—happy coding!