C Advanced

A Complete 6-Chapter Programming Course

Topics covered:

Bit manipulation & bit flags · Data structures built from scratch

Concurrency with pthreads · Undefined behavior in full depth

Debugging & tooling (gdb, valgrind, ASan/UBSan) · Capstone: a persistent key-value store

Exercises: 18 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · framed throughout against Rust

Course 3 of 3 · completes the full C track

Table of Contents

CHAPTER 1 OF 6

Bit Manipulation

COURSE 3 · CH 1

Bit Manipulation

The lowest level of control C offers over a single value

Course 3 starts at the smallest possible unit: individual bits. `c1-3` listed the bitwise operators briefly; this chapter finally uses each one deliberately, and shows what real code built on them looks like.

The Bitwise Operators, In Depth

- `&` (AND) — **masking**: clears every bit not set in both operands
- `|` (OR) — **setting**: turns on every bit set in either operand
- `^` (XOR) — **toggling**: flips exactly the bits set in one operand but not the other
- `~` (NOT) — **inverting**: flips every bit
- `<<` / `>>` (shifts) — multiply/divide by a power of two, one bit position at a time

```
unsigned int a = 0b1100;
unsigned int b = 0b1010;

a & b;   // 0b1000
a | b;   // 0b1110
a ^ b;   // 0b0110
a << 2;  // 0b110000 — multiplied by 4
```

Bit Flags

Packing several independent boolean flags into individual bits of one integer — a compact, historically common C idiom.

```c
#define FLAG_A (1 << 0)
#define FLAG_B (1 << 1)
#define FLAG_C (1 << 2)

unsigned int flags = 0;
flags |= FLAG_A;         // set FLAG_A
flags &= ~FLAG_B;        // clear FLAG_B
if (flags & FLAG_A) { /* FLAG_A is set */ }
```

Rust typically reaches for a dedicated `bitflags` crate, or simply a struct of separate `bool` fields, rather than packing everything into one integer by hand. C's version is partly a legacy of byte-economy habits — but it's still extremely common in real-world APIs.

Bit Fields in Structs

The `:N` syntax packs sub-byte-width fields into fewer total bytes — genuinely useful for memory-constrained code or mapping directly onto hardware registers.

```c
struct Status {
    unsigned int is_active : 1;
    unsigned int is_locked : 1;
    unsigned int priority  : 6;
};
```

A real caveat: the exact bit ordering and packing within a bit-field struct is **implementation-defined** — not portable across different compilers or platforms, unlike every other struct layout rule this course has relied on.

A Real Example — POSIX Open Flags

```c
open(path, O_RDONLY | O_CREAT | O_TRUNC, mode);
```

The exact bit-flag idiom from this chapter, used constantly in real systems code — three independent flags OR'd together into one argument, exactly the pattern just built by hand above.

CONCEPT	RUST	C
Multiple boolean flags	bitflags crate, or a struct of bools	OR-together bits in one integer
Sub-byte struct fields	not a language feature — hand-packed if needed	native bit-field syntax (:N), but implementation-defined layout

NAME EVERY FLAG BIT

Defining a named constant for each flag bit, rather than writing raw magic numbers, is what keeps bit-flag code readable — `FLAG_A` communicates intent; `1` alone does not.

SHIFTING TOO FAR IS UNDEFINED BEHAVIOR, NOT "SHIFTS IN ZEROS"

Shifting a value by an amount greater than or equal to its type's bit width — or left-shifting a negative signed value — is undefined behavior, not a harmless zero result. A 32-bit `int` shifted by 32 is a genuinely easy mistake to make, and Course 3's own Undefined Behavior Deep Dive covers exactly why the compiler is allowed to assume this never happens.

Coding Challenges

CHALLENGE 1

Define three named flag constants using left-shift (FLAG_READ, FLAG_WRITE, FLAG_EXEC), combine all three into one unsigned int with OR, then check and print whether each individual flag is set using AND.

View solution

CHALLENGE 2

Starting from a flags variable with FLAG_READ and FLAG_WRITE both set, clear just FLAG_WRITE using AND with a NOT'd mask, and print the flags variable's value before and after to confirm only FLAG_WRITE was removed.

View solution

CHALLENGE 3

Explain why bit-field struct layouts are described as implementation-defined rather than fully portable, and what practical risk this creates if a bit-field struct is used to interpret data written by a different compiler or platform.

View solution

CHAPTER 1 QUICK REFERENCE

- `&` mask, `|` set, `^` toggle, `~` invert, `<<` / `>>` shift
- **Bit flags** — OR to set, AND with a NOT'd mask to clear, AND to check
- Bit-field structs (`:N`) pack sub-byte fields — but layout is implementation-defined, not portable

- POSIX `open()`'s flag argument is the exact bit-flag idiom, used in real systems code
- Shifting by `>=` the type's bit width, or left-shifting a negative value, is undefined behavior
- **Next chapter:** building data structures from scratch — a linked list and a hash table, no standard-library containers

CHAPTER 2 OF 6

Building Data Structures From Scratch

COURSE 3 · CH 2

Building Data Structures From Scratch

What Rust's Vec and HashMap do for you, none of which exists in plain C

C's standard library has no growable list, no hash map, nothing beyond fixed-size arrays. This chapter builds two of the most fundamental structures by hand — deliberately, to see exactly what a language's standard collections are actually doing underneath.

No Standard Containers

Rust's `Vec` and `HashMap` are genuinely part of the language's own standard toolkit — reach for them, they're just there. C offers nothing equivalent at all. Every real C project either hand-rolls its own structures, exactly as this chapter does, or pulls in a third-party library. This is a real, structural gap, not a minor inconvenience.

A Singly Linked List

```c
typedef struct Node {
    int value;
    struct Node *next;
} Node;

Node *push_front(Node *head, int value) {
    Node *n = malloc(sizeof(Node));
    n->value = value;
    n->next = head;
    return n; // the new head
}
```

Every node is its own heap allocation — exactly `c2-2`'s discipline, now applied repeatedly, once per node.

Traversing and Freeing the List

```c
void free_list(Node *head) {
    while (head != NULL) {
        Node *next = head->next; // save next BEFORE freeing head
        free(head);
        head = next;
    }
}
```

A real, easy trap: freeing a node and *then* reading its `->next` pointer is a use-after-free — exactly `c2-3`'s own bug catalog. The next pointer must be saved *before* the current node is freed, every time.

A Simple Hash Table

An array of "buckets," each bucket its own linked list of key-value pairs, plus a hash function mapping a key to a bucket index.

```c
unsigned int hash(const char *key, int bucket_count) {
    unsigned int h = 5381;
    while (*key) {
        h = h * 33 + *key++;
    }
    return h % bucket_count;
}
```

Collisions & Separate Chaining

Two keys hashing to the same bucket index don't overwrite each other — they're both appended to that bucket's own linked list, exactly the structure built earlier in this chapter, reused as the collision-handling mechanism itself. Rust's `HashMap` handles collisions through a much more sophisticated, already-optimized algorithm internally — the programmer never sees or thinks about it at all.

CONCEPT	RUST	C
Growable list	`Vec<T>` — built in	hand-rolled linked list, or a third-party library
Key-value map	`HashMap<K, V>` — built in, collisions handled internally	hand-rolled, separate chaining written by hand
Freeing every element	automatic — Drop runs for the whole structure	manual — every single node must be individually freed

> **BUILDING THESE ONCE GENUINELY DEEPENS UNDERSTANDING**

Implementing a linked list and a hash table by hand, even once, makes it concretely clear what a "real" library structure like Rust's `HashMap` is actually doing underneath its convenient interface — not just an abstraction to trust blindly.

EVERY ALLOCATION NEEDS ITS OWN FREE — MANY NODES MEANS MANY CHANCES TO LEAK

Freeing only the head pointer, without first freeing every node it originally led to, leaks every one of them — the exact same discipline as `c2-2`, but now multiplied across potentially many separate allocations instead of just one.

Coding Challenges

CHALLENGE 1

Build a singly linked list by calling push_front three times with different values, print every value by traversing the list, then free the entire list correctly.

View solution

CHALLENGE 2

Rewrite free_list so that it reads head->next AFTER calling free(head) instead of before, deliberately reintroducing the bug the chapter warns about. Explain exactly what goes wrong and why.

View solution

CHALLENGE 3

Explain why separate chaining (appending to a bucket's own linked list) is a workable way to handle two keys that hash to the same bucket index, and what would go wrong if a hash table simply overwrote whatever was already in a bucket instead.

View solution

CHAPTER 2 QUICK REFERENCE

- C has no built-in growable list or hash map — every project hand-rolls or imports one
- A linked list node is its own heap allocation — one `malloc` / `free` pair per node
- Always save `->next` *before* calling `free` on the current node
- A hash table is buckets plus a hash function; **separate chaining** handles collisions by appending to a bucket's own list
- Rust's `Vec` / `HashMap` provide all of this built in, with collisions and freeing handled automatically

- **Next chapter:** concurrency with pthreads — threads, mutexes, and race conditions

CHAPTER 3 OF 6

Concurrency with pthreads

COURSE 3 · CH 3

Concurrency with pthreads

The same manual-discipline story as memory management — now applied to multiple threads at once

Every "manual control" story this course has told — pointers, memory, now data structures — has a concurrency counterpart. This chapter shows what happens when multiple threads touch the same data with nothing forcing them to coordinate.

Creating a Thread

```
void *worker(void *arg) {
    printf("Hello from a thread\n");
    return NULL;
}

pthread_t t;
pthread_create(&t, NULL, worker, NULL); // a function pointer + a void* argument
pthread_join(t, NULL);                  // wait for it to finish
```

`pthread_create`'s third argument is a function pointer — a real, direct application of `c2-7`'s own material.

A Genuine Race Condition

```
int counter = 0;

void *increment(void *arg) {
    for (int i = 0; i < 100000; i++) {
        counter++; // NOT atomic — read, modify, write, in three separate steps
    }
    return NULL;
}
```

Two threads both running this concurrently will, almost every time, produce a final `counter` value *less than* 200,000 — `counter++` is really read-then-modify-then-write, and two threads can interleave those three

steps, silently losing updates. The exact final value varies from run to run — a genuinely non-deterministic bug.

Mutexes

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *increment(void *arg) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&lock);
        counter++;
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}
```

Locking around the critical section ensures only one thread executes it at a time — the race is genuinely fixed.

What C Does *Not* Check

This is the real payoff. Nothing in C's type system or compiler prevents forgetting to lock `lock` before touching `counter` — any thread can read or write `counter` directly, at any point, with zero compile-time enforcement. The mutex is purely a **convention** the programmer must remember to follow at every single access site, with nothing tying the mutex to the data it's meant to protect.

Rust's `Send` and `Sync` — Fearless Concurrency, Revisited

`rust2-5`'s own "fearless concurrency" is exactly the fix for what C leaves entirely unchecked. `Send` and `Sync` are marker traits the *compiler* checks — a type that isn't `Sync` genuinely cannot be shared across threads without going through a synchronization primitive like `Mutex<T>` at all; the code simply won't compile otherwise. Critically, Rust's `Mutex<T>` **wraps the data itself** — the compiler refuses to compile any access to the inner value without first acquiring the lock. In C, the mutex and the data are two separate, unrelated variables, connected by nothing but the programmer's own memory.

CONCEPT	RUST	C
Mutex-to-data relationship	`Mutex<T>` wraps the data — enforced by the type system	two separate variables — connected only by convention
Accessing data without locking	compile error — the data isn't reachable at all	compiles fine — a genuine, undetected race condition

CONCEPT	RUST	C
Unlocking on an early return	`automatic — MutexGuard's Drop unlocks it`	`manual — every exit path must call unlock explicitly`

NAME MUTEXES AFTER WHAT THEY PROTECT, AND DOCUMENT IT

Since nothing in the language ties a mutex to its data, clear naming and comments are the only thing standing in for the compile-time guarantee Rust provides automatically.

A FORGOTTEN UNLOCK DEADLOCKS THE NEXT LOCK ATTEMPT

An early return or an error path that skips `pthread_mutex_unlock` leaves the mutex permanently locked — the next thread to call `lock` on it blocks forever. Rust's `MutexGuard` unlocks automatically via `Drop` when it goes out of scope, on every exit path, including early returns — the same automatic-cleanup guarantee `c2-2` already established for memory, now applying to locks as well.

Coding Challenges

CHALLENGE 1

Write a program that spawns two threads, each incrementing a shared global counter 100,000 times with no mutex protection. Run it a few times and report whether the final value is consistently 200,000.

View solution

CHALLENGE 2

Fix Challenge 1's program by adding a pthread_mutex_t around the increment. Run it several times and confirm the final value is now consistently and correctly 200,000.

View solution

CHALLENGE 3

Explain precisely why Rust's Mutex<T> makes an unprotected access to shared data a compile error, while C's pthread_mutex_t makes the identical mistake something that compiles and runs, with only sometimes-visible consequences.

View solution

CHAPTER 3 QUICK REFERENCE

- `pthread_create` / `pthread_join` — spawn and wait for a thread; the entry point is a function pointer
- `counter++` is read-modify-write, not atomic — concurrent access without synchronization is a real race condition
- `pthread_mutex_lock` / `unlock` around a critical section fixes the race
- Nothing in C ties a mutex to the data it protects — pure programmer convention, unchecked by the compiler
- Rust's `Mutex<T>` wraps the data itself, and `MutexGuard`'s automatic unlock (via Drop) prevents forgotten-unlock deadlocks entirely
- **Next chapter:** Undefined Behavior Deep Dive — every UB thread this course has flagged, finally covered in full

CHAPTER 4 OF 6

Undefined Behavior Deep Dive

COURSE 3 · CH 4

Undefined Behavior Deep Dive

Every UB thread this course has flagged since Chapter 2, finally given its full, formal explanation

"Undefined behavior" has appeared in nearly every chapter of this course. This one finally defines it precisely, explains the genuinely counter-intuitive reason it's dangerous, and catalogs every instance flagged so far in one place.

What "Undefined Behavior" Actually Means

The C standard defines three genuinely different categories, and conflating them is a real, common mistake:

- **Unspecified behavior** — the standard allows several possible results; the implementation picks one, but doesn't have to document which
- **Implementation-defined behavior** — unspecified, but the implementation *must* document its choice consistently (`c3-1`'s bit-field ordering is exactly this)
- **Undefined behavior** — the standard imposes *literally no requirements at all*. Anything is a conforming result — including things that look nonsensical, like appearing to work perfectly, or genuinely misbehaving in a way that has nothing obviously to do with the actual bug

Why the Compiler Is Allowed to Assume UB Never Happens

This is the real, counter-intuitive payoff. Compilers don't merely fail to detect UB — they actively use its *absence* as a proven fact during optimization. If code contains a signed overflow, the compiler is standard-sanctioned in assuming it never actually occurs, and can eliminate branches or checks that logically depended on that "impossible" case — producing results that look like a compiler bug, but are the compiler correctly exploiting a guarantee the standard itself grants it. The C community's own phrase for the theoretical extreme of this is that UB could licitly make "demons fly out of your nose" — a real, standard-permitted outcome, however absurd it sounds.

A Consolidated Catalog, By Chapter

- **Signed integer overflow** — `c1-2`

- **Out-of-bounds array access** — `c1-6`
- **Dereferencing NULL or a dangling pointer** — `c1-7`
- **Buffer overflow via strcpy** — `c1-8`
- **Use-after-free, double-free** — `c2-2`, `c2-3`
- **Reading the "wrong" union member** — `c2-1`, in most cases
- **Excessive or negative shifts** — `c3-1`
- **An unprotected data race** — `c3-3`

Signed Overflow, Revisited With the Real Mechanism

```
for (int i = 0; i + 1 > i; i++) {
    // intended to stop once i reaches INT_MAX
}
```

The programmer intended this loop to terminate once `i` overflows. But since signed overflow is UB, the compiler is entitled to assume `i + 1 > i` is *always* true — overflow "can't happen" — and may optimize this into a genuine infinite loop, silently removing the termination condition the programmer relied on. This is a real, documented class of surprising optimizer behavior, not a hypothetical.

Strict Aliasing

A new rule: the compiler assumes two pointers of different, unrelated types never point at the same memory (with narrow exceptions like `char *`). Reinterpreting a `float *` as an `int *` and dereferencing it violates this — undefined behavior — and permits the compiler to reorder or cache reads through such pointers in ways that produce genuinely wrong results if the assumption turns out to be false. `c2-1`'s union type-punning gotcha is closely related: a union is one of the narrow, standard-sanctioned ways to reinterpret bits in some cases — a raw pointer cast between unrelated types is not, and violates strict aliasing outright.

Rust's Answer

Rust's entire type system and borrow checker exist substantially to make every category on this chapter's own catalog **structurally impossible to write** in safe code — not detected as a separate check layered on top, but literally inexpressible. `rust3-2`'s own `unsafe` blocks are the deliberate, opt-in escape hatch — the one place a Rust programmer takes on exactly the same responsibility C always has, by choice, in a clearly marked and narrow scope.

UB CATEGORY	C	RUST
Signed overflow	UB — compiler may assume it never happens	panics in debug builds, wraps in release — always defined
Out-of-bounds access	UB — no check at all	panics — checked on every access
Type punning	UB via raw pointer casts; unions are narrowly sanctioned	requires an explicit unsafe block

UBSAN IS A DIFFERENT TOOL FROM ASAN

`-fsanitize=undefined` compiles in instrumentation specifically for UB classes like signed overflow and strict-aliasing violations — genuinely distinct from `c2-3`'s AddressSanitizer, which targets memory-safety bugs specifically. Real projects commonly run both together.

UB ISN'T A RARE EDGE CASE IN OBSCURE CODE

Several of this chapter's own catalog entries — signed overflow, an off-by-one array access, a forgotten free — are genuinely easy to write by accident in completely ordinary-looking code. Treating UB as something only exotic, unusual programs risk is itself a dangerous assumption.

Coding Challenges

CHALLENGE 1

For each entry in this chapter's consolidated UB catalog, name which specific chapter first introduced it and, in one sentence each, what triggers it.

View solution

CHALLENGE 2

Explain why "the compiler is allowed to assume UB never happens" is a genuinely different, stronger claim than "the compiler doesn't check for UB" — and why that distinction is what makes the i + 1 > i infinite-loop example possible.

View solution

CHALLENGE 3

Explain the difference between unspecified, implementation-defined, and undefined behavior, giving one concrete example of each from this course.

📄 View solution

CHAPTER 4 QUICK REFERENCE

- **Unspecified** — one of several allowed results, undocumented; **implementation-defined** — unspecified, but documented; **undefined** — no requirements at all
- The compiler actively assumes UB never happens, and optimizes on that assumption — not merely "fails to catch it"
- A full UB catalog spans nearly every chapter of this course, from signed overflow to data races
- **Strict aliasing** — pointers of unrelated types are assumed never to overlap; violating it is UB
- `-fsanitize=undefined` (UBSan) — a distinct tool from ASan, targeting overflow/aliasing-class UB specifically
- Rust's type system makes this entire catalog structurally inexpressible in safe code — `unsafe` is the deliberate, narrow opt-out
- **Next chapter:** Debugging & Tooling — gdb, valgrind, ASan/UBSan, and static analysis, brought together

CHAPTER 5 OF 6

Debugging & Tooling

COURSE 3 · CH 5

Debugging & Tooling

Every tool this course has mentioned, plus the interactive debugger — brought together into one real workflow

`c2-3` introduced `valgrind` and AddressSanitizer; `c3-4` added UBSan. This chapter adds the one genuinely interactive tool — `gdb` — and shows how a real project uses all of them together.

`gdb` — The GNU Debugger

Compile with `-g` to include debug symbols. Core commands: `break` (set a breakpoint), `run`, `next`/`step` (advance a line, stepping into or over calls), `print` (inspect a variable), `backtrace` (the call stack), `continue`.

A genuinely rare moment of overlap rather than contrast: Rust compiles to native code too, and debugs through the same underlying mechanism (breakpoints, stepping, variable inspection) via `gdb` or `lldb`. Most of this course has been about differences — this is a real, shared skill.

A Worked `gdb` Session

```
// buggy.c — an off-by-one loop
int arr[5] = {1, 2, 3, 4, 5};
int sum = 0;
for (int i = 0; i <= 5; i++) { // <= is the bug — should be <
    sum += arr[i];
}
```

```
$ gcc -g buggy.c -o buggy
$ gdb ./buggy
(gdb) break buggy.c:5
(gdb) run
(gdb) print i
$1 = 0
(gdb) next
(gdb) print sum
$2 = 1
... continue stepping to watch i reach 5, past the array's real bound
```

Post-Mortem Debugging With Core Dumps

A crashed program can leave behind a **core dump** — a snapshot of its memory at the moment of the crash. `gdb ./program core` inspects that exact state after the fact, genuinely useful for bugs that are hard to reproduce interactively on demand.

Bringing the Whole Toolkit Together

TOOL	CATEGORY	RECOMPILE NEEDED?
valgrind	Memory bugs (`c2-3`)	no
AddressSanitizer	Memory bugs (`c2-3`)	yes — fast, precise
UBSan	Undefined behavior (`c3-4`)	yes
gdb	Interactive step-through + post-mortem	no (with -g symbols)

These are complementary, not competing. A real workflow often compiles with ASan and UBSan together for routine testing, then reaches for `gdb` once a bug is narrowed down for deep, interactive investigation.

Static Analysis

A genuinely different category: tools that examine source code **without ever running it** — `clang-tidy`, `cppcheck`. They catch uninitialized variables, some overflow patterns, and suspicious casts before the program executes at all, complementing every runtime tool covered so far. Rust's own compiler performs far more of this kind of analysis *by default*, as a mandatory part of compilation — borrow checking and type checking aren't optional add-ons. In C, static analysis is a separate, optional tool a team has to deliberately choose to run; nothing about the compiler itself enforces it.

> **ROUTINE PRACTICE, NOT A ONE-TIME INVESTIGATION AID**

A mature C project typically runs several of these tools together as everyday development and CI practice — mirroring `pipelines1`'s own CI/CD material — not something reached for only after a bug report arrives.

NO TOOL HERE GUARANTEES CATCHING EVERY BUG

`valgrind`, ASan, and UBSan only catch bugs actually *exercised* by the specific code path run during that particular execution. A bug hiding in an untested branch remains completely invisible to all of them — these tools verify what they observe, not what could theoretically happen.

Coding Challenges

CHALLENGE 1

Compile the chapter's buggy.c with -g, run it under gdb, set a breakpoint inside the loop, and step through enough iterations to observe i reaching 5 — one past the array's valid indices. Report what print arr[i] shows at that point.

View solution

CHALLENGE 2

Explain why valgrind and AddressSanitizer, run against the exact same test input, might catch a bug on one execution but not on a different one — and what this implies about test coverage more broadly.

View solution

CHALLENGE 3

Explain the key difference between static analysis tools (like clang-tidy) and the dynamic tools (valgrind, ASan, UBSan, gdb) covered earlier in the chapter, and why Rust's compiler performing similar analysis by default is a meaningfully different guarantee than C teams choosing to run a static analyzer.

View solution

CHAPTER 5 QUICK REFERENCE

- `gdb` — `break` / `run` / `next` / `step` / `print` / `backtrace`, compiled with `-g`
- Debugging via breakpoints/stepping is a genuinely shared skill with Rust, not a C-only technique
- Core dumps enable post-mortem inspection of a crash's exact state after the fact
- **valgrind/ASan/UBSan/gdb** are complementary — a real project uses several together

- **Static analysis** (clang-tidy, cppcheck) examines source without running it — optional in C, mandatory-by-default in Rust's own compiler
- No tool here catches bugs in code paths that were never actually exercised
- **Next chapter:** the capstone — building a real CLI tool combining everything from this entire track

CHAPTER 6 OF 6

Capstone: Building a Small Project

COURSE 3 · CH 6 — TRACK FINALE

Capstone: Building a Small Project

A persistent key-value store, combining nearly every chapter of this entire 21-chapter track

Twenty chapters, one piece at a time. This closing chapter builds `kvstore` — a real, persisted, command-line key-value store — combining structs, a hand-built hash table, dynamic memory, file I/O, and a genuine multi-file Makefile build into one working program.

The Project — A Persistent Key-Value Store

```
$ kvstore set name Alice
$ kvstore set role admin
$ kvstore get name
Alice
$ kvstore delete role
$ kvstore get role
(not found)
```

Every value survives between runs — loaded from disk on startup, saved back after any change.

The Data Structure

`c3-2`'s hash table, reused directly — buckets of linked key-value pairs, separate chaining for collisions.

```
typedef struct Entry {
    char *key;
    char *value;
    struct Entry *next;
} Entry;

typedef struct {
    Entry *buckets[64];
} HashTable;
```

Reading & Writing Persistence

`c2-6`'s File I/O — a simple text format, one `key=value` per line, loaded on startup and rewritten after every mutation.

```
void save_to_file(HashTable *table, const char *path) {
    FILE *fp = fopen(path, "w");
    if (fp == NULL) return;
    for (int i = 0; i < 64; i++) {
        for (Entry *e = table->buckets[i]; e != NULL; e = e->next) {
            fprintf(fp, "%s=%s\n", e->key, e->value);
        }
    }
    fclose(fp);
}
```

The CLI Interface

`c1-5`'s functions and `c1-8`'s string handling, applied to `argv`.

```
int main(int argc, char *argv[]) {
    if (argc >= 2 && strcmp(argv[1], "set") == 0 && argc == 4) {
        ht_set(table, argv[2], argv[3]);
        save_to_file(table, "kvstore.db");
    }
    // "get" and "delete" follow the same pattern
}
```

Memory Management Throughout

Every key and value is duplicated onto the heap when inserted — `c2-2`'s discipline, applied consistently end to end: freed on `delete`, and freed entirely on program exit, matching `c3-2`'s own `free_list`-style traversal, now applied per bucket.

Structuring the Project With a Makefile

Split into `hashtable.c`/`.h`, `storage.c`/`.h`, and `main.c` — `c2-5`'s real multi-file discipline, with a Makefile tying it all together.

```
kvstore: main.o hashtable.o storage.o
	gcc main.o hashtable.o storage.o -o kvstore

main.o: main.c hashtable.h storage.h
	gcc -c main.c -o main.o

hashtable.o: hashtable.c hashtable.h
	gcc -c hashtable.c -o hashtable.o

storage.o: storage.c storage.h hashtable.h
	gcc -c storage.c -o storage.o
```

Where Each Piece Came From

PIECE	CHAPTER
Hash table + separate chaining	c3-2
File save/load format	c2-6
CLI argument parsing, string comparison	c1-5, c1-8
malloc/free discipline for every key/value	c2-2
Multi-file build, Makefile	c2-5
Verified with valgrind/ASan during development	c2-3, c3-5

What's Still Out of Scope, Honestly

`kvstore` is not thread-safe — no locking (`c3-3`) protects concurrent access to the file or the hash table. Values are plain strings only, not arbitrary binary data. There's no recovery logic for a corrupted save file. And no automated test suite or CI pipeline is wired up in this chapter itself, even though `valgrind`/ASan were used manually during development — a real next step, not something this capstone claims to have solved.

THE THROUGHLINE, RESTATED

Every piece of this capstone is a direct, unmodified application of a chapter's own material — nothing new was introduced here. That's deliberate, matching the same pattern this site's other capstones follow: real capability comes from combining well-understood, individually simple pieces correctly.

THIS IS STILL C — EVERY DISCIPLINE FROM THIS TRACK STILL APPLIES

Nothing about writing a "real" project changes any of the rules covered since Chapter 1. Every allocation still needs its matching free, every array access is still unchecked, every pointer can still dangle. Scale doesn't relax the discipline —

it just multiplies the number of places it has to be applied correctly.

Closing the Course & Track

From `c1-1`'s compile-then-link model to a real, persisted, multi-file program — every chapter in between added one specific piece of manual control C leaves entirely to the programmer, and one specific comparison to what Rust's compiler enforces instead. Twenty-one chapters, one throughline: C trusts you completely. Understanding exactly what that trust costs, and exactly what a compiler-enforced alternative buys back, is the actual lesson this entire track was built to teach.

Coding Challenges

CHALLENGE 1

Identify which specific chapter each of the following pieces of kvstore came from: (a) the separate-chaining collision strategy, (b) the save-file format, (c) the discipline of freeing every key and value on program exit.

View solution

CHALLENGE 2

Explain why kvstore, as described in this chapter, is not safe if two instances of it are run concurrently against the same kvstore.db file — name the specific chapter's material that would need to be applied to fix this.

View solution

CHALLENGE 3

Across the entire 21-chapter C track, name the single idea that recurs most often, and explain in your own words why understanding it deeply matters more than memorizing any individual function or syntax rule.

View solution

CHAPTER 6 QUICK REFERENCE — COURSE & TRACK COMPLETE

- `kvstore` combines: hash table (c3-2), file persistence (c2-6), CLI parsing (c1-5/c1-8), malloc/free discipline (c2-2), and a real Makefile build (c2-5)
- Still out of scope, honestly: thread safety, binary values, corrupted-file recovery, automated testing
- Every discipline from Chapter 1 onward still applies at capstone scale — nothing gets relaxed

- **Course 3 complete** — C Advanced, 6 chapters
- **Full C track complete** — Fundamentals + Intermediate + Advanced, 21 chapters across 3 courses