BUILD IT TO UNDERSTAND IT

Build an LLM Inference Engine in C++

Through Challenges

From a blank project to a working Transformer that loads a real model and generates text — forged one challenge at a time, in modern C++.

Hatem M.

An Engineering Field Guide · First Edition · 2026

Build an LLM Inference Engine in C++ — Through Challenges. Hatem M.

This is an educational text for programmers already fluent in C++ and comfortable with algorithms. It develops a complete CPU-first inference engine for Llama-family Transformer models, from project scaffolding to a full forward pass and a running, model-loading engine, entirely through hands-on challenges.

Reference solutions are provided for learning and verification. Type the code, run the tests, and treat each green test suite as a save point.

Typeset with XeLaTeX using TeX Gyre Pagella, TeX Gyre Heros, and DejaVu Sans Mono. Code is set with `listings`; boxes with `tcolorbox`; diagrams with Ti*k*Z.

First Edition, 2026.

Contents

Preface **iv**

How to Read This Book **vii**

0 Foundation & Setup **1**

0.1 The Project Skeleton 2

0.2 Proof of Life: a Test Framework 7

0.3 Errors and Logging: the Engine's Nervous System 12

0.4 One Command, Green and Clean 17

Unit 0 — Milestone check 20

1 Tensor & Memory **21**

1.1 The Tensor 21

1.2 Aligned Memory and the Arena 32

1.3 Views and Reshape (Zero-Copy) 36

1.4 Zero-Copy Discipline 43

Unit 1 — Milestone check 47

Preface

The pitch

There are two ways to learn how large language models actually run.

The first is to read. You read about attention, about KV caches, about quantization, and you nod along. You can draw the diagrams. You can explain RoPE at a whiteboard. And then someone hands you a two-gigabyte file full of 4-bit-packed weights and asks, *"make this generate a sentence,"* and you discover that the gap between understanding a Transformer and **running** one is enormous — and that almost nobody talks about what lives in that gap.

The second way is the one this book takes. You **build the engine.** Not a notebook that calls a library. Not a 200-line toy that multiplies a couple of matrices. A genuine, modular, CPU-first inference engine in modern C++ that:

- represents tensors with shapes, strides, dtypes, and shared storage;
- implements its own matrix multiplication and pushes it from "naive triple loop" toward something that actually uses your cache and your vector units;
- tokenizes text with byte-level BPE, exactly the way real models expect;
- assembles a full Llama-family Transformer — embeddings, RoPE, causal multi-head attention with GQA, SwiGLU feed-forward, RMSNorm — and runs a forward pass;
- caches keys and values so generation does not re-read the whole context every step;
- quantizes weights to `int8` and `int4` and multiplies the quantized blocks directly;
- memory-maps a GGUF model file, wires the weights up by name, and produces logits that match a reference;
- samples tokens, streams output, and — by the end — serves multiple concurrent requests with continuous batching.

By **Unit 14**, you have the whole thing working end-to-end on a CPU. The advanced machinery that everyone wants to understand — Mixture of Experts,

Flash Attention, speculative decoding, continuous batching and PagedAttention — is not bolted on at the end as a curiosity. Each of those topics lives *inside the unit that earns it*, right after you have built the thing it improves. **Unit 15**, GPU/CUDA, is a clearly-scoped optional capstone: we offload only the heaviest kernels, and we are honest that it is a bonus, not the point.

This is a hard book for a narrow audience, on purpose. If that audience is you, there is very little else like it.

Who this book is for

This book is written for **a programmer who is already fluent in C++ and comfortable with algorithms and data structures.** You will have a much better time if most of the following are true:

- You read and write modern C++ (we use **C++20**) without reaching for a reference every other line: templates, RAII, smart pointers, move semantics, `constexpr`, lambdas. Pointers and memory layout do not scare you.
- You know your way around a build. You have used **CMake** or can pick it up quickly, and you are comfortable on a command line.
- You remember enough linear algebra to know what a matrix–vector product *is* and why the dimensions must line up. We re-derive what we need, but we move fast.
- You have a rough mental model of a neural network — layers, weights, activations — even if you have never implemented one.
- Crucially: **you are willing to be stuck.** Some challenges will not fall on the first try. That discomfort is the mechanism by which this works.

You do not need prior experience with machine-learning frameworks, CUDA, or any LLM internals. That is what we are here to build. If you have never heard of RoPE or GQA, good — you will implement them and they will stop being acronyms. A book that demands strong C++ *and* a tolerance for systems detail *and* a willingness to build a Transformer by hand is not for everyone, and we are not pretending otherwise. That is the point.

A word on mindset

You will be stuck, and that is the product working. The challenges are calibrated so the answer is reachable but not obvious. When you hit the wall, resist jumping straight to the reference solution. Reread the background. Read *one* hint. Write something wrong and run the test to see *how* it is wrong. The understanding you build climbing out of a hole is the one that stays.

Numbers do not lie, and that is the point. Much of LLM systems work is a fight against vagueness — "it seems faster," "the output looks reasonable." This book pushes you toward *measuring*: GFLOP/S, bytes copied, bit-exact matches, tokens per second. The boss challenges convert "I think it works" into "the meter says it works."

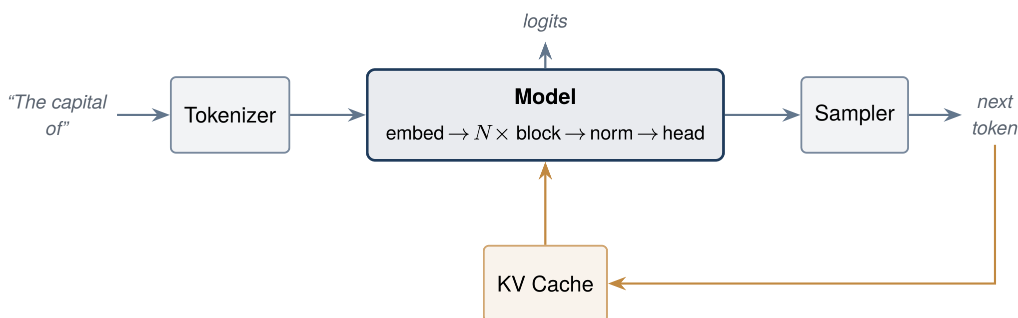
Correct first, then fast. We always build the obvious, slow, obviously correct version first and keep it forever as a reference oracle, then optimize against it. An optimization you cannot check against a reference is a bug you have not found yet.

Respect what you already know. This book moves quickly through what a strong programmer already understands and slows way down on what is genuinely new — memory layout for SIMD, the numerics of softmax, the bit-packing of `int4`, the bookkeeping of a KV cache. It is tuned for someone who finds "here is a for-loop" insulting and "here is why this for-loop's memory access pattern halves your throughput" delightful.

How to Read This Book

What you will build — the engine

It helps to hold the whole machine in your head before forging its parts. For a single generated token: the prompt becomes token IDs, the model turns those into logits, a sampler picks the next token, and a cache lets the next step skip recomputing the past.



The lifecycle of a full generation is the **prefill / decode** split that organizes everything we do:

1. **Encode.** The tokenizer turns your prompt into integer token IDs.
2. **Prefill.** The model runs a forward pass over *all* prompt tokens at once — compute-heavy (a big matrix–matrix multiply per layer) — and, as a side effect, fills the KV cache for every prompt position.
3. **Sample.** From the logits at the last position, the sampler picks the next token (greedy, or temperature / top-k / top-p).
4. **Decode.** Feed that one token back in. The forward pass now processes a *single* position but attends to *all* cached keys and values, so it never recomputes the past. This step is memory-bandwidth-bound — where the KV cache, quantization, Flash Attention, and speculative decoding earn their keep.
5. **Loop.** Append, sample, repeat — until end-of-sequence or a length limit.
6. **Decode (the other kind).** The tokenizer turns the produced IDs back into text.

We build this **from the bottom up**; each unit assumes the one beneath it works.

Unit 14 Performance & optimization — ENGINE COMPLETE
Unit 13 Serving & batching: continuous batching; PagedAttention
Unit 12 Speculative decoding
Unit 11 Sampling & generation: greedy/temp/top-k/top-p; streaming
Unit 10 Model loading: GGUF; mmap; weights-by-name
Unit 9 Quantization: int8 / int4 (Q8_0, Q4_0, K-quants)
Unit 8 Flash Attention
Unit 7 KV cache
Unit 6 Mixture of Experts
Unit 5 Transformer: embeddings; RoPE; attention; GQA/MQA; SwiGLU FFN
Unit 4 Tokenizer: byte-level BPE
Unit 3 Matrix multiplication: naive → blocked → parallel → SIMD
Unit 2 Math kernels: elementwise; SiLU/GELU/SwiGLU; RMSNorm; softmax
Unit 1 Tensor & memory: strides, arena, zero-copy views
Unit 0 Build system · tests · conventions

The “engine is alive” milestone is **Unit 14**, on CPU. Unit 15 (CUDA) offloads only the two heaviest kernels and is explicitly *not* a complete CUDA course.

How the challenges work

This is a book of **challenges**, and the challenge *is* the chapter. You learn by building the next piece, not by reading about it and maybe trying. Each challenge — without exception — has the same seven parts, in order:

1. **Goal** — what the engine gains, as capability, not code.
2. **Background** — the concepts and the *why*: the math, the systems insight, the trade-off, the gotcha that bit everyone before you.
3. **The Task** — a precise specification: interface, behavior, constraints. The contract you fulfill.

4. **Hints** — graduated nudges, from “shape of the approach” to “the specific trick,” so you take exactly as much help as you need.
5. **Tests** — the executable definition of *done*, given up front.
6. **Reference Solution** — a complete, correct, idiomatic implementation with commentary. A *reference*, not *the* answer.
7. **Going Further** — harder variants, real-world considerations, threads to later units.

> **On the reference solutions.** The fastest way to waste this book is to read each reference solution as if it were the assigned reading. The code is here so you are never permanently blocked and so you can compare your instincts against a known-good design — but the learning is in the struggle that precedes it. Write your version first. *Then* read ours. The delta between the two is the most valuable page in each challenge, and it is one only you can see.

Two layers. *Build challenges* carry construction forward in digestible steps; each adds a real, working piece. The *boss challenge* ends the unit — exactly one, hard, and **measured against an objective bar**: not “implement X” but “make X fast enough,” “match the reference byte-for-byte,” “prove this copied zero bytes.” You pass a boss because the measurement says so, not because you wrote code.

The advanced units. Units 6, 8, 12, and 13 (MoE, Flash Attention, speculative decoding, serving) are not quarantined in a final section, because that is pedagogically backwards: Flash Attention only makes sense once you have felt the pain of materializing a full attention matrix — right after you build naive attention. Each advanced unit sits immediately after the foundation it optimizes.

The reference model and test assets

We target the **Llama family** of decoder-only Transformers — the architecture behind Llama 2/3, TinyLlama, Mistral, and Qwen2. They share the ingredients we implement: RoPE, RMSNorm, SwiGLU feed-forward, grouped-query attention (GQA), and byte-level BPE. Once your engine runs one, running its cousins is mostly a matter of reading different metadata.

For most of the book you do *not* need a real model: building bottom-up lets us test each kernel against a mathematical reference using random or hand-constructed weights. Your matrix multiply is correct or it is not, regardless of where the numbers came from. When we reach **Unit 10 (Model Loading)** you will want a small **GGUF** file — TinyLlama-1.1B is an excellent first real model; Llama-3.2-1B/3B or

Qwen2.5-0.5B also work. Until then there is nothing to download.

> **Our north star, named honestly.** This book is, in spirit, *"build your own* `llama.cpp`*, and understand every line."* We borrow that project's hard-won vocabulary — GGUF, `Q4_0`, block quantization, mmap'd weights — because it has become the lingua franca of CPU LLM inference. We are not reproducing its code; we are re-deriving its ideas so they are yours.

Tooling and conventions

Language: modern **C++20** — the genuinely useful parts (`std::span`, `std::bit_cast`, light concepts, designated initializers, `std::source_location`), avoiding what hurts compile times. Any GCC 11+, Clang 14+, or MSVC 2022+ will do. **Build: CMake** (3.20+), target-based and presets-driven. **The bar:** it compiles warning-clean under `-Wall -Wextra -Wpedantic`; hot paths do not allocate; correctness checks compile out of release but scream in debug; tests are fast and dependency-free. **Code in this book** is tagged with the file it belongs in (`// include/llm/tensor.hpp`); engine code lives in `namespace llm`, tests in `namespace llmtest`. **Platforms:** Linux, macOS, Windows; where they diverge — aligned allocation is the first place — we say so and guard it cleanly.

The road ahead

Unit	Title	What you walk away able to do
0	Foundation & Setup	Build, test, and structure the project to engine standards
1	Tensor & Memory	Strided tensors over shared, aligned storage; zero-copy views
2	Math Kernels	Elementwise ops, SiLU/GELU/SwiGLU, RMSNorm, stable softmax
3	Matrix Multiplication	GEMM from naive to cache-blocked, multi-threaded, SIMD
4	Tokenizer	Encode/decode with byte-level BPE, as models expect
5	Transformer	Embeddings, RoPE, causal MHA with GQA, SwiGLU — a full forward pass
6	Mixture of Experts	Route to top-k experts and run sparse FFNS
7	KV Cache	Make generation fast by never recomputing the past
8	Flash Attention	Online softmax and tiling; never materialize the score matrix
9	Quantization	Pack weights to int8 and int4 and multiply them directly
10	Model Loading	mmap a GGUF file, parse metadata, bind weights by name
11	Sampling & Generation	Greedy/temperature/top-k/top-p, the loop, streaming
12	Speculative Decoding	A draft model proposes; a target accepts or rejects
13	Serving & Batching	Concurrent requests, continuous batching, PagedAttention
14	Performance	Profile the path; hand-tune SIMD and threading — **engine complete**
15	*Capstone:* GPU/CUDA	Offload the heaviest kernels to CUDA — a bonus

Enough preamble. Let us build the foundation.

UNIT **0**

Foundation & Setup

What you will build. The project skeleton: a CMake build producing a library, a CLI, and a test binary; a tiny, zero-dependency test framework used to verify every later challenge; and the error-handling and logging conventions the whole engine relies on. By the end, **one command** builds everything and runs a green test suite — warning-clean, reproducible, from a clean checkout.

Where this fits. This is the floor. It does nothing intelligent on its own, but every later unit stands on it. A messy foundation taxes every challenge that follows; a clean one disappears and lets you think about tensors instead of toolchains.

A reasonable instinct, for a strong programmer, is to skim a setup chapter. Skim if you like — but do not skip the two decisions we make here, because they shape the rest of the book: **how the engine handles errors** (an assert that is free in release but loud in debug, plus an always-on check) and **how we test** (a harness with floating-point–aware comparison, because *every* numerical kernel we write needs $\approx$, not $=$). Those two choices recur in literally every unit.

Here is the project layout we build toward; create it now, and the challenges fill it in.

```
llm-engine/
|-- CMakeLists.txt           # the build
|-- CMakePresets.json        # named configs (debug / release / ci)
|-- include/
|   `-- llm/
|       `-- common.hpp       # types, logging, assertions  (Challenge 0.3)
|-- src/
```

```
|   `-- version.cpp          # engine .cpp files arrive from Unit 1 on
|-- tools/
|   `-- cli/
|       `-- main.cpp         # the `llm` command-line tool (a stub for now)
`-- tests/
    |-- test_framework.hpp   # our tiny test harness        (Challenge 0.2)
    |-- test_main.cpp        # the test runner's entry point
    `-- test_smoke.cpp       # a first trivial test
```

We add `include/llm/tensor.hpp`, `src/tensor.cpp`, and friends starting in Unit 1. For now, three build challenges and a boss.

0.1 The Project Skeleton

GOAL

> A reproducible, cross-platform build: one CMake project that compiles a **core library**, a **command-line executable** that links it, and a **test executable**, on Linux, macOS, and Windows, with a single configure-and-build invocation.

BACKGROUND

Before any interesting code, you need a build that *gets out of your way*. For a project that will grow to dozens of source files, ad-hoc compilation (`g++ *.cpp`) collapses almost immediately. We use **CMake** because it is the de-facto standard for C++ and because its target-based style keeps dependencies honest as the project grows.

Two ideas worth internalizing. **Library plus thin executables:** the intelligence of the engine lives in a **library target** (`llm_core`); the CLI and the tests are *thin* executables that link against it. This lets the test binary exercise the exact same code the CLI runs. A common beginner mistake is to put logic in `main.cpp`; resist it. `main.cpp` parses arguments and calls into the library, nothing more.

Modern, target-based CMake: old CMake was a soup of global variables that leaked settings everywhere. Modern CMake attaches properties to *targets* and propagates them along dependency edges with `PUBLIC` / `PRIVATE` / `INTERFACE`. When `llm_core` declares its headers `PUBLIC`, anything that links it gets the include

path automatically. Describe each target's needs; let CMake compute the transitive closure.

We also pin a few things: the standard (**C++20**, required, extensions off), a sane default build type (**Release** — "why is my code 30× slower" is almost always "you built Debug"), and a centralized **warnings** target so every part of the project is held to one standard.

THE TASK

Create the directory structure above and a `CMakeLists.txt` that declares a versioned project `llm_engine`; sets **C++20** (required, extensions off); defaults to **Release** when unset (single-config generators); defines an `INTERFACE` library `llm_warnings` carrying `-Wall -Wextra -Wpedantic` (or `/W4 /permissive-` on MSVC) plus an `option(LLM_WERROR ...)`; defines a static library `llm_core` (with a placeholder source) exposing `include/` as `PUBLIC`; defines executable `llm` linking both; and enables testing with `llm_tests` registered via `add_test`. Write a `main.cpp` stub that prints the engine name and version.

HINTS

Hint 1 — the shape of a target-based CMakeLists

Think in three blocks: *project & global settings*, then *the warnings interface target*, then *one block per real target* (`llm_core`, `llm`, `llm_tests`). Each gets its `target_link_libraries`; the library gets `target_include_directories(... PUBLIC include)`.

Hint 2 — a library with no sources yet

CMake dislikes `add_library` with zero sources. Give it a placeholder `src/version.cpp` defining `const char* llm::version();` less surprising later than an `INTERFACE` library you must convert.

Hint 3 — Debug-only asserts via the build type

Make a compile definition conditional on the configuration: `target_compile_definitions(llm_core PUBLIC $<$<CONFIG:Debug>:LLM_ENABLE_ASSERTS>)`. We use it in Challenge 0.3.

TESTS

This challenge's "test" is the build itself plus a smoke run.

```
# from the project root
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build -j
./build/llm            # should print:  llm-engine 0.1.0
ctest --test-dir build # meaningful after 0.2; configures cleanly now
```

REFERENCE SOLUTION

CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.20)
project(llm_engine LANGUAGES CXX VERSION 0.1.0)

# ---- Global language settings ----
set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)   # for clangd / tooling

# Default to Release for single-config generators if unset.
get_property(_multi GLOBAL PROPERTY GENERATOR_IS_MULTI_CONFIG)
if(NOT _multi AND NOT CMAKE_BUILD_TYPE)
  set(CMAKE_BUILD_TYPE Release CACHE STRING "Build type" FORCE)
  set_property(CACHE CMAKE_BUILD_TYPE PROPERTY STRINGS
               Debug Release RelWithDebInfo MinSizeRel)
endif()

# ---- Centralized warnings ----
add_library(llm_warnings INTERFACE)
option(LLM_WERROR "Treat warnings as errors" OFF)
if(MSVC)
  target_compile_options(llm_warnings INTERFACE /W4 /permissive-)
  if(LLM_WERROR)
    target_compile_options(llm_warnings INTERFACE /WX)
  endif()
else()
  target_compile_options(llm_warnings INTERFACE -Wall -Wextra -Wpedantic)
  if(LLM_WERROR)
    target_compile_options(llm_warnings INTERFACE -Werror)
  endif()
```

```
endif()

# ---- Core library ----
add_library(llm_core STATIC
  src/version.cpp
  # Unit 1 onward appends sources here (src/tensor.cpp, ...).
)
target_include_directories(llm_core PUBLIC include)
target_compile_features(llm_core PUBLIC cxx_std_20)
target_link_libraries(llm_core PRIVATE llm_warnings)
target_compile_definitions(llm_core PUBLIC
  $<$<CONFIG:Debug>:LLM_ENABLE_ASSERTS>)

# ---- Command-line tool ----
add_executable(llm tools/cli/main.cpp)
target_link_libraries(llm PRIVATE llm_core llm_warnings)

# ---- Tests ----
enable_testing()
add_executable(llm_tests
  tests/test_main.cpp
  tests/test_smoke.cpp
)
target_include_directories(llm_tests PRIVATE tests)
target_link_libraries(llm_tests PRIVATE llm_core llm_warnings)
add_test(NAME unit COMMAND llm_tests)
```

`src/version.cpp` gives the library a real source and a version symbol:

```
// src/version.cpp
namespace llm {
const char* version() { return "0.1.0"; }
}
```

`tools/cli/main.cpp`:

```
// tools/cli/main.cpp
#include <cstdio>

namespace llm { const char* version(); }

int main(int argc, char** argv) {
    (void)argc; (void)argv;
```

```
    std::printf("llm-engine %s\n", llm::version());
    // Future subcommands: run, tokenize, perplexity, serve, ...
    return 0;
}
```

A minimal `CMakePresets.json` so nobody memorizes flags:

```
{
  "version": 3,
  "cmakeMinimumRequired": { "major": 3, "minor": 20, "patch": 0 },
  "configurePresets": [
    { "name": "debug",   "binaryDir": "${sourceDir}/build/debug",
      "cacheVariables": { "CMAKE_BUILD_TYPE": "Debug" } },
    { "name": "release", "binaryDir": "${sourceDir}/build/release",
      "cacheVariables": { "CMAKE_BUILD_TYPE": "Release" } },
    { "name": "ci",      "binaryDir": "${sourceDir}/build/ci",
      "cacheVariables": { "CMAKE_BUILD_TYPE": "Release", "LLM_WERROR": "ON" } }
  ],
  "buildPresets": [
    { "name": "debug",   "configurePreset": "debug" },
    { "name": "release", "configurePreset": "release" },
    { "name": "ci",      "configurePreset": "ci" }
  ]
}
```

With this, `cmake --preset release` then `cmake --build --preset release` and you are off. (`llm_tests` will not link until Challenge 0.2 adds `tests/test_main.cpp`; that is next.)

Why these choices. The `INTERFACE` warnings library is the most useful pattern here: every target links it, every target gets identical flags, and `-Werror` for CI is a one-line flip. Defaulting to Release prevents the most common "my engine is slow" question. Exporting `compile_commands.json` makes your editor's language server understand the project.

GOING FURTHER

- **Sanitizers.** A preset compiling Debug with `-fsanitize=address,undefined`. For a memory-heavy engine, ASAN / UBSAN catch out-of-bounds reads in stride math and signed overflow in index computations before they become mysterious wrong numbers.

- **`configure_file` for the version**, so the version lives in one place.
- **A `build.sh`** wrapper for configure + build + test — the healthy path as one command. (We make this the boss.)

0.2 Proof of Life: a Test Framework

GOAL

> A tiny, zero-dependency test framework — `TEST(...)`, `CHECK(...)`, `REQUIRE(...)`, and crucially `CHECK_CLOSE(...)` for floating-point — plus a runner that reports pass/fail counts and exits non-zero on any failure. This is the instrument you will use to verify *every* challenge from here on.

BACKGROUND

You could pull in **doctest** or **Catch2** (both excellent, and in a production engine you probably would). We write our own ~100-line harness instead, for three reasons: **zero dependencies** (the project stays trivially buildable anywhere), **full control** (we add exactly the assertions numerical code needs), and the fact that writing it demystifies what a test framework even *is*. If you prefer doctest, every "Tests" section maps onto it one-to-one.

The one feature we cannot live without — and why a generic `assert` will not do — is **approximate floating-point comparison**. Almost every kernel here produces floating-point numbers, and floating-point arithmetic is not associative: summing a vector left-to-right and summing it in blocks (as a SIMD or multi-threaded version will) gives *slightly* different results. Both are "correct." So our tests must ask "is this close enough?", not "is this bit-identical?". The standard tool uses both an **absolute** and a **relative** tolerance:

$$\text{pass} \iff |a - b| \le \texttt{atol} + \texttt{rtol} \cdot |b|.$$

The absolute term handles values near zero (where relative error is meaningless); the relative term handles large values (where a fixed absolute tolerance would be absurdly strict). Get this right once, here, and every numerical test becomes trustworthy.

We also want two *severities*. `CHECK` records a failure but keeps going, so one test can report all the ways it is broken in a single run. `REQUIRE` records a failure and

aborts that test (continuing is pointless or unsafe), but does not kill the suite; the runner moves on.

THE TASK

Implement `tests/test_framework.hpp` providing: `TEST(name) { ... }` (defines and auto-registers a test — no central list); `CHECK(cond)` (record-and-continue); `REQUIRE(cond)` (record-and-abort-this-test); `CHECK_CLOSE(a, b, tol)` passing iff $|a - b| \leq$ `tol` $+$ `tol` $\cdot |b|$; and `run_all()` executing every test, printing per-test `OK`/`FAIL` and a summary, catching exceptions (reporting rather than crashing), and returning `0` iff all passed. Add `tests/test_main.cpp` and a first `tests/test_smoke.cpp`.

HINTS

Hint 1 — auto-registration without a central list

A test is a function. Register it via *static initialization*: a file-scope object whose constructor runs before `main` and pushes the function into a global registry. `TEST` emits a forward-declared function, a static registrar, and the function's opening brace.

Hint 2 — the registry must be a function-local static

A namespace-scope registry can hit the *static initialization order fiasco*. Wrap it in a function returning a `static std::vector<...>&`; it is then constructed on first use, before any registrar touches it.

Hint 3 — REQUIRE aborts one test, not the suite

Make `REQUIRE` throw a private sentinel after recording the failure. In the runner, `catch` that sentinel silently (already recorded) and other exceptions as failures. A distinct sentinel avoids double-counting.

TESTS

The harness's own smoke test (these should *pass*):

```cpp
// tests/test_smoke.cpp
#include "test_framework.hpp"

TEST(smoke_basic_checks) {
    CHECK(1 + 1 == 2);
    REQUIRE(true);                         // does not abort
    CHECK_CLOSE(0.1 + 0.2, 0.3, 1e-9);    // != in binary fp, but close
}

TEST(smoke_close_handles_zero) {
    CHECK_CLOSE(0.0, 1e-12, 1e-6);        // absolute term saves us near zero
}
```

Then verify the *failure* path manually: temporarily add a `CHECK(false)`, confirm it is reported and the process exits non-zero, and remove it.

REFERENCE SOLUTION

```cpp
// tests/test_framework.hpp
#pragma once
#include <cmath>
#include <cstdio>
#include <exception>
#include <functional>
#include <string>
#include <vector>

namespace llmtest {

// ---- Registry ----
struct TestCase { std::string name; std::function<void()> fn; };

inline std::vector<TestCase>& registry() {
    static std::vector<TestCase> r;   // first-use init: no order fiasco
    return r;
}

struct Registrar {
    Registrar(std::string name, std::function<void()> fn) {
        registry().push_back({std::move(name), std::move(fn)});
    }
};
```

```
// ---- Failure bookkeeping ----
struct Stats { int checks = 0; int check_failures = 0; };
inline Stats& stats() { static Stats s; return s; }

struct RequireFailed {};   // thrown to abort the current test only

inline void report_failure(const char* file, int line, const std::string& what) {
    ++stats().check_failures;
    std::fprintf(stderr, "    FAIL  %s:%d   %s\n", file, line, what.c_str());
}

// ---- Runner ----
inline int run_all() {
    int tests_failed = 0;
    for (const auto& tc : registry()) {
        const int before = stats().check_failures;
        std::printf("[ RUN  ] %s\n", tc.name.c_str());
        try {
            tc.fn();
        } catch (const RequireFailed&) {
            // failure already recorded; just stop this test
        } catch (const std::exception& e) {
            report_failure("<exception>", 0, std::string("uncaught: ") + e.what());
    ↪
        } catch (...) {
            report_failure("<exception>", 0, "uncaught non-std exception");
        }
        if (stats().check_failures > before) {
            std::printf("[ FAIL ] %s\n", tc.name.c_str());
            ++tests_failed;
        } else {
            std::printf("[   OK  ] %s\n", tc.name.c_str());
        }
    }
    std::printf("\n%zu test(s) run, %d failed | %d checks, %d failed\n",
                registry().size(), tests_failed,
                stats().checks, stats().check_failures);
    return tests_failed == 0 ? 0 : 1;
}

}  // namespace llmtest

// ---- Macros ----
#define TEST(test_name)                                                  \
    static void test_name();                                             \
    static ::llmtest::Registrar registrar_##test_name(#test_name, test_name);  \
    static void test_name()
```

```
#define CHECK(cond)                                                         \
    do {                                                                    \
        ++::llmtest::stats().checks;                                        \
        if (!(cond)) ::llmtest::report_failure(__FILE__, __LINE__, #cond);     \
    } while (0)

#define REQUIRE(cond)                                                       \
    do {                                                                    \
        ++::llmtest::stats().checks;                                        \
        if (!(cond)) {                                                      \
            ::llmtest::report_failure(__FILE__, __LINE__, #cond);           \
            throw ::llmtest::RequireFailed{};                               \
        }                                                                   \
    } while (0)

// pass iff |a - b| <= tol + tol*|b|   (absolute + relative tolerance)
#define CHECK_CLOSE(a, b, tol)                                               \
    do {                                                                    \
        ++::llmtest::stats().checks;                                        \
        const double _a = (a), _b = (b), _t = (tol);                       \
        if (!(std::fabs(_a - _b) <= _t + _t * std::fabs(_b)))               \
            ::llmtest::report_failure(__FILE__, __LINE__,                   \
                                      #a " ~= " #b " (tol " #tol ")");      \
    } while (0)
```

```
// tests/test_main.cpp
#include "test_framework.hpp"

int main() {
    return llmtest::run_all();
}
```

Add the smoke test as `tests/test_smoke.cpp`, build, and run `./build/llm_tests`: two `[ OK ]` lines and `2 test(s) run, 0 failed`. `ctest --test-dir build` now reports the `unit` test passing.

Why it is built this way. Auto-registration via static constructors means adding a test is *just writing a* `TEST` — no central list to forget, which matters across the ~100 tests this book accumulates. The function-local-static registry sidesteps initialization-order bugs. The relative-plus-absolute tolerance in `CHECK_CLOSE` is the single most important line for everything numerical that follows; treat it as load-bearing.

GOING FURTHER

- **`CHECK_THROWS(expr, ExceptionType)`** — assert an expression throws (handy for the bounds-checks in Unit 1).
- **Per-test timing** via `std::chrono` — notice when a "unit test" became a benchmark.
- **Filtering:** `run_all(substring)` wired to `argv[1]`, so `./llm_tests tensor` runs the tensor tests.
- **A vector-closeness helper** `check_all_close(got, want, tol)` reporting the worst mismatch. You will reach for this constantly once kernels return arrays.

0.3 Errors and Logging: the Engine's Nervous System

GOAL

> Establish project-wide conventions for **failing loudly** and **observing what is happening**: an assertion that is free in release but aborts with a clear message in debug (`LLM_ASSERT`), an always-on check (`LLM_CHECK`), an `LLM_UNREACHABLE()` marker, and a minimal leveled logger.

BACKGROUND

In a high-performance numerical library, error handling is a *design decision with a performance cost*, not an afterthought. The tension: you want aggressive internal checks (is this index in bounds? do these shapes match? is this pointer aligned?) to catch bugs early — but you cannot afford to run them inside a matrix-multiply inner loop that executes billions of times. The resolution, used by essentially every serious engine, is **two tiers**.

Debug-only assertions (`LLM_ASSERT`) encode *internal invariants* — things true if your code is correct. They run during development (Debug builds), turning "silently wrong number three layers down" into "aborted exactly here with a message." In Release they compile to *nothing*, so the hot loop pays zero cost. Same philosophy as C's `assert`, but we own the message format and the enable flag (tied to the Debug configuration back in 0.1).

Always-on checks (`LLM_CHECK`) encode conditions that depend on *external input* and must hold even in Release — a corrupt model file, an allocation that failed,

a nonsensical user-supplied dimension. You cannot compile these away, because the bad input can arrive in production. They are not in hot loops, so their cost is irrelevant.

Why not exceptions everywhere? You *can* throw, and `LLM_CHECK` could be a throw. But for an engine a failed invariant usually means a programming bug, and the most useful behavior is to abort *immediately* with file/line/function so the stack is intact under a debugger — not to unwind through code that was not written to be exception-safe. We abort by default and note where throwing is the better choice (Unit 10, where a malformed model file is genuinely recoverable).

C++20 gives us `std::source_location`, which captures the caller's file, line, and function *automatically* as a default argument — no `__FILE__`/`__LINE__` threaded through every function. The logger is deliberately minimal: levels, a global threshold, output to `stderr` — enough to narrate what the engine is doing without a logging dependency.

THE TASK

Fill in `include/llm/common.hpp` with: `LLM_ASSERT(cond, msg)` (aborts with a located message when `LLM_ENABLE_ASSERTS` is defined; otherwise a no-op that still *compiles* `cond`); `LLM_CHECK(cond, msg)` (always aborts on false); `LLM_UNREACHABLE()`; a `[[noreturn]] panic(message, source_location)`; and a leveled logger (enum `LogLevel`, a settable global threshold, `log_message(level, msg)`, and convenience macros).

HINTS

Hint 1 — a disabled assert that still type-checks its condition

A no-op that expands to nothing lets the condition rot. Expand the disabled form to `do { (void)sizeof(cond); } while (0)`: `sizeof` does not evaluate its operand, so there is zero runtime cost, but the expression must still be well-formed.

Hint 2 — capturing the caller's location automatically

Give `panic` a parameter `std::source_location loc = std::source_location::current()`. The default argument is evaluated at the *call site*, so `loc` points at the caller.

Hint 3 — why LLM_UNREACHABLE helps beyond documentation

In a `switch` that returns from every case, the compiler may still warn "control reaches end of non-void function." Ending such a function with `LLM_UNREACHABLE();` silences that correctly and, routed to a `[[noreturn]]` function, tells the optimizer the path is dead.

TESTS

```
// tests/test_common.cpp
#include "test_framework.hpp"
#include "llm/common.hpp"

TEST(check_passes_silently_on_true) {
    LLM_CHECK(2 + 2 == 4, "arithmetic still works");
    CHECK(true);   // if we got here, LLM_CHECK did not abort
}

TEST(logger_respects_threshold) {
    llm::set_log_level(llm::LogLevel::Warn);
    LLM_LOG_INFO("this should be suppressed");   // should NOT appear
    LLM_LOG_WARN("this should appear");          // should appear
    CHECK(true);
    llm::set_log_level(llm::LogLevel::Info);      // restore for other tests
}
```

The failure paths (`LLM_CHECK(false, ...)` aborts; `LLM_ASSERT(false, ...)` aborts in Debug) are verified by tripping them once and observing the abort, then removing the line — aborting cannot be wrapped in a normal `CHECK`.

REFERENCE SOLUTION

```
// include/llm/common.hpp
#pragma once
#include <cstdint>
#include <cstdio>
#include <cstdlib>
#include <source_location>
#include <string_view>
```

```cpp
namespace llm {

const char* version();

// ---- Panic & checks ----
[[noreturn]] inline void panic(
    std::string_view msg,
    const std::source_location loc = std::source_location::current()) {
    std::fprintf(stderr, "\n[llm:PANIC] %s:%u in %s\n           %.*s\n",
                 loc.file_name(), loc.line(), loc.function_name(),
                 static_cast<int>(msg.size()), msg.data());
    std::abort();
}

// ---- Logging ----
enum class LogLevel : int { Trace = 0, Debug = 1, Info = 2, Warn = 3, Error = 4 };

inline LogLevel& log_threshold() { static LogLevel level = LogLevel::Info; return
    level; }
inline void set_log_level(LogLevel l) { log_threshold() = l; }

inline const char* level_name(LogLevel l) {
    switch (l) {
        case LogLevel::Trace: return "TRACE";
        case LogLevel::Debug: return "DEBUG";
        case LogLevel::Info:  return "INFO";
        case LogLevel::Warn:  return "WARN";
        case LogLevel::Error: return "ERROR";
    }
    return "?";
}

inline void log_message(LogLevel l, std::string_view msg) {
    if (static_cast<int>(l) < static_cast<int>(log_threshold())) return;
    std::fprintf(stderr, "[llm:%s] %.*s\n", level_name(l),
                 static_cast<int>(msg.size()), msg.data());
}

}  // namespace llm

// ---- Macros (outside any namespace) ----

// Always-on: enforces conditions that depend on external input.
#define LLM_CHECK(cond, msg)                                                \
    do { if (!(cond)) ::llm::panic(msg); } while (0)

// Debug-only invariant checks; zero cost in Release.
```

```
#if defined(LLM_ENABLE_ASSERTS)
  #define LLM_ASSERT(cond, msg)                                              \
      do { if (!(cond)) ::llm::panic(msg); } while (0)
#else
  #define LLM_ASSERT(cond, msg)                                              \
      do { (void)sizeof(cond); /* type-checks without evaluating */ } while (0)
#endif

#define LLM_UNREACHABLE() ::llm::panic("unreachable code reached")

#define LLM_LOG_TRACE(msg) ::llm::log_message(::llm::LogLevel::Trace, msg)
#define LLM_LOG_DEBUG(msg) ::llm::log_message(::llm::LogLevel::Debug, msg)
#define LLM_LOG_INFO(msg)  ::llm::log_message(::llm::LogLevel::Info,  msg)
#define LLM_LOG_WARN(msg)  ::llm::log_message(::llm::LogLevel::Warn,  msg)
#define LLM_LOG_ERROR(msg) ::llm::log_message(::llm::LogLevel::Error, msg)
```

Register `tests/test_common.cpp` in the `llm_tests` sources, build, and run: both tests pass, and (threshold at `Warn`) only the `WARN` line prints during the logger test.

Design notes. Everything is `inline` in the header, so there is no separate `.cpp` to keep in sync; if `panic`/`log_message` grow, move them to `src/common.cpp`. Tying `LLM_ENABLE_ASSERTS` to the build *type* means the right behavior happens automatically: develop and test in Debug with every invariant live, ship Release with the hot paths unburdened. The `(void)sizeof(cond)` trick separates a robust disabled-assert from one that silently lets conditions become uncompilable.

GOING FURTHER

- **Formatted messages** via `std::format`, wrapped so formatting happens only on failure.
- **Throwing variants** (`LLM_CHECK_OR_THROW`) for the genuinely recoverable cases (model loading).
- **A scoped trace timer** — an RAII `ScopedTimer{"prefill"}` logging elapsed time on scope exit. A free, always-available profiler once sprinkled through the forward pass.
- **Compile-time log floor** (`LLM_LOG_MIN_LEVEL`) so verbose logging costs nothing in release.

0.4 One Command, Green and Clean

BOSS CHALLENGE

From a clean checkout, one command builds everything and runs the whole suite — green, with zero warnings under `-Werror`. The bar is objective: no warnings, no failures, reproducible from scratch.

GOAL

> From a **clean checkout**, a **single command** configures, builds, and runs the entire test suite — and the result is **green with zero warnings**. This is the standard every later unit's code is held to, so we make achieving it a one-liner.

BACKGROUND

The difference between a project pleasant to work in for months and one that rots is, more than anything, whether "build it and check it is healthy" is *frictionless*. If it takes six remembered flags, people stop doing it, warnings accumulate, and one day a warning that was actually a bug hides in the noise. The fix: make the healthy path the *easy* path — one command, warnings as errors, tests required to pass. That is also exactly what a continuous-integration server runs, so "passes locally" and "passes CI" become the same statement.

The **measurement** is unusually crisp for a setup unit, and that is intentional. **Zero warnings** under `-Wall -Wextra -Wpedantic` with `-Werror`: a warning is the compiler telling you about a likely bug, and in an engine doing pointer and integer arithmetic all day you cannot afford to ignore them. **Zero test failures**: `ctest` returns success. **Reproducible from clean**: delete `build/`, run the one command, and it all happens again.

THE TASK

Provide a single entry point — `scripts/build.sh` (and/or `build.ps1`), or a documented preset invocation — that configures with the `ci` preset (Release, `LLM_WERROR=ON`), builds all targets, and runs `ctest`. Then make it pass: everything from 0.1–0.3 must compile **warning-clean** with `-Werror` and all tests must pass. Verify reproducibility with `rm -rf build && ./scripts/build.sh`.

The bar (the boss): clean checkout → one command → build succeeds with *zero* warnings (−`Werror` on) → all tests pass → exit code `0`.

HINTS

Hint 1 — fail fast in the script

Start a bash script with `set -euo pipefail` so any failing step aborts the whole script with a non-zero status. A script that plows on after a failed compile and reports "done" is worse than none.

Hint 2 — turning on -Werror via the preset

You already wired `option(LLM_WERROR ...)` and a `ci` configure preset that sets it `ON`. The script just needs `cmake --preset ci`, `cmake --build --preset ci`, then `ctest` pointed at that binary dir.

Hint 3 — the warnings you are most likely to hit first

With `-Wextra`, unused parameters (`argc`, `argv`) and sign-conversion in size/index arithmetic are the usual first offenders. Cast-to-`void` the genuinely-unused, and be deliberate about signed vs unsigned in size math — a habit that prevents real bugs in the stride arithmetic coming next unit.

TESTS

The boss *is* the test. Run it and read the exit code:

```
rm -rf build
./scripts/build.sh
echo "exit: $?"     # must be 0; no 'warning:' lines; ends with all tests passed
```

REFERENCE SOLUTION

`scripts/build.sh`:

```
#!/usr/bin/env bash
# One command: configure (CI preset), build everything, run all tests.
```

```
set -euo pipefail

cd "$(dirname "$0")/.."        # project root, wherever we're called from

cmake --preset ci             # Release + LLM_WERROR=ON
cmake --build --preset ci -j
ctest --test-dir build/ci --output-on-failure

echo "build + tests green"
```

Make it executable (`chmod +x scripts/build.sh`). A PowerShell twin would run the same three `cmake`/`ctest` commands with `$ErrorActionPreference = "Stop"`.

Now make it pass. The most likely fix, given the stubs from 0.1, is the unused-parameter warning in the CLI `main` — handled by the `(void)argc; (void)argv;` already in the reference. A clean run looks like:

```
$ rm -rf build && ./scripts/build.sh
... cmake configure output ...
... compiling llm_core, llm, llm_tests ...
[ RUN  ] smoke_basic_checks
[   OK  ] smoke_basic_checks
[ RUN  ] smoke_close_handles_zero
[   OK  ] smoke_close_handles_zero
[ RUN  ] check_passes_silently_on_true
[   OK  ] check_passes_silently_on_true
[ RUN  ] logger_respects_threshold
[llm:WARN] this should appear
[   OK  ] logger_respects_threshold
...
N test(s) run, 0 failed | M checks, 0 failed
build + tests green
```

Exit code `0`, no `warning:` anywhere, reproducible after `rm -rf build`. **Boss cleared.**

Why this is the right bar to set first. Every challenge after this ends with "the tests pass." By making "build everything warning-clean and run all tests" a single reproducible command *now*, the verification step in every future challenge is trivial to run — which means you will actually run it, which means bugs get caught the moment they are introduced rather than three units later as a wrong logit and a long debugging night.

GOING FURTHER

- **A real CI file** (e.g. GitHub Actions) running `./scripts/build.sh` on Linux, macOS, and Windows — "is it healthy?" answered automatically on every push.
- **Add the sanitizer build to the gate.** Also run a Debug+ASAN/ UBSAN build of the tests; from Unit 1 on, this catches memory bugs in your tensor code the instant they appear.
- **Build timing budget.** Print total wall-clock time. A foundation that takes 90 seconds to build clean is one you will stop rebuilding; watch it.

Unit 0 — Milestone check

MILESTONE CHECK

You now have:

- a **target-based CMake build** producing a core library, a CLI, and a test binary, defaulting to Release, with centralized warnings and a `ci` preset that treats warnings as errors;
- a **zero-dependency test framework** with `TEST`/`CHECK`/ `REQUIRE` and — the load-bearing piece — `CHECK_CLOSE` for floating-point, the instrument you will use to verify every kernel in the book;
- project-wide **error and logging conventions**: `LLM_ASSERT` (free in release, loud in debug), `LLM_CHECK` (always on), `LLM_UNREACHABLE`, and a leveled logger;
- and a **one-command, green-and-clean** boss result proving it all fits together reproducibly.

The engine still does nothing — but the workshop is built, the tools are sharp, and the safety equipment is on. In **Unit 1** we forge the first real component: the **tensor**, with its shape, strides, dtype, shared storage, an aligned arena allocator, and zero-copy views — the data structure every other unit manipulates.

UNIT 1

Tensor & Memory

What you will build. The data structure the entire engine manipulates: a `Tensor` with a shape, strides, a dtype, and a reference to shared storage; an **aligned arena allocator** for the scratch memory a forward pass churns through; and **zero-copy views** — reshape, transpose, permute, slice — that re-describe data without touching a byte of it. The boss proves your views are honest with a hard number: **zero bytes copied.**

Where this fits. Everything above this line — every kernel, every layer, the whole model — is operations on tensors. If the tensor abstraction is wrong or slow, the whole engine is wrong or slow. This is the most-used code you will write, so it is worth getting genuinely right.

A tensor, stripped of mystique, is **a flat buffer of numbers plus a recipe for interpreting it as a multi-dimensional grid.** The buffer is just bytes. The recipe — shape, strides, dtype, offset — is what turns those bytes into "a $2 \times 3 \times 4$ array of 32-bit floats" or "the transpose of that array" or "rows 1 through 3 of it." The deep idea of this unit, the one that makes high-performance tensor code possible, is that *the recipe and the bytes are separate*, so you can change the recipe (reshape, transpose, slice) without rewriting the bytes. Hold onto that; it is the whole game.

We build it in three steps — define the tensor, give it good memory, give it free views — then a boss that holds the views to account.

1.1 The Tensor

GOAL

> Define the core `Tensor` type — shape, strides, dtype, and a shared, aligned storage buffer — with the ability to allocate a contiguous tensor, query its layout, and read/write individual elements for testing.

BACKGROUND

Four pieces of metadata turn a byte buffer into a tensor. Understand each and you understand the abstraction.

Shape. The size along each dimension. A shape of $[2, 3, 4]$ is a 3-dimensional tensor with 2 "matrices" of 3 rows by 4 columns — 24 elements total. The number of dimensions is the *rank* (here, 3). Rank 0 is a scalar; rank 1 a vector; rank 2 a matrix. The total element count is the product of the shape.

Dtype. What kind of number each element is, and therefore how many bytes it occupies. We start with `F32` (4 bytes), `F16` (2 bytes), and `I8` (1 byte). This matters enormously later: the entire point of Unit 9 is to store weights as 4-bit integers to fit big models in small memory. For now, `F32` is the working type; the others are declared so the machinery is dtype-aware from day one. (Block-quantized types like `Q4_0` break the "one element = N bytes" model — their bytes-per-element is defined per *block* — so we handle them specially when we get there. Designing for that now would be premature; designing so it is *possible* is just keeping a `dtype` field.)

Strides — the key idea. A stride is *how many elements you skip in the flat buffer to advance one step along a given dimension.* This is the mechanism that decouples logical shape from physical layout. For a contiguous, **row-major** tensor (the default, and what C and PyTorch use), the last dimension is the fastest-varying: stepping one column moves one element in memory, stepping one row moves a whole row's worth. The strides of a contiguous shape are computed back-to-front: the last dimension has stride 1, and each earlier stride is the product of all the dimensions to its right.

```
shape   = [2, 3, 4]
strides = [12, 4, 1]
              |   |  +- step 1 element   -> move one column
              |   +---- step 4 elements  -> move one row (4 columns)
              +-------- step 12 elements -> move one "matrix" (3x4)
```

```
flat index of element (i, j, k) = i*12 + j*4 + k*1   (+ the view's offset)
```

Once indexing goes through strides, a startling amount becomes *free*. Transpose? Swap two entries in the shape and the strides — the data never moves. Slice off some rows? Adjust the starting offset and shrink one shape entry. We exploit this relentlessly in Challenge 1.3. For now, internalize the indexing formula: **flat index** $= \mathit{offset} + \sum_i (\text{coordinate}_i \times \text{stride}_i)$.

Storage and the storage/view split. The bytes live in a `Storage` object — a reference-counted, aligned buffer. A `Tensor` does not *own* a buffer outright; it holds a `shared_ptr<Storage>` plus its own shape/strides/offset. Two tensors can point at the *same* `Storage` while describing it differently (one as a matrix, the other as its transpose). The buffer is freed when the last tensor referencing it is destroyed — RAII via `shared_ptr`, no manual lifetime juggling. This shared-storage design is precisely what makes zero-copy views possible: a view is just another `Tensor` pointing at the same `Storage` with a different recipe.

Contiguity. A tensor is *contiguous* when its elements sit in memory in the exact order row-major iteration would visit them — i.e. its strides equal the freshly-computed contiguous strides for its shape. Freshly-allocated tensors are contiguous; transposed views generally are not. Contiguity matters because the fast kernels (and some operations like reshape) assume it. (One subtlety, standard across frameworks: dimensions of size 1 impose no constraint on their stride — there is only one position along them — so a correct contiguity check skips size-1 dims.)

THE TASK

Implement, across `include/llm/dtype.hpp`, `include/llm/tensor.hpp`, and `src/tensor.cpp`:

1. `enum class DType { F32, F16, I8 }` with `dtype_size(DType)` and `dtype_name(DType)`.
2. Free helpers `int64_t numel(const Shape&)` (product of dims; `1` for a scalar) and `Strides contiguous_strides(const Shape&)` (row-major).
3. A `Storage` class: an aligned byte buffer (use the `aligned_malloc`/ `aligned_free` helpers — we motivate the alignment in 1.2), non-copyable, that frees itself. Include a static `alloc_count()` counter (incremented per construction) — instrumentation the boss will use to prove views do not allocate.

4. A `Tensor` class holding `shared_ptr<Storage>`, an element `offset`, a `Shape`, a `Strides`, and a `DType`, with: `static Tensor empty(Shape, DType, alignment=64)`; layout queries (`shape`, `strides`, `dtype`, `ndim`, `numel`, `nbytes`, `is_contiguous`); a templated `data_ptr<T>()` returning a pointer to this tensor's first element (including the offset); and a templated `at<T>({i, j, ...})` for element read/write (for tests), bounds-checked via `LLM_ASSERT`.

Use `Shape = std::vector<int64_t>` and `Strides = std::vector<int64_t>` for clarity (the production alternative is in Going Further).

HINTS

Hint 1 — why offset is in elements, and what data_ptr returns

Store the view's `offset_` as a count of *elements* (not bytes). Then `data_ptr<T>()` is simply `reinterpret_cast<T*>(storage->data()) + offset_`. Because `data_ptr` already folds in the offset, your `at()` indexing should add only $\sum_i \text{coord}_i \cdot \text{stride}_i$ *relative to* `data_ptr` — do not add the offset twice.

Hint 2 — computing contiguous strides

Walk the shape from the last dimension to the first with a running product starting at 1: keep `acc = 1`, and for `i` from `ndim-1` down to `0` set `stride[i] = acc; acc *= shape[i];`.

Hint 3 — the contiguity check and size-1 dimensions

Reconstruct the expected contiguous stride on the fly while scanning right-to-left with a running product `expected` (starting at 1). For each dim, if `shape[i] == 1` skip it; otherwise require `strides[i] == expected`, then multiply `expected` by `shape[i]`. If you never mismatch, it is contiguous.

Hint 4 — keep templated methods in the header

`data_ptr<T>` and `at<T>` are templates, so their definitions must live in the header. The non-template methods (`empty`, `is_contiguous`, the views in 1.3) go in `src/tensor.cpp`, which your `CMakeLists.txt` already lists under `llm_core`.

TESTS

```cpp
// tests/test_tensor.cpp
#include "test_framework.hpp"
#include "llm/tensor.hpp"
using namespace llm;

TEST(tensor_shape_strides_numel) {
    Tensor t = Tensor::empty({2, 3, 4}, DType::F32);
    CHECK(t.ndim() == 3);
    CHECK(t.numel() == 24);
    CHECK(t.nbytes() == 24u * 4u);
    CHECK(t.strides()[0] == 12);   // row-major strides for [2,3,4] are [12,4,1]
    CHECK(t.strides()[1] == 4);
    CHECK(t.strides()[2] == 1);
    CHECK(t.is_contiguous());
    CHECK(t.dtype() == DType::F32);
}

TEST(tensor_set_get_roundtrip) {
    Tensor t = Tensor::empty({2, 2}, DType::F32);
    t.at<float>({0, 0}) = 1.0f;  t.at<float>({0, 1}) = 2.0f;
    t.at<float>({1, 0}) = 3.0f;  t.at<float>({1, 1}) = 4.0f;
    CHECK_CLOSE(t.at<float>({1, 0}), 3.0f, 1e-6);
    const float* p = t.data_ptr<float>();   // contiguous order is 1,2,3,4
    CHECK_CLOSE(p[0], 1.0f, 1e-6);
    CHECK_CLOSE(p[3], 4.0f, 1e-6);
}

TEST(dtype_sizes) {
    CHECK(dtype_size(DType::F32) == 4);
    CHECK(dtype_size(DType::F16) == 2);
    CHECK(dtype_size(DType::I8)  == 1);
}
```

Add `tests/test_tensor.cpp` to the `llm_tests` sources and `src/tensor.cpp` to the `llm_core` sources in `CMakeLists.txt`.

REFERENCE SOLUTION

`include/llm/dtype.hpp`:

```cpp
// include/llm/dtype.hpp
#pragma once
#include <cstddef>
#include <cstdint>
#include "llm/common.hpp"

namespace llm {

// Dense element types for now. Block-quantized types (Q8_0, Q4_0, ...) arrive in
// Unit 9 and need special handling: their bytes-per-element is per block, not
// per scalar, so they don't fit this simple size model.
enum class DType : uint8_t { F32, F16, I8 };

inline size_t dtype_size(DType dt) {
    switch (dt) {
        case DType::F32: return 4;
        case DType::F16: return 2;
        case DType::I8:  return 1;
    }
    LLM_UNREACHABLE();
}

inline const char* dtype_name(DType dt) {
    switch (dt) {
        case DType::F32: return "f32";
        case DType::F16: return "f16";
        case DType::I8:  return "i8";
    }
    LLM_UNREACHABLE();
}

}  // namespace llm
```

`include/llm/memory.hpp` (the aligned-allocation helpers; we add the `Arena` in 1.2):

```cpp
// include/llm/memory.hpp
#pragma once
#include <cstddef>
#include <cstdint>
#include <cstdlib>
#if defined(_WIN32)
  #include <malloc.h>   // _aligned_malloc / _aligned_free
#endif
#include "llm/common.hpp"
```

```
namespace llm {

inline bool is_power_of_two(size_t x) { return x != 0 && (x & (x - 1)) == 0; }

// Allocate `nbytes` with the given (power-of-two) alignment. Pair with
    ↪ aligned_free.
// (Why 64 bytes by default? See Challenge 1.2 -- cache lines and SIMD.)
inline void* aligned_malloc(size_t nbytes, size_t alignment = 64) {
    LLM_ASSERT(is_power_of_two(alignment), "alignment must be a power of two");
#if defined(_WIN32)
    return _aligned_malloc(nbytes ? nbytes : 1, alignment);
#else
    // NOTE: std::aligned_alloc (C++17) requires size be a multiple of alignment,
    // so we round up. (It is also not provided by MSVC's runtime, which is why
    // Windows uses _aligned_malloc above.)
    const size_t rounded = (nbytes + alignment - 1) & ~(alignment - 1);
    return std::aligned_alloc(alignment, rounded ? rounded : alignment);
#endif
}

inline void aligned_free(void* p) noexcept {
#if defined(_WIN32)
    _aligned_free(p);
#else
    std::free(p);
#endif
}

}  // namespace llm
```

`include/llm/tensor.hpp`:

```
// include/llm/tensor.hpp
#pragma once
#include <cstdint>
#include <initializer_list>
#include <memory>
#include <vector>

#include "llm/common.hpp"
#include "llm/dtype.hpp"
#include "llm/memory.hpp"

namespace llm {
```

```cpp
using Shape   = std::vector<int64_t>;
using Strides = std::vector<int64_t>;

int64_t numel(const Shape& shape);               // product of dims; 1 for a scalar
    ↪
Strides contiguous_strides(const Shape& shape);  // row-major strides

// Raw, reference-counted, aligned byte buffer. Tensors hold a shared_ptr to one,
// so views share storage and the buffer is freed when the last view dies.
class Storage {
public:
    explicit Storage(size_t nbytes, size_t alignment = 64);
    ~Storage();
    Storage(const Storage&)            = delete;
    Storage& operator=(const Storage&) = delete;

    uint8_t*       data()       noexcept { return data_; }
    const uint8_t* data() const noexcept { return data_; }
    size_t         nbytes() const noexcept { return nbytes_; }

    // Instrumentation (single-threaded): how many Storages have ever been built.
    // The boss challenge uses this to prove view ops allocate nothing.
    static uint64_t alloc_count() noexcept { return s_alloc_count; }

private:
    uint8_t* data_   = nullptr;
    size_t   nbytes_ = 0;
    inline static uint64_t s_alloc_count = 0;
};

class Tensor {
public:
    Tensor() = default;
    static Tensor empty(Shape shape, DType dtype, size_t alignment = 64);

    // ---- layout queries ----
    const Shape&   shape()   const noexcept { return shape_; }
    const Strides& strides() const noexcept { return strides_; }
    DType          dtype()   const noexcept { return dtype_; }
    size_t         ndim()    const noexcept { return shape_.size(); }
    int64_t        numel()   const noexcept { return llm::numel(shape_); }
    size_t  nbytes() const noexcept {
        return static_cast<size_t>(numel()) * dtype_size(dtype_);
    }
    bool is_contiguous() const noexcept;
    bool defined()       const noexcept { return storage_ != nullptr; }
```

```cpp
    // ---- data access ----
    // Pointer to *this tensor's* first element (already includes the view offset).
    template <typename T> T* data_ptr() {
        return reinterpret_cast<T*>(storage_->data()) + offset_;
    }
    template <typename T> const T* data_ptr() const {
        return reinterpret_cast<const T*>(storage_->data()) + offset_;
    }

    // Element access for tests/debugging only (strided, branchy -- not hot paths).
    template <typename T> T& at(std::initializer_list<int64_t> idx) {
        return data_ptr<T>()[rel_offset(idx)];
    }
    template <typename T> const T& at(std::initializer_list<int64_t> idx) const {
        return data_ptr<T>()[rel_offset(idx)];
    }

    // ---- zero-copy views (Challenge 1.3) ----
    Tensor reshape(Shape new_shape) const;   // requires contiguous; shares storage
    Tensor view(Shape new_shape) const { return reshape(std::move(new_shape)); }
    Tensor permute(const std::vector<int>& dims) const;
    Tensor transpose(int dim0, int dim1) const;
    Tensor slice(int dim, int64_t start, int64_t stop) const;

    // Materialize a contiguous copy (allocates iff not already contiguous).
    Tensor contiguous() const;

    // Identity helpers for tests: tensors sharing storage return the same pointer.
    const void* storage_id() const noexcept { return storage_.get(); }
    int64_t     offset()     const noexcept { return offset_; }

private:
    int64_t rel_offset(std::initializer_list<int64_t> idx) const;

    std::shared_ptr<Storage> storage_;
    int64_t offset_ = 0;            // in elements, from start of storage
    Shape    shape_;
    Strides  strides_;
    DType    dtype_  = DType::F32;
};

}  // namespace llm
```

`src/tensor.cpp` (the non-template methods; the view methods are added in 1.3):

```
// src/tensor.cpp
#include "llm/tensor.hpp"
#include <cstring>
#include <utility>

namespace llm {

int64_t numel(const Shape& shape) {
    int64_t n = 1;
    for (int64_t d : shape) n *= d;
    return n;   // empty shape -> 1 (a scalar)
}

Strides contiguous_strides(const Shape& shape) {
    Strides s(shape.size());
    int64_t acc = 1;
    for (int i = static_cast<int>(shape.size()) - 1; i >= 0; --i) {
        s[i] = acc;
        acc *= shape[i];
    }
    return s;
}

// ---- Storage ----
Storage::Storage(size_t nbytes, size_t alignment) : nbytes_(nbytes) {
    data_ = static_cast<uint8_t*>(aligned_malloc(nbytes, alignment));
    LLM_CHECK(data_ != nullptr, "Storage: allocation failed");
    ++s_alloc_count;
}
Storage::~Storage() { aligned_free(data_); }

// ---- Tensor ----
Tensor Tensor::empty(Shape shape, DType dtype, size_t alignment) {
    Tensor t;
    t.dtype_   = dtype;
    t.shape_   = std::move(shape);
    t.strides_ = contiguous_strides(t.shape_);
    t.offset_  = 0;
    const size_t bytes =
        static_cast<size_t>(llm::numel(t.shape_)) * dtype_size(dtype);
    t.storage_ = std::make_shared<Storage>(bytes, alignment);
    return t;
}
```

```cpp
bool Tensor::is_contiguous() const noexcept {
    int64_t expected = 1;
    for (int i = static_cast<int>(ndim()) - 1; i >= 0; --i) {
        if (shape_[i] == 1) continue;          // size-1 dims impose no constraint
        if (strides_[i] != expected) return false;
        expected *= shape_[i];
    }
    return true;
}

int64_t Tensor::rel_offset(std::initializer_list<int64_t> idx) const {
    LLM_ASSERT(idx.size() == ndim(), "at(): wrong number of indices");
    int64_t off = 0;
    size_t d = 0;
    for (int64_t i : idx) {
        LLM_ASSERT(i >= 0 && i < shape_[d], "at(): index out of range");
        off += i * strides_[d];
        ++d;
    }
    return off;   // relative to data_ptr(), which already includes offset_
}

}  // namespace llm
```

Why these decisions. `int64_t` for shapes/strides/offsets is deliberate: real models have tensors that overflow 32-bit element counts (a vocabulary of 128k times a hidden size of 4096 is already half a billion elements, and intermediate products go higher). Storing the offset in *elements* and having `data_ptr` fold it in keeps `at()` indexing simple and prevents the classic "added the offset twice" bug. The `s_alloc_count` instrumentation looks like a curiosity now, but it is the meter the boss reads to *prove* — not assert by inspection — that views do not copy.

GOING FURTHER

- **Inline-capacity shape/strides.** `std::vector<int64_t>` heap-allocates, and every view copies two of them. Tensors rarely exceed ~6 dimensions, so production engines use a small-vector with inline storage (a fixed array plus a size, spilling to the heap only past the cap). Swapping `Shape`/`Strides` for such a type later removes a real allocation from the view path — and your boss challenge already isolates *data* allocations from these, so you can measure the win cleanly.
- **`F16` conversions.** Implement `float f16_to_f32(uint16_t)` and `uint16_t`

`f32_to_f16(float)` via bit manipulation (or `std::bit_cast` plus the IEEE-754 half format). You will want them the moment a real model stores anything in half precision.

- **A `fill/arange` helper.** A tiny `Tensor::arange(n)` filling `0,1,2,...` makes every later test more readable.
- **Negative indices.** PyTorch-style negative indexing (`-1` = last) is a five-line change to `rel_offset` and surprisingly nice when debugging.

1.2 Aligned Memory and the Arena

GOAL

> Add a **bump (arena) allocator** for the transient memory a forward pass produces and discards, and understand *why* we align everything to 64 bytes. After this, allocating scratch during inference is a pointer increment, and resetting it between tokens is setting one integer to zero.

BACKGROUND

A forward pass is an allocation storm. Every layer produces intermediate activations — the output of a matmul, a normalized vector, an attention score buffer — that are consumed by the next step and then thrown away. If each went through `malloc/free`, you would pay two costs: the raw overhead of the allocator (locking, bookkeeping, searching free lists) on a path you run thousands of times per generated token, and **fragmentation** as differently-sized transient buffers churn. Neither is acceptable in a hot loop.

The fix is an **arena** (a.k.a. bump or linear allocator), beautiful in its simplicity. You allocate one big block up front. To "allocate" N bytes, you return the current position and advance a cursor by N. That is it — no free list, no search, no per-allocation header. You never free individual allocations; instead, when the whole batch of transients is no longer needed (e.g. at the end of a forward pass), you **reset** the arena by setting the cursor back to zero, reclaiming everything at once. The allocation cost is a few instructions; the deallocation cost is *one* instruction for the entire forward pass.

This maps perfectly onto inference's structure: allocate an arena once when the engine starts; during each forward pass, carve activations out of it; `reset()` be-

tween passes. The same physical memory is reused token after token, hot in cache, never returned to the OS. We wire the forward pass to an arena in Unit 5; here we build the allocator and prove it behaves.

Now, alignment — why 64 bytes? Two hardware realities. *Cache lines:* CPUs move memory in fixed-size chunks called cache lines — **64 bytes** on essentially all modern x86-64 and ARM64. When a buffer's start is aligned to 64, each logical "row" tends to begin at a cache-line boundary, avoiding a subtle penalty: a value straddling two cache lines forces the CPU to touch *both*, doubling memory traffic for that access. *SIMD:* the vector units of Unit 3 (SSE, AVX, AVX-512, NEON) load and store many elements at once — AVX-512 moves a full **64 bytes** (sixteen `float`s) per instruction. These wide loads are happiest, and on some microarchitectures *only correct* for the aligned-load instructions, when the address is a multiple of the vector width. 64 bytes satisfies both the cache line and the widest common SIMD register, which is why it is the alignment you see throughout serious inference engines.

One implementation detail, stated plainly: aligning a *size* up to a multiple of A is `(n + A - 1) & ~(A - 1)` when A is a power of two — clear the low bits after rounding up. We use it both to satisfy `std::aligned_alloc`'s multiple-of-alignment requirement and to align each bump-allocation's starting offset.

THE TASK

Add an `Arena` class to `include/llm/memory.hpp`: construct with a capacity (bytes) and an alignment (default 64), allocating one aligned backing block; `void* allocate(size_t)` aligns the cursor up, checks it still fits (`LLM_CHECK` on overflow), returns the pointer, advances the cursor; `reset()` sets the cursor to zero (remembering the peak via a high-water mark); and `used()`, `high_water()`, `capacity()` queries. Non-copyable; frees its block in the destructor. Every pointer `allocate` returns is aligned, and successive allocations do not overlap.

HINTS

Hint 1 — the bump

Keep `uint8_t* base_`, `size_t capacity_`, and a `size_t offset_` cursor. In `allocate`: compute `aligned_off = (offset_ + alignment_ - 1) & ~(alignment_ - 1)`, check `aligned_off + nbytes <= capacity_`, set the result to `base_ + aligned_off`, then `offset_ = aligned_off + nbytes`.

Hint 2 — high-water mark

After advancing, `high_water_ = max(high_water_, offset_)`. `reset()` zeroes `offset_` but leaves `high_water_`. This tells you the largest amount the arena ever needed simultaneously — invaluable for sizing the arena so it never overflows but does not waste memory.

Hint 3 — alignment must be a power of two

The `& ~(alignment-1)` trick only works for power-of-two alignments. Assert it in the constructor (reuse `is_power_of_two`). Reuse `aligned_malloc`/`aligned_free` for the backing block so the arena's base is itself aligned.

TESTS

```
// add to tests/test_tensor.cpp (or a new tests/test_memory.cpp)
#include "llm/memory.hpp"

TEST(aligned_alloc_returns_aligned_pointer) {
    void* p = aligned_malloc(1000, 64);
    REQUIRE(p != nullptr);
    CHECK(reinterpret_cast<uintptr_t>(p) % 64 == 0);
    aligned_free(p);
}

TEST(arena_aligned_and_nonoverlapping) {
    Arena a(1u << 20, 64);                // 1 MiB
    void* x = a.allocate(100);
    void* y = a.allocate(100);
    CHECK(reinterpret_cast<uintptr_t>(x) % 64 == 0);
    CHECK(reinterpret_cast<uintptr_t>(y) % 64 == 0);
    CHECK(x != y);
    // 100 bytes rounds up to 128 for the next aligned slot, so y is >= 64 past x
    CHECK(reinterpret_cast<char*>(y) - reinterpret_cast<char*>(x) >= 64);
}

TEST(arena_reset_reuses_memory_and_tracks_peak) {
    Arena a(1u << 20, 64);
    void* first = a.allocate(256);
    CHECK(a.used() == 256);
    a.reset();
    CHECK(a.used() == 0);
    void* again = a.allocate(256);
```

```
    CHECK(first == again);              // same address after reset
    CHECK(a.high_water() >= 256);       // peak remembered across the reset
}
```

REFERENCE SOLUTION

Add to `include/llm/memory.hpp`, inside `namespace llm`, after the aligned-allocation helpers:

```
// A bump / arena allocator for transient activations.
// Allocate once; carve out scratch with allocate(); reclaim it all with reset().
class Arena {
public:
    explicit Arena(size_t capacity_bytes, size_t alignment = 64)
        : alignment_(alignment), capacity_(capacity_bytes) {
        LLM_ASSERT(is_power_of_two(alignment), "alignment must be a power of two");
↪
        base_ = static_cast<uint8_t*>(aligned_malloc(capacity_bytes, alignment));
        LLM_CHECK(base_ != nullptr, "Arena: backing allocation failed");
    }
    ~Arena() { aligned_free(base_); }
    Arena(const Arena&)            = delete;
    Arena& operator=(const Arena&) = delete;

    void* allocate(size_t nbytes) {
        const size_t aligned_off = (offset_ + alignment_ - 1) & ~(alignment_ - 1);
        LLM_CHECK(aligned_off + nbytes <= capacity_, "Arena: out of memory");
        void* p  = base_ + aligned_off;
        offset_  = aligned_off + nbytes;
        if (offset_ > high_water_) high_water_ = offset_;
        return p;
    }

    void   reset()      noexcept { offset_ = 0; }       // keep high_water_
    size_t used()       const noexcept { return offset_; }
    size_t high_water() const noexcept { return high_water_; }
    size_t capacity()   const noexcept { return capacity_; }

private:
    uint8_t* base_       = nullptr;
    size_t   alignment_  = 64;
    size_t   capacity_   = 0;
    size_t   offset_     = 0;
    size_t   high_water_ = 0;
```

```
};
```

That is the whole allocator. Allocation is now a couple of arithmetic ops and a pointer add, and freeing a forward pass's worth of activations is a single `reset()`.

Why it matters later. When we build the forward pass, we hand it an `Arena` and every intermediate tensor is carved from it instead of heap-allocated. The high-water mark tells us exactly how large to make that arena: run a forward pass at the maximum sequence length once, read `high_water()`, and size the arena to that. The result is an inference loop that, after warm-up, performs **zero heap allocations per token** — a property you can verify with the same `Storage::alloc_count()` meter from 1.1.

GOING FURTHER

- **Scoped sub-arenas.** Add a `Marker mark()` returning the current offset and a `void rewind(Marker)` restoring it. A function can then allocate temporaries and free *just its own* on return — a nested lifetime within the linear one.
- **Debug guard bytes.** In Debug builds, write a known pattern (`0xCD`) into freshly bumped regions, or place guard bytes between allocations and check them on reset, to catch buffer overruns in your kernels.
- **Over-alignment for specific tensors.** Let `allocate(nbytes, align)` take a per-call alignment $\geq$ the arena's, so individual tensors can be more strongly aligned than the default.

1.3 Views and Reshape (Zero-Copy)

GOAL

> Implement the operations that re-describe a tensor without moving data — `reshape`, `permute`, `transpose`, `slice` — as **views** that share storage with the original, plus a `contiguous()` that materializes a real copy only when one is genuinely needed.

BACKGROUND

This is the payoff for the storage/recipe separation. Each of these operations produces a new `Tensor` that points at the *same* `Storage` and differs only in shape, strides, and offset. No bytes are read or written. That is not just a memory optimization — it is a throughput optimization, because the data stays exactly where it is, hot in cache, while you change your mind about how to read it.

`reshape` reinterprets the same elements under a new shape with the same total count: a contiguous $[2, 6]$ becomes $[3, 4]$ or $[12]$ by recomputing contiguous strides. The catch: reshape-as-a-view is only valid when the tensor is **contiguous**, because it assumes the elements are in plain row-major order. If they are not (say you transposed first), there is no stride pattern that expresses the new shape over the old physical layout, so a copy is unavoidable. We make this explicit: `reshape` requires contiguity and tells you to call `contiguous()` first. (Many frameworks auto-copy; making it explicit teaches the invariant, and an engine generally *wants* to know when a copy is about to happen.)

`transpose(d0, d1)` swaps two dimensions by swapping their shape entries *and* their stride entries. The data does not move — you have changed which direction counts as "rows" and which as "columns." The result is almost always non-contiguous, which is fine: strided kernels read it correctly via the indexing formula. This is the operation behind reading the same weight matrix as W or $W^\top$ for free.

`permute(order)` is the general case: an arbitrary reordering of all dimensions. Transpose is just permute restricted to swapping two axes. We use permute constantly in attention, where activations get reshaped to `[batch, heads, seq, head_dim]` and the axes shuffled so each head's data is laid out for a matmul.

`slice(dim, start, stop)` takes a sub-range along one dimension. The trick: advance the **offset** by `start * stride[dim]` to point at the first element of the slice, and shrink `shape[dim]` to `stop - start`. The strides are unchanged — you are looking at a window into the same buffer. This is how we will grab "the keys for positions 0 through t" out of the KV cache without copying them.

`contiguous()` is the escape hatch. When you genuinely need elements laid out densely, it allocates a fresh buffer and copies the elements in row-major order, walking the source via its strides. If the tensor is *already* contiguous, it is a no-op that returns itself — no allocation. The art of fast tensor code is, in large part, calling `contiguous()` as rarely as possible: every call is a copy, and copies are what views exist to avoid.

The invariant tying it together: **a view shares storage; a copy allocates new storage.** You should be able to point at any line and say which it is. The boss will make you prove it.

THE TASK

Implement in `src/tensor.cpp` (declarations already in `tensor.hpp`): `reshape` (require equal element count and contiguity; share storage; recompute contiguous strides; keep the offset); `permute` (validate it is a true permutation; reorder shape and strides; share storage); `transpose` (swap the two shape and stride entries; share storage); `slice` (validate the range; advance offset by `start*strides[dim]`; set `shape[dim] = stop-start`; share storage); and `contiguous` (return `*this` if already contiguous; else allocate and copy by walking the source strides). Each view op leaves `storage_id()` unchanged and copies zero bytes; `contiguous()` on a non-contiguous tensor is the only one that allocates.

HINTS

Hint 1 — a view op is "copy *this, then edit the metadata"

For all four view ops, start with `Tensor out = *this;`. Copying a `Tensor` copies the `shared_ptr` (bumping the storage refcount — same bytes) and the small shape/stride vectors. Then mutate only `out.shape_`, `out.strides_`, and/or `out.offset_`. The storage pointer rides along untouched.

Hint 2 — slice changes the offset, not the strides

Slicing along `dim` means "skip the first `start` elements in that direction": `offset_ += start * strides_[dim]`. The strides stay the same, and only `shape_[dim]` shrinks. Convince yourself with a 4x4: slicing rows `[1,3)` should land `at({0,0})` on the original `at({1,0})`.

Hint 3 — the contiguous() copy loop

The output is contiguous with offset 0, so its element `k` is at byte `k * element_size`. For each linear output index, find the matching source element: walk an N-dimensional coordinate counter (last dimension fastest), compute the source element index `offset_ + sum(coord_i * stride_i)`, and `memcpy` one element's bytes. Increment the coordinate like an odome-

ter.

Hint 4 — validating a permutation

A valid permutation of N dims uses each index in $[0, N)$ exactly once. Track a seen array; for each requested dim, check it is in range and not already seen, then mark it.

TESTS

```
// add to tests/test_tensor.cpp

TEST(reshape_zero_copy_preserves_data) {
    Tensor t = Tensor::empty({2, 6}, DType::F32);
    for (int i = 0; i < 12; ++i) t.data_ptr<float>()[i] = float(i);
    Tensor r = t.reshape({3, 4});
    CHECK(r.storage_id() == t.storage_id());        // SAME storage -> no copy
    CHECK(r.is_contiguous());
    CHECK_CLOSE(r.at<float>({2, 3}), 11.0f, 1e-6);  // last element preserved
    r.at<float>({0, 0}) = 99.0f;                    // mutate the view...
    CHECK_CLOSE(t.at<float>({0, 0}), 99.0f, 1e-6);  // ...base sees it (same bytes)
    ↪
}

TEST(transpose_swaps_strides_zero_copy) {
    Tensor t = Tensor::empty({2, 3}, DType::F32);
    for (int i = 0; i < 6; ++i) t.data_ptr<float>()[i] = float(i);
    Tensor tt = t.transpose(0, 1);                  // shape {3,2}
    CHECK(tt.storage_id() == t.storage_id());
    CHECK(tt.shape()[0] == 3 && tt.shape()[1] == 2);
    CHECK(!tt.is_contiguous());
    for (int i = 0; i < 2; ++i)                      // A[i][j] == A^T[j][i]
        for (int j = 0; j < 3; ++j)
            CHECK_CLOSE(t.at<float>({i, j}), tt.at<float>({j, i}), 1e-6);
}

TEST(slice_views_subregion) {
    Tensor t = Tensor::empty({4, 4}, DType::F32);
    for (int i = 0; i < 16; ++i) t.data_ptr<float>()[i] = float(i);
    Tensor rows = t.slice(0, 1, 3);                 // rows 1..2 -> shape {2,4}
    CHECK(rows.storage_id() == t.storage_id());
    CHECK(rows.shape()[0] == 2 && rows.shape()[1] == 4);
    CHECK_CLOSE(rows.at<float>({0, 0}),  4.0f, 1e-6);  // == t[1][0]
```

```
    CHECK_CLOSE(rows.at<float>({1, 3}), 11.0f, 1e-6);  // == t[2][3]
}

TEST(contiguous_materializes_a_transpose) {
    Tensor t = Tensor::empty({2, 3}, DType::F32);
    for (int i = 0; i < 6; ++i) t.data_ptr<float>()[i] = float(i);
    Tensor c = t.transpose(0, 1).contiguous();      // {3,2}, contiguous, real copy
    ↪
    CHECK(c.is_contiguous());
    CHECK(c.storage_id() != t.storage_id());        // different storage -> a copy
    const float* p = c.data_ptr<float>();           // memory order now
    ↪ 0,3,1,4,2,5
    CHECK_CLOSE(p[0], 0.0f, 1e-6);
    CHECK_CLOSE(p[1], 3.0f, 1e-6);
    CHECK_CLOSE(p[2], 1.0f, 1e-6);
}

TEST(reshape_requires_contiguous) {
    Tensor t  = Tensor::empty({2, 3}, DType::F32);
    Tensor tt = t.transpose(0, 1);                  // non-contiguous
    Tensor ok = tt.contiguous().reshape({6});       // the supported path
    CHECK(ok.numel() == 6);
    CHECK(ok.is_contiguous());
}
```

REFERENCE SOLUTION

Add to `src/tensor.cpp` (inside `namespace llm`):

```
Tensor Tensor::reshape(Shape new_shape) const {
    LLM_CHECK(llm::numel(new_shape) == numel(),
              "reshape: element count must be preserved");
    LLM_CHECK(is_contiguous(),
              "reshape: tensor must be contiguous; call contiguous() first");
    Tensor out    = *this;                // shares storage_ (shared_ptr), same
    ↪ offset_
    out.shape_    = std::move(new_shape);
    out.strides_ = contiguous_strides(out.shape_);
    return out;
}

Tensor Tensor::permute(const std::vector<int>& dims) const {
    LLM_CHECK(dims.size() == ndim(), "permute: need one entry per dimension");
    Shape   ns(ndim());
```

```
    Strides nst(ndim());
    std::vector<bool> seen(ndim(), false);
    for (size_t i = 0; i < dims.size(); ++i) {
        const int d = dims[i];
        LLM_CHECK(d >= 0 && d < static_cast<int>(ndim()) && !seen[d],
                  "permute: invalid permutation");
        seen[d] = true;
        ns[i]   = shape_[d];
        nst[i]  = strides_[d];
    }
    Tensor out   = *this;
    out.shape_   = std::move(ns);
    out.strides_ = std::move(nst);
    return out;
}

Tensor Tensor::transpose(int dim0, int dim1) const {
    LLM_CHECK(dim0 >= 0 && dim0 < static_cast<int>(ndim()) &&
              dim1 >= 0 && dim1 < static_cast<int>(ndim()),
              "transpose: dim out of range");
    Tensor out = *this;
    std::swap(out.shape_[dim0],   out.shape_[dim1]);
    std::swap(out.strides_[dim0], out.strides_[dim1]);
    return out;
}

Tensor Tensor::slice(int dim, int64_t start, int64_t stop) const {
    LLM_CHECK(dim >= 0 && dim < static_cast<int>(ndim()), "slice: dim out of range
   ↪ ");
    LLM_CHECK(0 <= start && start <= stop && stop <= shape_[dim],
              "slice: range out of bounds");
    Tensor out      = *this;
    out.offset_     = offset_ + start * strides_[dim];   // shift into the buffer
    out.shape_[dim] = stop - start;                      // shrink that dimension
    // strides unchanged
    return out;
}

Tensor Tensor::contiguous() const {
    if (is_contiguous()) return *this;                   // no copy needed
    Tensor out      = Tensor::empty(shape_, dtype_);
    const size_t es = dtype_size(dtype_);
    const uint8_t* src = storage_->data();
    uint8_t*       dst = out.storage_->data();
    const int64_t  n   = numel();
    std::vector<int64_t> coord(ndim(), 0);
    for (int64_t linear = 0; linear < n; ++linear) {
```

```
        int64_t src_elem = offset_;
        for (size_t d = 0; d < ndim(); ++d) src_elem += coord[d] * strides_[d];
        std::memcpy(dst + static_cast<size_t>(linear)   * es,
                    src + static_cast<size_t>(src_elem) * es, es);
        // odometer increment: last axis fastest
        for (int d = static_cast<int>(ndim()) - 1; d >= 0; --d) {
            if (++coord[d] < shape_[d]) break;
            coord[d] = 0;
        }
    }
    return out;
}
```

The mental model to keep. Reading these five functions back to back, notice that four of them are *pure metadata edits over shared storage* and exactly one touches bytes. That ratio — describe-don't-copy by default, materialize only on demand — is the entire design philosophy of fast tensor libraries, and it is why the next units can shuffle activations into whatever shape a kernel wants without drowning in memcpys.

GOING FURTHER

- **Broadcasting.** Implement `expand`, which sets a size-1 dimension's *stride to 0* so iterating that axis re-reads the same element — the trick behind adding a bias vector to a matrix without copying. You will use it for bias adds and the attention mask.
- **`as_strided` / `unfold`.** A general "give me a view with these explicit shape and strides" primitive unlocks sliding-window views.
- **Negative-stride reversal.** A `flip(dim)` that points the offset at the last element and negates that dimension's stride reverses an axis with zero copies — a good check on whether your indexing is truly stride-driven.
- **Combine and stress.** Chain `slice` → `transpose` → `slice` and verify against a hand-computed layout.

1.4 Zero-Copy Discipline

BOSS CHALLENGE

A measured guarantee. Not “the views look right,” but a hard count proving they copy nothing: across a pipeline of slices, transposes, reshapes, and a thousand views, the number of data allocations stays constant — and a transpose-as-view is still element-for-element correct.

GOAL

> Prove, with a meter rather than an inspection, that your view operations are genuinely zero-copy: across a pipeline of slices, transposes, reshapes, and a thousand successive views, the number of **data allocations stays constant** and the **storage stays shared** — while a transpose-as-view satisfies $A[i][j] = A^{\top}[j][i]$ for every element.

BACKGROUND

Anyone can write a `transpose` that *looks* zero-copy. The discipline — the thing that separates an engine you can trust from one you hope is fast — is being able to *measure* that it copied nothing, so a future “optimization” that quietly introduces a copy gets caught immediately. This boss installs that habit.

The instrument is `Storage::alloc_count()` from Challenge 1.1: it ticks once per data-buffer allocation. A correct view operation never constructs a `Storage`, so across any sequence of views the count must not move. We anchor on a `before` reading and check *deltas*, so the test is robust no matter what other tests allocated first.

One honest caveat, stated because the expert reader will wonder about it: creating a view *does* still allocate the small `shape` and `strides` `std::vectors`. Those are *metadata* allocations, not *data* copies, and they are exactly what the inline-capacity small-vector extension from 1.1 removes. The boss deliberately measures `Storage` allocations — the megabyte-scale data buffers that actually matter — and not those few-dozen-byte vectors. Naming this precisely is itself part of the discipline.

THE TASK

Write a single test, `boss_zero_copy_discipline`, that: records `Storage::alloc_count()` as `before`; allocates one base tensor and asserts the count rose by exactly **1**; builds a pipeline of pure views (`slice` → `transpose` → `slice`) and asserts the count is *still* `before + 1` and the result *shares storage*; reshapes the contiguous base and asserts no new allocation; takes **1000** successive views in a loop and asserts the count is unchanged; then builds a fresh tensor, transposes it as a view, asserts the transpose added **zero** allocations, and verifies $A[i][j] = A^{\top}[j][i]$ across *all* elements.

The bar (the boss): every view step adds **zero** `Storage` allocations, storage identity is preserved through the whole pipeline, and the transpose-as-view is element-for-element correct. If the meter ever moves on a view op, you have a hidden copy — fix it.

HINTS

Hint 1 — anchor on deltas, not absolutes

Other tests run first and allocate, so never assert `alloc_count() == 1`. Capture `const uint64_t before = Storage::alloc_count();` at the top and assert `alloc_count() == before + k` for the small, known `k`.

Hint 2 — a guaranteed-view "no-op" slice for the 1000-loop

`t.slice(dim, 0, t.shape()[dim])` selects the whole dimension — logically a no-op, but it still goes through the view path and produces a new `Tensor` sharing storage. Reassign `t` to it 1000 times; the `Storage` count must not budge.

Hint 3 — what failure looks like

If `alloc_count()` jumps inside the pipeline, some "view" is calling `Tensor::empty` or `contiguous()` internally. The usual culprit is a `reshape` that silently materialized, or a slice/transpose that copied instead of editing metadata. The meter points you straight at the offending op.

TESTS

```
// add to tests/test_tensor.cpp

TEST(boss_zero_copy_discipline) {
    const uint64_t before = Storage::alloc_count();

    Tensor base = Tensor::empty({1024, 1024}, DType::F32);   // (1 allocation)
    CHECK(Storage::alloc_count() == before + 1);

    // A pipeline of pure views: none may allocate data or copy bytes.
    Tensor v = base.slice(0, 0, 512)     // {512,1024}
                   .transpose(0, 1)      // {1024,512}
                   .slice(0, 0, 256);    // {256,512}
    CHECK(Storage::alloc_count() == before + 1);     // still just the base
    CHECK(v.storage_id() == base.storage_id());      // same bytes

    Tensor r = base.reshape({1024 * 1024});          // contiguous reshape: zero-
    ↪ copy
    CHECK(Storage::alloc_count() == before + 1);
    CHECK(r.storage_id() == base.storage_id());

    Tensor t = base;                                 // 1000 views allocate no
    ↪ data
    for (int i = 0; i < 1000; ++i) t = t.slice(1, 0, t.shape()[1]);
    CHECK(Storage::alloc_count() == before + 1);
    CHECK(t.storage_id() == base.storage_id());

    Tensor a = Tensor::empty({64, 48}, DType::F32);  // (+1 allocation)
    for (int i = 0; i < 64 * 48; ++i) a.data_ptr<float>()[i] = float(i);
    const uint64_t after_a = Storage::alloc_count();
    Tensor at = a.transpose(0, 1);                   // {48,64}
    CHECK(Storage::alloc_count() == after_a);        // transpose copied nothing
    CHECK(at.storage_id() == a.storage_id());

    bool all_match = true;
    for (int i = 0; i < 64 && all_match; ++i)
        for (int j = 0; j < 48; ++j)
            if (a.at<float>({i, j}) != at.at<float>({j, i})) { all_match = false;
    ↪ break; }
    CHECK(all_match);
}
```

REFERENCE SOLUTION

There is no new production code to write — the boss is cleared by the Unit 1 implementations behaving correctly under the meter. The test above *is* the deliverable. Build and run:

```
./scripts/build.sh        # or: cmake --build build && ./build/llm_tests
```

Expected: `boss_zero_copy_discipline` reports `[ OK ]`, and the full suite ends `0 failed`. If it does not, the failing `CHECK` tells you precisely which view operation introduced a copy — that is the boss doing its job.

What you have actually proven. Not "I wrote views" but "across a realistic pipeline plus a thousand-iteration stress, my view operations performed exactly zero data allocations, kept storage shared throughout, and produced bit-correct transposed access." That is a property you can now *defend with a number* — and re-check after any future change to the tensor code. This is the texture of the whole book: build the thing, then make a measurement assert that it is right.

GOING FURTHER

- **Forward-pass dry run.** Combine this with the `Arena`: allocate one arena, simulate carving a few activation tensors per "step," `reset()` between steps, and assert via `Storage::alloc_count()` that after the first step no further data allocations occur — a preview of the zero-allocation inference loop you achieve for real in Unit 5/14.
- **Metadata-allocation count.** Add a second counter for shape/stride vector allocations and watch the inline-small-vector extension drive *it* to zero across the 1000-view loop.
- **Thread-safety note.** `s_alloc_count` is a plain `uint64_t`, fine for single-threaded tests. When Unit 3 introduces threads, make it `std::atomic` (or scope the meter to a single thread) so the instrument itself is not a data race.

Unit 1 — Milestone check

MILESTONE CHECK

You now have the foundation every kernel and layer will stand on:

- a `Tensor` type with shape, strides, dtype, and `shared_ptr<Storage>`, that knows its element count, byte size, and contiguity, and can read/write elements for testing;
- **aligned allocation** and an **arena allocator** that turns per-forward-pass scratch into a pointer bump and a single `reset()` — the basis for zero-allocation inference;
- **zero-copy views** — `reshape`, `permute`, `transpose`, `slice` — that re-describe data for free, with `contiguous()` as the explicit, measurable escape hatch when a real copy is required;
- and a **boss-grade guarantee**, backed by an allocation meter, that your views copy zero bytes of data.

The engine can now *hold and reshape* numbers. In **Unit 2** we make it *compute* with them: the elementwise math, the activation functions every Transformer needs (SiLU, GELU, and the gated SwiGLU), RMSNorm, and a numerically-stable softmax — the kernels that, together with matrix multiplication in Unit 3, do the actual arithmetic of a forward pass.

End of the Free Sample

This free sample includes the front matter, **Unit 0** (**Foundation & Setup**), and **Unit 1** (**Tensor & Memory**) in full — enough to see how every challenge works, end to end, from goal and background through the tests and the reference solution.

The complete book continues through **Units 2–15**: the math kernels and activation functions, matrix multiplication from naive to SIMD, the byte-level BPE tokenizer, the full Transformer (RoPE, attention, GQA/MQA, SwiGLU), Mixture of Experts, the KV cache, Flash Attention, int8/int4 quantization, GGUF model loading, sampling and generation, speculative decoding, and serving with continuous batching — building, piece by piece, to a complete inference engine that loads a real model and generates text on the CPU (the "engine is alive" milestone in Unit 14), with an optional CUDA capstone in Unit 15.

Get the complete book

leanpub.com/*your-book-url*