

A SYSTEMS PROGRAMMING BOOK

FlatBuffers & FlexBuffers

From Bytes to Mutable Memory

Memory layout, C++ implementation, protocol design,
hardware-aware engineering, and a schemaless
mutable-memory capstone.

03

01

02

03

04

04

04

06

28

01

ONE COMPLETE FLEXBUFFER / UNTYPED VECTOR [1, 2, 3]

30 CHAPTERS · 32 EXERCISES · C++20 SOURCE

FIRST EDITION / 18 SEPTEMBER 2026

Contents

Start with bytes and ownership; follow the two wire formats; then build a mutable execution representation.

Preface What this book will teach you	4
1 Two representations, two jobs	6
2 Bytes, alignment, integers, and floating point	9
3 A safe C++ vocabulary for byte formats	12
4 FlatBuffers: the root, the table, and the vtable	15
5 A complete 32-byte FlatBuffer	18
6 Structs, vectors, strings, unions, and framing	21
7 Reconstructing the FlatBuffers builder	24
8 Schema evolution, generated code, and reflection	27
9 FlatBuffers verification and the boundary of mutation	29
10 FlexBuffers starts at the end	31
11 Widths are a small layout solver	34
12 Keys, strings, and blobs are different objects	37
13 Every vector family, one at a time	40
14 A map is two coordinated vectors	43
15 Building a reader as a set of views	46
16 Building the writer from first principles	49
17 Mutation, sharing, and why one outlier matters	52

18 Verification is a small security boundary	55
19 What “close to official quality” actually requires	58
20 Hardware-aware optimization without unsafe shortcuts	61
21 What modern languages help with—and what they do not	64
22 Benchmark the workload, not the advertisement	67
23 Optimizations worth earning	70
24 Capstone: choose the compatibility contract first	72
25 The 16-byte cell and the 32-byte descriptor	75
26 Allocation, roots, reclamation, and compaction	78
27 Fast containers without a declared schema	81
28 Transactions, borrows, and concurrent snapshots	85
29 Freeze and thaw without changing the standard	88
30 Build the capstone in stages, then earn its speed	91
Appendix A Type codes and byte equations	95
Appendix B Exercise solutions and implementation hints	99
Appendix C Reading the official implementation	104
Appendix D Complete checked FlexBuffers laboratory	108
Appendix E Complete mutable semantic laboratory	117
Appendix F Complete FlatBuffers table laboratory	122
Appendix G Sources and edition notes	124

BEFORE YOU BEGIN

What this book will teach you

A serialization format is an agreement about bytes. A runtime data structure is an agreement about ownership, operations, and cost. They can share a representation, but they are not the same problem. This book follows that distinction all the way from a three-byte integer to the design of a mutable, schemaless memory engine.

The request that led to this book named “FlexBuffers” twice. This edition treats that as FlatBuffers and FlexBuffers, with the greater emphasis on FlexBuffers. FlatBuffers provides a particularly useful contrast: it makes many decisions at schema-compilation time that FlexBuffers has to represent in the data itself. Understanding both makes the design space much easier to see.

The goal is not to memorize an API. It is to become capable of looking at a buffer, reconstructing its meaning, implementing a reader and writer, identifying the invariants that make them safe, and deciding where the official representation is the right execution representation—and where it is not.

The source baseline

The implementation discussion is pinned to the Google FlatBuffers repository at commit `b8431fbcd7a5c71817f314e18b332c0648554efa`, dated 14 September 2026. The main source is `include/flatbuffers/flexbuffers.h`; the FlatBuffers chapters also inspect `table.h` and `flatbuffer_builder.h`. This is a source snapshot, not a claim that a particular release package contains every feature described. Public documentation and the source sometimes differ in emphasis. Where that matters, the book names the implementation behavior explicitly. [1–3]

Three kinds of material appear throughout the book. **Wire-format facts** describe what bytes mean. **Implementation observations** describe the pinned C++ source. **Design proposals** describe the new mutable engine developed here. The last category is deliberately not presented as an upstream feature or a measured performance result.

The companion and its limits

The accompanying C++20 project contains an original checked FlexBuffers reader, writer, verifier, constrained mutation functions, a small FlatBuffers table laboratory, and a mutable semantic laboratory. It has no dependency on the official FlatBuffers runtime. Its implementation is readable rather than micro-optimized. The complete core source is also included in the appendices.

The supplied test suite passed 20,220 assertions with GCC 14.2, and with Clang 17 using AddressSanitizer and UndefinedBehaviorSanitizer. It includes 2,000 generated documents, re-encoding checks, 30,000 bounded malformed/random-input probes, width boundaries, map ordering, mutation, and handle lifetime tests. A separate 25,000-run coverage-guided verifier

fuzzing smoke test also completed without a reported sanitizer failure. These are actual executed checks; the receipts are in the companion.

The official C++ or Python runtime was **not executed** against this implementation during preparation of the book. Optional bidirectional C++ and one-way Python oracle programs are included, but neither was executed here. Self-round-trips and source inspection are useful evidence, not a substitute for independent interoperability testing. The code is an educational implementation, not a production-certified replacement.

The optimized capstone is a detailed architecture with executable foundations. The laboratory implements its semantic core using standard containers. Slab allocation, adaptive representations, SIMD hash probing, transactions, and concurrent snapshots are design work described in the book, not features secretly supplied by the laboratory.

How to read

Read Chapters 1–3 even when the vocabulary feels familiar. They establish the arithmetic and lifetime rules that keep the later code honest. Chapters 4–9 explain FlatBuffers. Chapters 10–18 build FlexBuffers from the trailer outward, then reconstruct the builder and verifier. Chapters 19–23 cover implementation quality, hardware, languages, and measurement. Chapters 24–30 develop the mutable capstone and its delivery plan.

At each byte example, pause before reading the explanation. Find the root. Identify the width that reads the current slot. Only then follow a reference and identify the width of the destination. Draw positions, not just values. Most difficult bugs in these formats arise because a correct number is measured from the wrong place.

The exercises are meant to be implemented. Their solutions are in Appendix B. The complete type table, equations, source-reading guide, and bibliography make the book usable as a desk reference after the first reading.

A promise about “maximum performance”

There is no data structure that simultaneously minimizes resident memory, update latency, lookup latency, snapshot cost, and implementation complexity for every workload. The capstone therefore specifies a workload envelope and makes its trade-offs visible. It aims at a strong performance frontier: very cheap scalar updates and typed scans, low-allocation container growth, fast repeated key lookup, stable logical handles, and standard FlexBuffers export.

A claim that it is faster than a particular alternative must come from the benchmark suite, not from the attractiveness of its diagram. The book gives you the machinery to make that claim responsibly.

PART I • BYTES AND OWNERSHIP

1 Two representations, two jobs

Start with an ordinary object

Imagine a record containing a person's name, a counter, and a sequence of measurements. In C++, a straightforward representation might use `std::string`, an integer, and `std::vector<double>`. The record is easy to modify. The string can grow. The vector has capacity. Its implementation can replace one allocation with another and update a pointer.

Now write the raw bytes of that C++ object to a file. Those bytes contain implementation-specific metadata and, commonly, pointers into allocations that were not included in the write. Another process cannot use those pointers. Another compiler may lay out the object differently. Even the same process cannot treat that byte dump as a portable reconstruction of the original ownership relationships.

A wire format replaces those implicit agreements with explicit ones: integer widths, byte order, object boundaries, reference rules, and a way to locate the root. FlatBuffers and FlexBuffers both allow readers to navigate a serialized buffer without first rebuilding a conventional object tree. Their accessors interpret offsets and metadata in place. That is the useful meaning of their direct-access design—not a promise that every interaction with the data costs zero work. [1,2,4]

What “zero-copy” does and does not mean

Consider reading the third measurement. A conventional parser may first allocate a vector and populate it from input bytes. A direct-access reader can calculate the element's position and load the number there. That avoids materializing the vector merely to read it.

But the input still had to arrive somewhere. A network stack may copy it. Validation may scan it. A language binding may allocate a wrapper. Converting its string to an owning `std::string` copies bytes. Decompressing a transport envelope produces a new buffer. “Zero-copy” should therefore always be followed by a boundary: no copy between which two representations, for which operation, under which lifetime conditions?

There are four different costs worth separating. **Acquisition** obtains the bytes. **Verification** establishes that navigation is safe and that required invariants hold. **Access** loads selected data. **Materialization** converts it into another representation. An application that reports only access time while paying the other three on every request has not measured its actual pipeline.

A schema is information, not magic

Suppose a four-byte slot contains `00 00 80 3f`. Is that the integer 1,065,353,216 or the floating-point value 1.0? Bytes alone do not answer. A schema can tell the reader that the slot is a float. Alternatively, a type tag in the buffer can tell it.

FlatBuffers usually obtains this information from a schema and generated accessors. A table's vtable describes the presence and location of fields; it does not generally repeat each field's full semantic type. FlexBuffers stores enough type information in the buffer to navigate its supported value domain without an application-specific schema. Schemaless does not mean typeless. It means the structural typing information travels with the data. [\[1,2,4\]](#)

This difference has a cost model. Removing repeated field names and type information can make known-layout data compact and cheap to access. Carrying names and tags makes unknown-layout data discoverable. Neither choice makes application validation disappear: knowing that a value is an integer does not tell you whether it is a valid account age or an allowed state transition.

Wire compactness and mutation pull in different directions

A compact buffer tends not to reserve unused space. A growable vector does. A compact integer can occupy one byte; an in-memory integer that must accept every future signed 64-bit value is conveniently represented in eight. A wire reference can be a short relative offset; an independently movable object often benefits from a stable handle or a pointer.

These are not accidental implementation weaknesses. They are consequences of different objectives. Every unused byte removed from a wire representation is a byte that is not immediately available for growth. Every native indirection removed can make relocation harder. Every object that is shared to save memory raises a question about what mutation should do to its aliases.

The central design question is therefore not “Can I make FlexBuffers mutable?” Some mutations already fit its existing layout. The stronger question is: “Which mutations must be cheap, and how much compatibility must exist during them?” Chapters 17 and 24 make that question precise.

Three levels of compatibility

Value compatibility means two systems can represent and exchange the same logical values.

Wire compatibility means a produced byte sequence is readable by an ordinary implementation of the format. **Continuous wire compatibility** means the live representation remains a valid ordinary buffer after every published update.

A native map backed by a hash table can have value compatibility and produce wire-compatible snapshots without being continuously wire-compatible. Conversely, an existing FlexBuffers scalar can be changed in place while retaining continuous wire compatibility, provided the new value fits and ownership is controlled.

Keep these levels distinct. A design that silently changes the definition halfway through its argument can appear to achieve impossible combinations of compactness, stable addresses, and arbitrary growth.

Checkpoint

Exercise 1. A service receives a FlexBuffer, copies it into an owned allocation, verifies it once, then reads ten fields. Which costs were avoided? Which costs still occurred? Write a benchmark plan that measures all four stages separately.

2 Bytes, alignment, integers, and floating point

A byte position is not an address

Throughout this book, a buffer position is a non-negative integer measured from the beginning of a particular buffer. An address is a machine pointer. The distinction lets us reason about malformed inputs without first constructing invalid pointers.

For example, a reference at position 100 may contain the backward distance 40. Its target position is 60. After checking that 40 is no larger than 100, we can form the address $\text{base} + 60$. Reversing those steps—first constructing $\text{base} + 100 - 4000$, then asking whether it was valid—is a poor foundation for a safe C++ parser.

Let N be the buffer length, p a position, and n a requested extent. The safe range predicate is:

TEXT

```
 $p \leq N$  and  $n \leq N - p$ 
```

The seemingly equivalent $p + n \leq N$ can fail when the addition overflows. Subtraction after a bounded comparison is not just stylistic caution; it is a way to keep the proof and implementation aligned.

Little-endian storage

Both formats use little-endian scalar storage. The least significant byte appears first. The unsigned integer 300 is hexadecimal $0x012c$, so a two-byte representation is $2c\ 01$. A four-byte representation of the same value is $2c\ 01\ 00\ 00$. The byte order is a wire rule, independent of the host's native endianness. [\[1,4\]](#)

A deliberately simple decoder is:

CPP

```
std::uint64_t u = 0;
for (unsigned j = 0; j < width; ++j) {
    u |= std::uint64_t{bytes[position + j]} << (8 * j);
}
```

Before this loop, validate that the width is one of 1, 2, 4, or 8 and that the entire range exists. Casting each byte to 64 bits before shifting avoids accidental promotion to an insufficient integer type. The largest shift is 56, not 64.

This version is easy to audit. Later we will replace selected paths with fixed-size `memcpy` loads and endian conversion. Do not start by optimizing away the invariant that makes the load legal.

Signed values and sign extension

The signed integer -1 is all one bits at any supported width. At one byte it is `ff`; at two bytes it is `ff ff`. Reading `ff` as unsigned gives 255. Reading it as a signed one-byte two's-complement value gives -1. The decoder therefore needs the width of the stored signed value, not just its raw bits.

For a width smaller than eight bytes, first read an unsigned value `u`. Let $b = 8 * \text{width}$. If bit $b - 1$ is set, the signed value is $u - 2^b$; otherwise it is `u`. The companion uses this expression only where both operands fit its chosen signed intermediate. The eight-byte case is handled separately using `std::bit_cast` between equal-sized integer representations. [12]

This separation avoids two common bugs: shifting a 64-bit value by 64, and attempting to represent 2^{64} in a 64-bit intermediate. Boundary tests must include -128, -129, 127, 128, and the corresponding transitions at 16 and 32 bits. Testing only positive values leaves half the width logic untested.

Alignment is a layout constraint, not a permission slip

A width `w` that is a power of two can be aligned with padding:

TEXT

```
padding(n, w) = (w - (n & (w - 1))) & (w - 1)
aligned(n, w) = n + padding(n, w)
```

If the current length is 5 and the next object needs four-byte alignment, three padding bytes bring the start to position 8. The addition must still be checked for overflow.

There are two distinct alignment questions. Is an object's **position within the format** correctly aligned? And is the **actual address used by a typed C++ load** aligned for that type? A buffer whose internal positions satisfy the format can still be placed at an unusual base address. A bitwise reader can handle that placement; a reinterpret-cast reader may not.

Nor does a suitably aligned address automatically contain a live C++ object of the desired type. Object lifetime and aliasing rules remain relevant. The safe baseline is to read bytes into a real scalar object with `memcpy`, then interpret or endian-convert that scalar. Exact language guarantees matter more than a processor's willingness to execute an unaligned instruction. [12,13]

Floating-point payloads

The practical FlexBuffers C++ paths discussed here use `binary32` and `binary64` floating point. The source contains placeholders for smaller floating types, but these are not an implemented portable `binary16` wire feature to use in a new writer. The teaching implementation accepts floating widths of four or eight bytes only. [1]

A floating-point representation has properties that integer intuition misses. Positive and negative zero compare equal but have different bit patterns. NaN does not compare equal to

itself. A finite double may not survive conversion to float and back without a value change. Infinities are representable in both sizes.

The official width-selection rule for a double tests whether converting to float and back preserves equality. This commonly allows a value such as 1.5 to use four bytes while leaving a value such as a non-exactly-representable double at eight. NaN fails the equality test and therefore does not take the narrow path. This is a storage policy, not a general numeric equivalence policy. [1]

When an application cares about NaN payloads, signaling behavior, or exact storage width, it must define those requirements separately. Avoid fast-math compilation for tests whose contract includes signed zero, NaN, or exact round-tripping. The wire's available types and the application's equivalence relation are different layers.

Checkpoint

Exercise 2. Decode 80 as an unsigned one-byte value and as a signed one-byte value. Decode 80 00 as a signed two-byte value. Explain why copying the first byte and ignoring the stored width gives the wrong answer.

Exercise 3. A buffer has 100 bytes. Show why checking a requested extent with `length <= 100 - position` is preferable to adding position and length first.

3 A safe C++ vocabulary for byte formats

Own the allocation; borrow the view

An owning buffer and a view into it should be different types. The companion uses `std::vector<std::uint8_t>` for owned bytes and `std::span<const std::uint8_t>` for a borrowed range. A Ref contains such a range plus positions and widths. Copying a Ref does not keep the allocation alive.

This is convenient and dangerous in exactly the way an ordinary string view is convenient and dangerous. Returning a reference into a local vector produces a dangling view. Growing the vector can move its storage and invalidate every view into it. Mutating its bytes can invalidate the reasoning established by earlier verification even when the allocation does not move.

A production-facing API should make the ownership boundary harder to cross accidentally. One possible design is an immutable `VerifiedBuffer` that owns its allocation and exposes small borrowing references. Another uses a shared immutable owner when references must outlive the calling scope. The additional ownership cost belongs at the boundary, not necessarily in every scalar accessor.

Four layers of an implementation

The lowest layer performs checked arithmetic and scalar loading. It knows about bytes and widths, but not maps or strings. The next layer implements format navigation: root lookup, indirect targets, length prefixes, and vector indexing. The third layer verifies a reachable object graph and its required semantic invariants. The fourth layer presents application-facing operations and ownership policies.

Keeping these layers distinct pays off during optimization. A verified, immutable buffer may use a faster internal accessor without repeating every structural check. Untrusted input still enters through the verifier. Application range checks remain application checks. A single giant “read anything” function tends to mix these guarantees until none of them can be stated precisely.

The companion chooses a conservative teaching path: accessors retain many bounds checks even after verification. This is not the fastest possible architecture. It makes individual operations useful while you are learning and makes arithmetic defects easier to localize.

A readable scalar load

A production little-endian load specialized for a known width can use an actual scalar object:

CPP

```
std::uint32_t native;  
std::memcpy(&native, bytes.data() + position, sizeof native);  
// Convert native from little-endian if the host is big-endian.  
return native;
```

The range check belongs before the `memcpy`. This is a conceptual excerpt: the complete portable companion uses explicit byte assembly rather than leaving the endian step implicit. Compilers commonly optimize small constant-sized copies, but an implementation should inspect its generated code rather than assuming a particular instruction sequence. [16]

Do not replace the load with an unconditional eight-byte read followed by a mask unless eight readable bytes have actually been guaranteed. A one-byte field at the end of a mapped page is not permission to read seven bytes past the buffer. A padded internal allocation can support a carefully documented fast path; arbitrary caller-owned input cannot inherit that promise.

Error handling is part of the format boundary

The companion throws a small `Error` exception for a violated invariant. That makes examples compact and allows tests to assert rejection. A production library might instead return an error code or an expected-like result containing a byte position, an invariant category, and a path through the document.

The important distinction is between **malformed input**, **unsupported-but-valid input**, **resource-limit rejection**, and **application validation failure**. A nesting limit is not evidence that the bytes are structurally corrupt. A valid unsigned 64-bit integer can be unsupported by a consumer that accepts only signed integers. An empty string can be structurally fine but invalid for a required name.

A useful error object might carry `kind`, `position`, `type`, and `detail`. Avoid allocating a full human-readable path on every successful traversal. Build rich diagnostics on the failure path or in a separate diagnostic mode.

Arithmetic helpers deserve their own tests

The companion's `add`, `mul`, `align_up`, `range`, `load_u`, `load_i`, and `load_f` are short. Their shortness is a reason to test them thoroughly, not a reason to trust them implicitly. Almost every higher-level operation depends on them.

For multiplication, `a == 0 || b <= SIZE_MAX / a` prevents overflow. For a backward reference, verify `distance <= slot` before subtraction. For a length prefix, verify that the prefix fits before the target. For a forward reference, verify `distance <= buffer_size - slot` before addition. These patterns make the permitted arithmetic visible in the code review.

The best tests combine boundary values with malformed layout. A length that fits its integer type can still describe more elements than remain in the buffer. An offset that fits its slot can still point outside the allocation. Width validity and range validity are independent predicates.

Checkpoint

Exercise 4. Design the public lifetime contract for `VerifiedBuffer::root()`. Decide whether the root view may outlive the owner, whether the owner may move, and whether the owner exposes any mutable byte access. Explain the cost of your choices.

PART II • FLATBUFFERS

4 FlatBuffers: the root, the table, and the vtable

The first reference points forward

In the ordinary 32-bit FlatBuffers layout, the buffer begins with an unsigned 32-bit offset to the root object. If the first four bytes encode 12, the root begins at position 12. The offset is measured from the position of the offset itself. For the first word that position happens to be zero; elsewhere it is not. [2,4]

For an ordinary offset slot at position p , the rule is:

TEXT

$$\text{target} = p + \text{U32}(p)$$

The root word does not generally identify the root's schema. The reader is expected to know which root type to use. An optional four-byte file identifier can help recognize an application format, but it is not a complete schema or a security guarantee. A size-prefixed buffer adds framing before the ordinary buffer; the consumer must know which framing convention it receives. [6,8]

A table begins with a different kind of offset

A table's first word is a signed displacement to a vtable. Its rule is subtraction:

TEXT

$$\text{vtable} = \text{table_position} - \text{S32}(\text{table_position})$$

This is deliberately not the rule for an ordinary forward offset. The signedness matters because a vtable may be located before or after a table, particularly when layouts are shared. A positive displacement points backward. A negative displacement makes the subtraction move forward. [2,3]

A FlatBuffers vtable is a small metadata array, not a C++ virtual-function table. Its first 16-bit value is the vtable's byte length. Its second is the table's inline byte size. Subsequent 16-bit values are field displacements measured from the beginning of the table. A zero displacement means the field is absent. The table's inline byte size does not include all the out-of-line strings, vectors, or child tables reachable from it. [2,4]

Finding a field slowly

Suppose a schema assigns field ID 2 to a signed integer. Its vtable entry is at byte offset $4 + 2 * 2 = 8$ within the vtable. A generated accessor performs the conceptual steps below:

```
TEXT
T = table position
V = T - signed32(T)
L = unsigned16(V)
E = 4 + 2 * field_id

if E is outside the vtable:
    return schema default
D = unsigned16(V + E)
if D == 0:
    return schema default
return signed32(T + D)
```

The real implementation must validate the vtable and field ranges before trusting them. Once verification has established the necessary invariants, a fast accessor can perform just the small number of loads needed for the requested field. The pinned `Table::GetOptionalFieldOffset` explicitly handles a field beyond the stored vtable as absent. [2]

This is how a newer reader can ask an older buffer for a field that was not present when the buffer was written. The absence behavior is not a vague compatibility feature; it comes directly from the vtable-length check and the schema's default.

Why not store fields in schema order?

A fixed field order would make some offsets predictable, but it would make omission and evolution less flexible. FlatBuffers lets the writer choose physical field placement while the vtable tells the reader where each present field ended up. Fields with different alignment requirements can be arranged efficiently. Fields equal to their default can be omitted. Tables with identical layouts can share a vtable. [3,4]

The cost is an extra dependency when locating a field: read the table's vtable displacement, then the relevant field displacement, then the field itself. If you repeatedly access many fields in a hot loop, the compiler and cache may make this cheap, but the logical dependency remains. A struct trades away this flexibility to remove the per-object vtable.

A vtable also makes bitwise equality a poor default definition of value equality. Two valid writers may choose different field order, omit different explicitly written defaults, use different sharing, or place padding differently while representing equivalent values. Canonical bytes require an additional deterministic encoding policy.

Why 16-bit field displacements?

A table's inline body is normally small: scalar fields and references, not the complete nested document. Sixteen-bit vtable entries keep its metadata compact. Large child vectors remain out of line and are reached through their own offsets. The pinned builder checks that the inline table body fits the representable vtable displacement range. [3]

This is a useful example of a local limit that should not be confused with a global one. The limit on an inline table body, the width of an ordinary reference, and a particular implementation's maximum total buffer size are three separate constraints.

Checkpoint

Exercise 5. A table starts at position 64 and its first signed word is -16. Where is its vtable? If field ID 3 has displacement 12, where is that field? Which of those two calculations uses the table position as its base?

5 A complete 32-byte FlatBuffer

The schema and the values

We will encode this deliberately small schema:

```
TEXT
table Sample {
  count:int = 0;
  name:string;
}
root_type Sample;
```

The logical value is `count = 13`, `name = "Ada"`. The companion's `flat_table_lab.hpp` constructs a valid planned layout for this one schema. It is not the official backward-growing builder, and it is not a schema compiler. Its purpose is to expose every byte in a complete table example. The governing layout rules come from the FlatBuffers table and offset representation. [\[2,4\]](#)

The complete buffer is:

```
TEXT
00: 0c 00 00 00 08 00 0c 00
08: 04 00 08 00 08 00 00 00
10: 0d 00 00 00 04 00 00 00
18: 03 00 00 00 41 64 61 00
```

The left column is a hexadecimal byte position. Thus 10 means decimal 16, not decimal 10. Every other number in the walkthrough below is decimal unless explicitly prefixed with 0x.

Position	Size	Meaning	Decoded value
0	4	Root offset, measured from position 0	12
4	2	Vtable byte length	8
6	2	Table inline byte size	12
8	2	Field 0 displacement	4
10	2	Field 1 displacement	8
12	4	Signed vtable displacement	8
16	4	count	13
20	4	Forward reference to the string	4
24	4	String byte length	3

Position	Size	Meaning	Decoded value
28	4	A, d, a, terminating zero	—

Find the count

Read the unsigned word at position 0: 12. The root table is therefore at position 12. Read the signed word there: 8. The vtable is at $12 - 8 = 4$.

Field 0 uses the vtable entry at $4 + 4 + 2 * 0 = 8$. Its value is 4. The field itself is at $12 + 4 = 16$. Reading a signed 32-bit integer at position 16 gives 13.

Notice the two different bases. The vtable entry's location is measured from the vtable. The entry's contents are measured from the table. Confusing these bases is a classic way to write an accessor that passes one carefully aligned example and fails on ordinary data.

Find the name

Field 1 uses the entry at $4 + 4 + 2 * 1 = 10$. Its displacement is 8. The field's offset slot is therefore at $12 + 8 = 20$.

The word at position 20 is 4. This is an ordinary forward reference, so the string object begins at $20 + 4 = 24$. The length at position 24 is 3. The three content bytes begin at position 28, followed by a zero terminator.

A FlatBuffers string reference points to its length field. Later, a FlexBuffers string reference will point to the content bytes, with its length immediately before them. Remember that contrast now; it will prevent a deceptively plausible four-byte error later. [\[1,4\]](#)

Make the field absent without moving anything

In the laboratory test, replacing the field-0 vtable entry with zero makes `count()` return its schema default, 0. The old bytes holding 13 may still be present in the allocation. They are no longer the value of the field because the metadata no longer identifies them as such.

This illustrates an important separation between physical bytes and reachable semantic data. A buffer can contain padding or unreachable bytes. A decoder follows the format's references, not every byte that happens to resemble a value. Conversely, leaving a secret in unreachable storage is not the same as erasing it before publishing the allocation.

Changing a vtable can affect multiple tables if that vtable is shared. The demonstration modifies a purpose-built unshared example. It is not a recommendation to patch arbitrary production vtables in place.

A negative displacement is valid

The test suite also places the table at position 4 and its vtable at position 16. The table's signed displacement is then -12. Its bytes are `f4 ff ff ff`. The reader computes $4 - (-12) = 16$ and finds the vtable correctly.

A reader that treats that word as an unsigned backward distance will fail. A verifier that assumes every vtable must precede its table will reject a representation the signed-displacement design permits. The pinned reader uses a signed scalar for exactly this calculation. [2]

From one table to a real reader

The laboratory accessor checks the ranges needed for this schema and exposes only `count()` and `name()`. A general generated reader additionally needs accessors for all declared fields and compound types. A schema-aware verifier must validate those types, required fields, nested objects, and resource limits.

Do not expand this example by simply casting the whole buffer to a C++ struct. Expand it by preserving the explicit navigation rules and generating the schema-specific parts. The smallness of the example is meant to isolate the mechanism, not to bypass it.

Checkpoint

Exercise 6. In the example, what must change to replace "Ada" with the same-length string "Lin"? What additional problems appear when replacing it with "A longer name"? Consider both storage space and references.

6 Structs, vectors, strings, unions, and framing

Structs exchange flexibility for predictable placement

A FlatBuffers struct is an inline, fixed-layout aggregate. Its fields are present according to its schema-defined layout, with alignment and padding determined by the format's struct rules. It does not carry a per-instance vtable. This makes it suitable for small, stable records such as coordinates or numeric samples. A table can contain a struct inline; a vector of structs stores their fixed-size records consecutively. [4,6]

For a conceptual struct containing a 32-bit float followed by a 16-bit integer, the fields begin at offsets 0 and 4. With four-byte aggregate alignment, the total stride is rounded to 8, leaving tail padding. That tail padding matters in a vector: the next record begins at offset 8, not 6. The actual generated definition is the authority for a particular schema.

Adding a field changes the stride and often the alignment. That is why a struct is not the same evolution mechanism as a table. A reader that assumes the old stride cannot simply discover the new field from a vtable, because there is no vtable to consult.

Vectors are contiguous, but their elements may be references

An ordinary FlatBuffers vector begins with a count and then its elements. Numeric elements are stored consecutively at their schema-defined width. A vector of structs has a fixed struct stride. A vector of tables or strings contains offsets, one per element, with each offset measured from its own slot. [4,6]

Suppose two string-reference slots begin at positions 40 and 44. If their words are 20 and 24, their targets are 60 and 68 respectively. The second target is not $40 + 24$. Every relative reference has its own origin.

This arrangement makes the vector itself compact and indexable while allowing variable-size children. It also distinguishes an array scan from a pointer chase. Scanning a million inline floats can be a regular memory stream. Scanning a million references to scattered strings includes the cost of resolving and visiting those strings.

FlatBuffers fixed-length arrays are associated with struct layouts, rather than being a drop-in evolvable table-field mechanism. Multidimensional or nested collection needs are commonly represented through an explicit table or struct layer according to the schema's supported constructs. Check the chosen language generator's support before relying on less common schema features. [6,8]

Strings have both a length and a terminator

The length permits bounded access and allows embedded zero bytes to be distinguished from the terminator. The terminator supports C-string-style interoperability where appropriate. A consumer that calls `strlen` discards the advantage of the stored length and will stop at an embedded zero. A consumer that constructs an owning string may allocate even though the wire access itself was direct. [3,4]

The byte count is not a count of Unicode characters or grapheme clusters. The schema's string convention and an application's text-validation policy should be treated explicitly. A binary payload that is not intended to be text is better modeled as a byte vector than silently treated as a user-visible string.

A union is a discriminator plus a payload

A union field needs information about which alternative is present. In the normal generated model, a discriminator accompanies the payload reference. The payload is interpreted according to that discriminator. A discriminator for one table paired with a reference to another table is not made correct merely because both references point inside the buffer. A verifier must validate the selected alternative. [6,8]

This is a reminder that bounds safety is not complete type safety. A range can exist and still be interpreted incorrectly. Schema evolution involving unions requires careful handling of new alternatives and the behavior of older readers. Union-vector support and more advanced alternatives should be checked for every target language, rather than inferred from one C++ API.

Size prefixes and identifiers solve different problems

A file identifier helps identify an application format. A size prefix helps determine a message's extent in a stream or container. Neither replaces structural verification. A four-byte identifier is easy for an attacker to copy. A declared length can be wrong. Framing code must ensure the complete declared range is available before constructing a format view. [6,8]

This distinction matters when comparing FlatBuffers and FlexBuffers roots. Ordinary FlatBuffers has a front root offset. FlexBuffers discovers its root from the end of a known buffer extent. A FlexBuffers receiver therefore needs an external boundary: a transport frame, a file size, or a containing field. Its trailer is not a universal stream delimiter.

Nested buffers are opaque until validated

A byte vector can contain another serialized format. FlatBuffers schemas and tooling support conventions for nested FlatBuffers and FlexBuffers payloads. The outer verifier establishes that the byte vector is in range; it does not automatically establish every semantic property your application needs from the inner format. Treat the inner buffer's parser and lifetime as a second boundary. [6,8]

Embedding is often cleaner than forcing one schema to encode every dynamic value. A mostly stable record can use FlatBuffers fields for known data and a FlexBuffers blob for a genuinely dynamic extension. The cost is an additional validation and access path. That cost is explicit and often preferable to turning the whole record into a generic map.

Checkpoint

Exercise 7. Design a schema for a stable header, a million 3D points, and an optional dynamic metadata map. Explain which parts should be structs, tables, vectors, and an embedded FlexBuffer, and identify every validation boundary.

7 Reconstructing the FlatBuffers builder

Why build backward?

In the usual FlatBuffers builder, children are constructed before parents, while the allocation grows toward lower addresses. When the final buffer is viewed from beginning to end, parents can point forward to children that were already built. This avoids leaving ordinary child references unresolved. [3,4]

The builder does not need to know the final base address to retain a child location. It records a construction offset measured from the end of the active region. If the backing allocation grows and existing bytes move together, those end-relative positions remain meaningful. Raw pointers into the old allocation would not.

This is a construction technique, not the definition of every valid FlatBuffer. Chapter 5 used a forward planned layout because it knew all positions in advance. Both approaches can produce the same kind of wire references.

Derive the offset conversion

Let S be the current used size after alignment. Let D be the child's construction offset from the end. We are about to prepend an ordinary four-byte offset slot. After that prepend, the slot is $S + 4$ bytes from the end. The child is D bytes from the end. The forward distance from the new slot to the child is therefore:

TEXT

```
relative_distance = S + 4 - D
```

For example, if the current used size is 20 and the child is 12 bytes from the end, the new slot stores $20 + 4 - 12 = 12$. Its target is twelve bytes forward from its final position. The pinned builder's `ReferTo` implements this relationship, with separate handling for its 32-bit and 64-bit regions. [3]

Alignment must be completed before calculating S . Otherwise the formula uses a position that is no longer where the slot will be written. This is the same general problem we will encounter in FlexBuffers width selection: layout decisions and offset arithmetic cannot be separated casually.

Building a table

A table construction scope records the positions of fields that are actually emitted. A scalar equal to its default may be omitted unless the builder is configured to force defaults. An offset field whose child is absent can be omitted. Inline structs are copied into the table body with their required alignment. [3]

When the table ends, the builder prepends a placeholder signed vtable displacement. That position becomes the beginning of the table. It can now compute each recorded field's displacement from the table beginning. It emits the vtable's byte length, the inline table byte size, and the field displacement entries, then fills in the table's signed displacement to the selected vtable.

Conceptually:

TEXT

```
begin table
  remember start size
  write selected fields; record (field_id, construction_position)
end table
  prepend signed displacement placeholder
  compute table start and inline size
  build vtable entries from recorded positions
  reuse an identical vtable or emit a new one
  patch the signed displacement
```

The builder's temporary field-location data is not serialized as application data. It is construction state. Confusing scratch metadata with wire metadata is another common mistake when reading a compact builder implementation.

Why constructing children inside the parent scope is discouraged

A child string or vector is out-of-line relative to its parent table. Constructing such objects in the middle of the parent's inline construction can inflate the apparent table body and complicate field displacements and vtable reuse. The pinned builder's nesting checks explicitly guide callers to construct children before opening the parent table scope. [3]

Generated convenience functions make this discipline less intrusive. They accept already-created child offsets or create appropriately ordered code around the builder. A handwritten replacement should enforce a clear construction state machine rather than relying on users to infer when nested operations happen to work.

Vtable deduplication

Two tables can have the same presence pattern, inline size, and field displacements. They can share one vtable even when their actual field values differ. The pinned implementation compares candidate vtable bytes against previously remembered layouts and reuses an equal one. [3]

The shared object is the layout description, not the table's value data. This is a small form of structure sharing. It saves bytes at the cost of builder bookkeeping and comparison. It also explains why signed vtable displacements are necessary: a reused vtable's position need not be on the same side of every table.

The comparison strategy is an implementation choice. A different builder could index vtables by a hash, use a size bucket first, or decline to deduplicate for workloads where the extra work is not worthwhile. It must still emit valid displacement arithmetic.

Finishing and ownership

Finishing the buffer writes the root information and any requested identifier or size framing, with the alignment needed by the contained objects. The final byte range is then exposed to the caller. The builder's storage ownership must remain clear: a borrowed pointer into the builder is invalid if the builder is destroyed or reuses its allocation. [3,8]

The natural optimization is to transfer ownership of the finished allocation rather than copy it unnecessarily. That is an API and allocator design issue, not a change to the wire format. Be careful not to market a no-copy ownership transfer as evidence that earlier construction allocated nothing.

Checkpoint

Exercise 8. Derive the `ReferTo` formula for an eight-byte offset slot. Then explain why a construction offset must not be confused with the offset eventually stored in the wire buffer.

8 Schema evolution, generated code, and reflection

Evolution works because absence is meaningful

A table field has an identity that survives changes in physical layout. A reader looks up that field's vtable entry; absent entries yield the schema's defined absence or default behavior. This permits a newer table schema to add fields while retaining access to older buffers. The supported evolution rules still need to be followed. [2,7]

Appending fields is the straightforward pattern. Explicit field IDs can make identity more visible, but they introduce their own consistency rules and should be checked by the schema compiler. Reusing a field's identity for a different meaning defeats the mechanism. A field that once meant meters does not safely become milliseconds merely because both are stored as integers.

Deprecation is different from deletion and reuse. Keeping the historical identity reserved prevents old data from being interpreted as a new field. The wire format solves location compatibility; the schema author remains responsible for semantic compatibility.

Defaults are part of the stored meaning

A writer can omit a scalar whose value equals its schema default. An older buffer containing no physical slot for that field is therefore interpreted through the reader's default. Changing the default can change the meaning of existing bytes without changing those bytes. This is why defaults belong in evolution review, not just API ergonomics. [3,7]

Optional scalar support distinguishes absence from an ordinary numeric default where the schema and target language support it. Applications should decide whether they need “not provided,” “provided as zero,” and “provided as a nonzero value” to be distinct. A serialization format cannot recover a distinction that the writer intentionally erased.

Required fields make another trade-off. They can express an invariant, but introducing a newly required field into a population of older buffers can cause rejection. Compatibility plans must include the verifier, not just accessor behavior.

What generated code actually supplies

A schema compiler parses declarations, resolves types and identifiers, validates rules, assigns layout information, and emits language-specific accessors and builder helpers. The runtime supplies common primitives such as scalar loads, vector views, table field lookup, and verification support. [8,9]

The generated accessor for a field often contains very little logic because the schema has already supplied its type, default, and vtable entry. Reimplementing “FlatBuffers in C++” can

therefore mean several different projects: a runtime for generated code, a schema compiler, a compatible generator for a subset, or a handwritten reader for one schema. State which project you are undertaking.

The companion's table laboratory is the last of these. The FlexBuffers companion is broader because FlexBuffers carries its supported structural types in the buffer and does not require an application schema compiler.

Reflection moves schema information to runtime

A schema-less-looking FlatBuffers tool still needs schema information from somewhere. Reflection can supply a binary schema describing tables, fields, and types. Runtime code can then navigate a table using that schema rather than a statically generated accessor. The C++ documentation describes reflection and object-oriented access paths with different trade-offs from the direct generated API. [8]

Reflection is not evidence that an arbitrary FlatBuffer is self-describing by itself. A raw table's vtable gives locations, not enough information to infer whether a four-byte field is an integer, a float, or an offset. FlexBuffers makes a different trade-off by including its type information with each relevant value or container.

An object API can materialize a conventional owning object graph, making complex edits more convenient. Rebuilding a serialized buffer afterward is often a sensible engineering choice. It gives up some direct-access advantages during editing, but it avoids pretending that a compact immutable layout is a general-purpose mutable heap.

Test evolution in both directions

A useful compatibility matrix includes old writer to new reader and new writer to old reader. It should test missing fields, changed presence patterns, default values, deprecated fields, union alternatives, and verification outcomes. Parsing successfully is not the same as preserving meaning.

Keep byte fixtures produced by historical versions. A schema change that passes tests generated only from the new schema may still reinterpret old data. Test the actual historical bytes, especially when defaults, field IDs, or requiredness are involved.

For reproducible artifacts, pin the schema, generator version, runtime version, and relevant builder settings. Two valid implementations need not produce byte-identical output. Use semantic comparisons for interoperability and byte comparisons only where the encoding policy is deliberately fixed.

Checkpoint

Exercise 9. A field `retries:int = 3` was usually omitted by old writers. A new schema changes the default to 5. What do new readers see in old buffers? Propose an evolution strategy that preserves historical meaning.

9 FlatBuffers verification and the boundary of mutation

Verify before trusting navigation

A generated accessor can be extremely small because it assumes the buffer already satisfies the conditions required for its loads. A verifier checks those conditions using the schema: root range, vtable structure, field extents and alignments, referenced objects, vector counts, string boundaries, union alternatives, and resource limits. The exact checks and options depend on the runtime version. [2,8]

The critical arithmetic is familiar. A vector count must not overflow when multiplied by element size. A table field must fit within the permitted buffer region. A signed vtable displacement must be resolved without forming an invalid pointer first. A recursive structure needs depth and work limits even when individual offsets are in bounds.

Verification should be connected to ownership. If another thread or a caller can mutate the bytes after verification, the proof no longer necessarily applies to later access. A verified immutable snapshot is a much simpler contract than a writable shared allocation whose metadata can change at any time.

Mutation of an existing scalar

Suppose a present field contains a 32-bit integer. Replacing its value with another 32-bit integer does not change any field location or child extent. The existing slot is sufficient. A generated mutable accessor or an appropriate runtime operation can perform that update, provided the underlying allocation is actually writable and no conflicting access occurs. [2,8]

An absent scalar is different. There may be no slot to write. The pinned `Table::SetField` overload that also receives the default can report success for an absent field when the requested value equals that default; it has not allocated a new field. The overload without that default check fails when the slot is absent. Read return values according to their contract rather than assuming “true” always means bytes changed. [2]

Length-preserving edits are a narrow case

A present string's content can sometimes be replaced within existing storage. A vector element can be updated when its representation does not require a different layout. But inserting fields, increasing vector length, or growing a child object generally needs storage and metadata changes beyond a simple scalar write.

Patching an offset is not by itself a general allocator. The new destination must exist, be correctly aligned and encoded, lie in the reference's permitted direction and range, and remain alive as part of the buffer. Other aliases and verifiers must still observe a coherent

object graph. An official mutable accessor should not be mistaken for an arbitrary editing engine.

The most robust editing strategy for infrequent structural changes is often materialize, edit, rebuild. The capstone later develops a native representation for workloads where that repeated rebuilding becomes expensive.

The modern 64-bit nuance

The ordinary layout taught in Chapters 4–7 uses 32-bit offsets and counts and 16-bit vtable entries. The pinned C++ source also contains 64-bit offset accessors and builder machinery that distinguishes 32-bit and 64-bit regions. It would be incorrect to claim that all modern FlatBuffers facilities are permanently restricted to a single uniform 32-bit reference representation. [2,3]

It would be equally incorrect to infer universal large-buffer support across every language and tool from those C++ symbols. A production design must pin the exact schema features, builder type, verifier, size limits, and consumer implementations it intends to use. Practical implementation limits may be stricter than the unsigned width of an individual field.

The companion deliberately implements the classic small-buffer case only. Its examples are intended to teach the mechanisms, not to silently advertise support for the 64-bit extensions.

When FlatBuffers is the right execution representation

Known schemas, mostly immutable records, sparse field access, memory-mapped data, and compact transport are strong reasons to preserve the serialized representation. Repeated mutation, dynamic field insertion, unknown keys, and heavy aggregate computation can favor a native layer or a different layout.

The correct decision can differ within one application. A received message can remain a verified FlatBuffer. A mutable working set can use native arrays and maps. A completed result can be serialized again. Trying to force every phase into one representation is an architectural choice with a measurable cost, not an automatic simplification.

Checkpoint

Exercise 10. A table's integer field is absent because its value equaled the default. Why is reserving extra bytes at the end of the whole buffer insufficient to make a normal scalar setter create that field? Identify the metadata that would also have to change.

PART III · FLEXBUFFERS

10 FlexBuffers starts at the end

A three-byte document

The smallest useful way to understand FlexBuffers is to decode a whole document containing the signed integer 13:

TEXT

```
position: 0  1  2
byte:     0d 04 01
meaning:  13 tag width
```

The last byte is the **actual byte width of the root slot**. Here it is 1. The preceding byte is the packed type. The root slot occupies one byte immediately before those two trailer bytes. There is no separate front root pointer in this example. [\[1,4\]](#)

For a buffer of length N:

TEXT

```
root_parent_width = bytes[N - 1]
root_packed_type  = bytes[N - 2]
root_slot         = N - 2 - root_parent_width
```

Validate the length and width before performing the final subtraction. An input shorter than three bytes cannot contain even the smallest root. A last byte of 3 is not a three-byte width; it is invalid as a root width.

Split the packed type

A packed type stores a six-bit type code in the high bits and a two-bit width code in the low bits:

TEXT

```
type_code  = packed >> 2
width_code = packed & 3
child_width = 1 << width_code
```

Thus 0x04 is type 1 with width code 0. Type 1 is a signed integer, and width code 0 corresponds to one byte. The last byte of the document was also 1, but these are different encodings in different places: the tag stores the logarithmic width code; the trailer stores the actual width. [\[1\]](#)

For a true boolean, the complete buffer is 01 68 01. The packed type 0x68 is decimal 104, and $104 \gg 2$ is 26, the boolean type. Boolean is an inline type even though its numeric type code is

well above the indirect types. Do not decide inline versus indirect with an oversimplified “type code below some threshold” rule. [1]

Inline values and indirect values

An inline root stores its scalar payload in the root slot. An indirect root stores an unsigned backward distance there. For an indirect slot at position P , the destination is:

TEXT

```
target = P - unsigned_value_at(P, parent_width)
```

The distance is measured from the offset slot itself. It is not measured from the beginning of the buffer, the trailer, or the beginning of the enclosing vector. The source must be checked before subtraction so a malicious distance cannot underflow the position.

The normal builder writes referenced children before their parents and appends the root at the end. Backward references fit that construction order. Like the backward-growing FlatBuffers builder, this avoids unresolved ordinary child references, but the two formats choose opposite wire directions. [1,3,4]

Why the trailer contains two widths in disguise

The root's final byte tells us how many bytes to read for the root's own slot. The packed tag's low bits tell us the width relevant to the object being described. When the root references a vector, that is the vector's element width. When it references a string, that is the string's length-prefix width.

These widths can differ. A large vector may have eight-byte elements but lie only a short distance before the root slot, allowing a one-byte root offset. A tiny string may use a one-byte length prefix but be referenced from an eight-byte slot in a wide parent vector.

A reliable internal reference object therefore carries at least the buffer, the slot position, the parent slot width, the child width, and the type. Collapsing the two widths into one field is the source of many plausible but incorrect reimplementations. The official Reference stores them separately. [1]

A string root

The complete buffer for "Ada" produced by the companion is:

TEXT

```
03 41 64 61 00 04 14 01
```

The final width is 1. The packed type is 0x14: type 5, string, with a one-byte child width. The root slot is at position 5 and contains 4. Its target is $5 - 4 = 1$, the A byte. The one-byte length prefix is immediately before the target, at position 0, and contains 3. The zero terminator follows the three content bytes.

This example is eight bytes long. No step required an application schema. The tag, the trailer, the relative reference, and the prefix provided the structural information. The reader still needs to decide whether it accepts a string as its application root and whether "Ada" is a meaningful value for that application.

What the trailer does not provide

The trailer does not tell a stream receiver where the document ends until the receiver already knows which bytes belong to it. An external frame or containing object must supply the extent. Nor does the trailer contain a checksum, authentication tag, schema version, or application type identifier. Those belong in an envelope or a deliberate root convention when required.

A root map with a version field can be an application convention, but it is not a built-in FlexBuffers protocol version field. Keep transport framing, application versioning, and the core byte format separate.

Checkpoint

Exercise 11. Decode 01 68 01 without consulting an API. Then explain why a packed width code of 3 is valid while a final root-width byte of 3 is invalid.

11 Widths are a small layout solver

The four storage widths

FlexBuffers uses widths of 1, 2, 4, or 8 bytes for scalar slots, offsets, and relevant length fields. These are fixed-width values once the layout has been chosen; they are not variable-length integers decoded byte by byte until a continuation bit ends. Width metadata tells the reader how much to load. [1]

For an unsigned integer, choose the smallest width whose range contains the value. For a signed integer, choose the smallest width whose signed range contains it. The transitions differ: unsigned 255 fits in one byte, but signed 255 does not. Signed -128 fits; signed -129 does not.

Width	Unsigned maximum	Signed range
1	255	-128 through 127
2	65,535	-32,768 through 32,767
4	4,294,967,295	-2,147,483,648 through 2,147,483,647
8	$2^{64} - 1$	-2^{63} through $2^{63} - 1$

The official signed-width helper uses bit operations resembling a signed-to-unsigned range transformation. That does not mean signed values are stored with zigzag encoding. The payload remains a width-selected two's-complement integer. [1]

A parent gives all its slots one stride

A heterogeneous vector uses a common element width. It may contain a one-byte-sized integer, a string reference, and a double, but the slots themselves all occupy the chosen vector width. The values have different meanings and type bytes, not different slot strides.

This makes index calculation simple: $V + i * W$. It also means a single wide inline value can widen every slot. The writer has to choose W large enough for the count, every inline scalar, and every relative offset stored in those slots. The referenced children's own widths are separate. [1]

The count participates because a variable-length vector stores its count in the same width as its elements. A typed vector of 256 tiny integers therefore cannot use one-byte slots: its count does not fit. This can surprise someone who considers only the element values.

A full example with different widths

Here is the companion's encoding of the untyped vector ["hi", 300]:

TEXT

02 68 69 00 02 00 05 00 2c 01 14 05 06 29 01

Position	Bytes	Interpretation
0	02	String length, width 1
1–3	68 69 00	hi and terminator
4–5	02 00	Vector count 2, width 2
6–7	05 00	String offset: target 6 - 5 = 1
8–9	2c 01	Signed integer 300
10	14	String tag; child width 1
11	05	Integer tag; stored inline width 2
12	06	Root offset: target 12 - 6 = 6
13	29	Vector tag; child width 2
14	01	Root slot width 1

Read that table from the bottom first. The root offset occupies one byte, yet its destination vector has two-byte elements. Then read the string element: its offset occupies two bytes, yet the destination string has a one-byte length prefix. Both differences occur in one fifteen-byte document.

Why distance depends on the candidate width

Suppose a child is already at position 0, the current buffer length is 254, and a two-element vector will be written next. With one-byte elements, its count is at 254 and its element slots are at 255 and 256. An offset from the first slot back to position 0 fits in one byte; the offset from the second does not.

Trying two-byte elements changes the layout. Alignment rounds 254 to 256 because it is already even. The two-byte count occupies positions 256–257. The element slots are now at 258 and 259. Their distances are larger, but two bytes can hold them.

It is incorrect to compute distances once using one-byte slots, discover that some need two bytes, and then write the wider layout without recomputing positions. The earlier slots and padding have moved the later slots. Width choice is a small constraint-solving problem, not just a maximum over already known distances.

A simple global candidate algorithm

The companion deliberately tries the four widths in ascending order. For each candidate it calculates the aligned prefix position, the actual position of every future slot, and whether every value fits. The first candidate that satisfies all constraints is selected.

TEXT

```
for W in {1, 2, 4, 8}:
    B = align(current_size, W)
    check count fits W, when a count exists
    check map prefix references, when present
    for element i:
        P = B + (prefix_slots + i) * W
        if element is inline:
            check its required width <= W
        else:
            check target <= P
            check P - target fits W
    if all checks pass:
        choose W
```

The official builder factors the computation into per-value width helpers and vector construction. The global trial formulation is easier to explain and independently implement. Both must honor the final slot positions, alignment, and parent/child width distinction. Exact output bytes can differ while remaining valid. [\[1\]](#)

Floats and indirect numeric values

An inline float has to be written at the parent's slot width, which must be four or eight bytes. A numeric value can also be stored out of line using an indirect numeric type. Its payload then has its own width, and the parent stores only an offset. This can prevent one large value from widening an otherwise narrow heterogeneous vector.

Do not assume indirection always helps. The out-of-line scalar requires payload bytes and alignment. Its offset can itself require a wider parent slot if the distance is large. Chapter 17 calculates a concrete case and explains how to decide rather than applying indirection everywhere.

Checkpoint

Exercise 12. In `["hi", 300]`, identify the width used for each of these operations: reading the root offset, reading the vector count, reading the string offset, and reading the string length.

Exercise 13. Explain why a vector of 256 booleans may occupy substantially more than 256 payload bytes in the wire representation even though each logical value needs only one bit.

12 Keys, strings, and blobs are different objects

A key has no length prefix

A key is a zero-terminated sequence used for map keys and the standalone key type. The offset points at its first content byte. Its end is found by scanning for the terminator. There is no independent stored byte count before that target. Map lookup in the pinned C++ reader compares keys with C-string-style lexical comparison. [1]

An embedded zero would end the key, so an application-facing key constructor should reject it rather than silently creating two different interpretations of the same supplied string. The companion also requires valid UTF-8 for newly written keys. Its byte ordering is lexical ordering of the encoded bytes, not locale-aware collation or Unicode normalization.

The keys "A" and "a" are different. Canonically equivalent Unicode strings with different byte sequences are different unless the application normalizes them before insertion. The format does not make those policy decisions for you.

A string has a length and a terminator

A FlexBuffers string is laid out as:

```
TEXT
[length in W bytes][content bytes][zero]
                    ^ target
```

The reference's child width tells the reader how wide the length prefix is. The prefix is immediately before the target. The offset slot that reaches this string may use a different width. The length counts content bytes and excludes the terminator. [1]

Because the length exists, a string can contain embedded zero bytes while retaining its full length. A reader using the length can return "a\0b" as three bytes; one using `strlen` sees only "a". This distinction is especially important when importing legacy typed string vectors, where the necessary per-string prefix-width information was lost.

The string prefix widens at length boundaries, but the content remains a byte sequence. A 256-byte string has a two-byte length prefix; it does not store every character in two bytes. The vector count/stride coupling discussed in Chapter 11 does not apply to string content.

A blob is not a typed byte vector

A blob uses a length prefix followed by raw bytes, with no required string terminator. Its content is not interpreted as UTF-8. The child width describes the length prefix, not a stride larger than one for the payload. [1]

This makes a blob the natural representation for arbitrary binary data. A typed unsigned vector can instead communicate a sequence of unsigned numeric values, but its element width must also hold the vector length. Using a blob for a large byte payload avoids widening every byte merely to represent a larger count.

The distinction survives even if both happen to contain the same byte values. A blob is one binary object. A typed vector exposes numeric elements. A semantic compatibility layer should not silently convert between them unless its contract explicitly treats those forms as equivalent.

Empty values reveal useful edge cases

The companion emits these complete documents:

Logical value	Complete bytes	Size
Empty string	00 00 01 14 01	5
Empty blob	00 00 64 01	4
Empty untyped vector	00 00 28 01	4

For the empty blob, the first byte is its zero length. Its content target is position 1, which is also where the root offset slot is written. The backward distance is therefore zero. This is valid because the blob has no content bytes to consume. A blanket rule that every indirect offset must be strictly positive would reject this ordinary empty representation.

A zero offset is not a blanket permission for arbitrary self-reference either. The target type, prefix, extent, and traversal rules still matter. A verifier must judge the actual object, not replace all that reasoning with one overbroad distance rule.

Pooling changes aliasing

A builder can reuse the same serialized key or string for multiple references. The pinned C++ builder pools keys by default and offers optional string pooling. It can also reuse previously built values explicitly. These choices reduce duplicate bytes at the cost of construction bookkeeping. [1]

Now suppose two fields reference one pooled string. A same-length in-place edit changes the shared payload. Both fields observe the new text. A caller expecting value-copy semantics would be surprised, even though the mutation preserved the wire layout perfectly.

Pooling therefore belongs in the ownership and mutation story. “Immutable strings are shared” and “mutable strings behave like independently owned values” are compatible only with a rule such as copy-on-write or replacement of references. They are not compatible merely because the API uses the word string.

Boundaries and text validation

A safe reader checks the length prefix before reading it, the content extent before exposing it, and the terminator according to the contract it promises. UTF-8 validation is an additional scan. A structural verifier's success should not be assumed to imply Unicode validity or application-specific text rules.

For very large payloads, repeated validation can dominate a workload. A verified immutable owner can cache the result for a particular range and interpretation. A writable buffer cannot retain that conclusion after arbitrary mutation. Cache invalidation here is a correctness issue, not just a performance detail.

Checkpoint

Exercise 14. Why does a 300-byte blob avoid the width inflation of a 300-element typed byte vector? Draw the two layouts and identify which quantities use the container width.

13 Every vector family, one at a time

Untyped vectors carry a separate type table

An untyped vector has a count before its data, uniformly sized slots, and one packed type byte per element after the slot array:

TEXT

```
[count: W][slot 0: W] ... [slot N-1: W][tag 0] ... [tag N-1]
      ^ V
```

Its element slot is at $V + i * W$. Its type byte is at $V + N * W + i$. The type table is byte-dense; its entries are not spaced W bytes apart. This is one of the most important indexing formulas in a reimplement. [1]

The complete untyped vector [1, 2, 3] is:

TEXT

```
03 01 02 03 04 04 04 06 28 01
```

The count is at position 0. Slots begin at 1. Their tags begin at 4. The root slot is at 7, stores distance 6, and therefore points to position 1. The root tag is 0x28, an untyped vector with one-byte elements. The total is ten bytes.

Typed vectors remove the per-element tags

If all elements have one supported type, the vector's own type identifies it. The count and slot array remain, but the type table disappears. The active variable-length typed families discussed here are signed integers, unsigned integers, floats, keys, and booleans. [1]

For [1, 2, 3] as a typed signed vector, the companion emits:

TEXT

```
03 01 02 03 03 2c 01
```

The root is now at position 4 and stores distance 3. 0x2c is the packed code for a typed signed vector with one-byte elements. The result is seven bytes: the three element tags have disappeared.

Typed does not mean that an application had to declare a schema before creating the document. A builder can observe the element types at runtime and choose a typed representation. This is an early example of a theme used in the capstone: schemaless interfaces can still exploit homogeneous data.

Fixed typed vectors remove the count too

Numeric tuples of length two, three, or four have dedicated type codes. Their length is encoded in the vector type, so no length prefix or element type table is stored. The three supported numeric element kinds are signed integer, unsigned integer, and float. [1]

For fixed typed [1, 2, 3], the complete document is:

TEXT

```
01 02 03 03 4c 01
```

The three element bytes begin at position 0. The root slot at position 3 stores distance 3. The tag `0x4c` identifies a fixed signed triple with one-byte elements. The total is six bytes.

For a fixed-vector type code `T` between 16 and 24:

TEXT

```
length = 2 + (T - 16) / 3  
element_type = 1 + (T - 16) % 3
```

The divisions are integer divisions. There is no corresponding fixed boolean tuple family in these type codes. A writer must reject unsupported combinations rather than manufacture a plausible-looking arithmetic tag.

The deprecated typed-string family

The historical typed string-vector type removed the per-element tags. That also removed the per-string width information needed to locate each string's length prefix. Strings of different lengths can require different prefix widths, so one vector-level type is insufficient to recover that information reliably.

The pinned reader treats this legacy family using key-like access as a compatibility workaround: it can find a terminating zero but does not recover the full length-prefix semantics of arbitrary strings. Embedded zeros make the loss especially visible. New writers should use an untyped vector of strings, retaining each child's width in its tag, or a key vector when key semantics are actually intended. [1]

Do not interpret the continued presence of an enum value as a recommendation to emit it. Compatibility readers often retain old forms precisely so writers can stop producing them without making old data wholly unreadable.

Count width can dominate element width

A variable-length typed vector still uses one width for its count and its elements. The companion tests the transitions at 255/256 and 65,535/65,536 elements. Even a vector filled entirely with the integer 1 widens at these transitions.

The pinned C++ source also has a direct scalar-vector construction path based on the C++ element size. It checks that the count fits that width rather than promising to widen every such input automatically. API selection can therefore matter at these boundaries. A blob is often the better model for a large uninterpreted byte sequence. [1]

A fixed tuple does not have a count prefix, so this particular count-width issue disappears. Its element values still need to fit the chosen common width, and its float elements still need an implemented floating width.

Typed scans and heterogeneous discovery

A typed numeric vector supports a regular scan: one type and one width can be selected before entering the loop. An untyped vector requires consulting tags unless the consumer has separately proved a stronger invariant. The absence of per-element dispatch can matter more than a few bytes of metadata for a long numerical operation.

Conversely, a heterogeneous vector can mix records, blobs, strings, and numbers without an application schema. Its generality is real, and so is its execution cost. A good system preserves this flexibility at its boundaries while offering typed paths for hot homogeneous regions.

Checkpoint

Exercise 15. Starting from the three encodings of [1, 2, 3], account for every byte removed by the typed and fixed typed forms. Then explain why a typed vector of arbitrary maps is not created by simply adding a new element-type number to the existing arithmetic.

14 A map is two coordinated vectors

Begin with sorted keys

A FlexBuffers map associates string keys with values. Its wire representation is not a hash table. It stores a sorted typed vector of key references and a corresponding heterogeneous vector of values. Position i in the key vector corresponds to position i in the value vector. Sorting must therefore move key/value pairs together. [1]

For the logical map {"foo": 13, "bar": 14}, the wire order is bar, then foo. The order is bitwise lexical key order, not insertion order. A reader can binary-search the key vector, then index the value vector using the resulting position.

The cost of a lookup includes key comparison. Saying only “logarithmic” hides the work needed to compare long strings or strings with a common prefix. Later, the capstone will use interned key IDs and a hash index for repeated mutable lookups, then restore lexical order when exporting.

The map prefix

Let V be the start of the value slots and w their width. Three fields precede the value slots:

TEXT

```
V - 3W : backward offset to the key-vector data
V - 2W : actual byte width K of key-vector elements
V - W : number of values N
V      : value slots, followed by N type bytes
```

The key-vector data has its own count immediately before it, using width K . The key-vector reference in the map prefix is read using w , because that prefix belongs to the value-side layout. Individual key offsets are read using K . The prefix stores the actual width—1, 2, 4, or 8—not the logarithmic packed-type width code. [1]

This creates a third width relationship to remember: root width, map value width, and key-vector width can all be independently relevant. The strings used as keys are still terminated byte strings with no length prefix.

Decode a complete map

The companion emits this twenty-one-byte document:

TEXT

```
62 61 72 00 66 6f 6f 00 02 09 06 02 01 02 0e 0d 04 04 04 24 01
```

Position	Byte(s)	Meaning
0-3	62 61 72 00	bar and terminator
4-7	66 6f 6f 00	foo and terminator
8	02	Key count 2
9	09	Key 0 offset: $9 - 9 = 0$
10	06	Key 1 offset: $10 - 6 = 4$
11	02	Key-vector offset: $11 - 2 = 9$
12	01	Key-vector element width $K = 1$
13	02	Value count 2
14	0e	Value for bar: 14
15	0d	Value for foo: 13
16-17	04 04	Two signed-integer tags
18	04	Root offset: $18 - 4 = 14$
19	24	Map tag, value width $W = 1$
20	01	Root offset-slot width 1

Start at the trailer. The root slot is at 18, and the map data begins at 14. The map tag gives $W = 1$. The prefix is therefore at positions 11, 12, and 13. Position 11 points to the key-vector data at 9. Position 12 tells us to read its elements and count with width 1. Position 8 supplies its count.

Only after those steps do we compare a requested key. Finding `foo` gives index 1, so its value is at $14 + 1 * 1 = 15$, with its tag at $14 + 2 * 1 + 1 = 17$.

Key width and value width need not match

Long keys placed early in a buffer can require two-byte key offsets even when a map has few enough entries, narrow enough values, and a nearby enough key vector for one-byte value slots. Conversely, a large inline numeric value can widen the value vector without widening each key offset.

This is why the key-vector width is stored explicitly in the map prefix. Inferring it from the map value width would work on many small examples and fail on exactly the documents where width compression is doing useful work.

A good test corpus must deliberately produce such different-width maps. Small maps with short keys often exercise only the $W = K = 1$ case and can hide a broken implementation for a long time.

Duplicate keys and absent values

A map must have a well-defined lookup policy. The pinned builder sorts entries and exposes a duplicate-key indication rather than silently making all duplicates retrievable. A new strict writer should reject duplicates or apply a clearly documented replacement policy before serialization. The companion rejects duplicate keys. [1]

A missing key is different from a present key whose value is null. The companion's `find` returns `std::optional<Ref>`, preserving that distinction. A convenience API that returns a null-like reference for missing data can be useful, but it should not be used where presence itself matters.

Likewise, structural verification alone should not be assumed to establish ordering, uniqueness, and equal key/value counts. The companion's strict profile checks those explicitly. Chapter 18 compares that policy with the pinned upstream structural verifier.

Sharing key vectors

The wire layout permits multiple maps to reference the same key vector when their field population and order agree. That can amortize repeated field names and key-offset arrays. Sharing keys individually and sharing an entire key vector are different optimizations.

The pinned C++ source exposes a key-vector-sharing flag, but its `EndMap` implementation still contains a TODO at the point where that sharing would be performed. Do not infer implemented behavior from the enum name alone. The format permits the optimization; this particular inspected code path does not establish that it performs it. [1]

Checkpoint

Exercise 16. Starting from the map's root at position 18, list every load required to obtain the value for `foo`. Identify the base used by every offset and the width used by every load.

15 Building a reader as a set of views

The reference object is not the wire object

The companion represents a reference with these logical fields:

C++

```
struct Ref {  
    ByteSpan bytes;  
    std::size_t slot;  
    unsigned parent_width;  
    unsigned child_width;  
    Type type;  
};
```

This C++ structure is a navigation aid. It is not serialized. Its size is not the size of a FlexBuffers value. Its span borrows an allocation; its position and widths tell the reader how to interpret that allocation. A production library can use more compact fields or specialized views when measurement justifies them.

The important invariant is semantic: `slot` identifies the location from which the parent stores this value. If the value is inline, load it at `slot` with `parent_width`. If it is indirect, read its offset with `parent_width`, compute the target, and interpret the destination using `child_width` and `type`.

Scalar accessors should make conversion policy explicit

The teaching reader's `as_int()` accepts signed direct or indirect integer types. It rejects an unsigned integer rather than silently narrowing it. `as_uint()` and `as_float()` similarly have explicit accepted families. This strictness prevents a malformed or unexpected application shape from quietly becoming a plausible default value.

The official C++ convenience API has broader conversion behavior and default-like results for some mismatches and missing accesses. That can be useful for dynamic data exploration, but it is not a substitute for validation. Applications that depend on exact type and presence should check those conditions explicitly. [\[1\]](#)

A separate conversion API can offer checked conversions: an unsigned value may convert to signed only when it fits; a floating value may convert to integer only under a specified rounding and range policy. Keeping that operation separate makes both correctness and performance easier to reason about.

Container access is mostly arithmetic

For an untyped vector, `at(i)` first checks that `i < count`. It computes the slot using the container width and reads the packed type from the byte-dense type table. The resulting child Ref gets the container width as its `parent_width` and the tag's width as its `child_width`.

For a typed vector, the element type is obtained from the vector type rather than a type table. Inline numeric elements use the vector width as their parent width. A key element follows an offset of that width and scans its target for a terminator. For a fixed numeric tuple, the length comes from the type code rather than a prefix.

This is why separate `Vector`, `TypedVector`, and fixed-vector views can be useful: the view can establish the representation once and provide a simpler loop. The companion keeps the main logic in one `Ref` for readability; the official implementation uses several specialized wrapper classes. [1]

Map lookup is an indexed relationship

A map view first recovers its key-vector view. Binary search compares the requested key with keys at selected indices. Once a match is found, the same index is applied to the value vector. The keys and values must agree in count and order.

A successful bounds check does not establish sortedness. Calling binary search on unsorted keys can return “missing” for a key that is physically present. This is not an out-of-bounds crash; it is a semantic failure. Verification and application API design need to account for both kinds of error.

A useful optimization for repeated lookups is to cache a validated key position or a compiled access path. Such a cache belongs to a particular immutable buffer or a particular mutable version. Reusing it after a structural edit without a version check changes a performance optimization into a correctness bug.

Do not let convenience hide ownership

Consider this incorrect conceptual function:

CPP

```
Ref make_root() {  
    Writer writer;  
    auto bytes = writer.finish(Writer::integer(13));  
    return verify(bytes); // Wrong: bytes dies on return.  
}
```

`auto bytes` copies the returned vector in this example, but that copy is still local. Returning a view into it does not extend its lifetime. Using `const auto&` would instead borrow the writer's storage, which also dies on return. Neither spelling fixes the ownership problem.

Return the owning bytes, or return an owning verified-buffer object whose root view is tied to that owner. A reader library should make its common correct use easier than its common dangling-view mistake.

A trusted fast path is a different contract

After a buffer is verified and immutable, a specialized internal accessor can omit checks that are already established. This is a reasonable optimization. It must not be accidentally exposed as the untrusted-input API.

One design uses a token or owner type produced only by successful verification. Another keeps unchecked accessors private to generated or validated traversal code. The exact technique matters less than making the trust transition visible and preventing later mutation from invalidating it silently.

The companion does not claim to implement this optimized typestate boundary. It leaves many checks in place and exposes plain structs for study. A production API should make representation descriptors opaque enough that callers cannot forge inconsistent type, width, and payload combinations.

Checkpoint

Exercise 17. Design separate `get_signed()` and `convert_to_signed()` APIs. Specify how each handles a null, an unsigned value larger than `INT64_MAX`, a non-integral float, and a missing map key.

16 Building the writer from first principles

Store construction descriptors, not unstable pointers

The companion's Encoded descriptor contains a type, a required width, a payload, and an owner identity. For an inline scalar, the payload holds its bits. For an indirect object, it holds an absolute position in the writer's current byte sequence. Those positions remain valid when the vector backing the writer reallocates, because the bytes retain their logical positions.

The owner identity rejects accidentally using a child from one writer in another writer. The writer itself is non-copyable and non-movable in this teaching implementation so that its identity stays simple. A more elaborate movable builder could use an independent stable state object instead.

Public descriptors are convenient in a laboratory but are not a hardened application boundary. A production implementation should keep constructors controlled so callers cannot claim that a wide payload requires only one byte or forge a child position that was never emitted.

Emit children eagerly

Strings, keys, blobs, and indirect scalars can be written when they are created. The writer records their payload positions and widths in descriptors. A vector or map can then be constructed from descriptors for already-existing children and inline scalar values.

For a string, choose the width of its length, align, write the length, remember the content start, write the bytes, and write the terminator. For a blob, omit the terminator. For an indirect scalar, align to the scalar's width, write its payload, and remember that position.

Nothing in this process requires final buffer addresses. Only the current logical length and the fixed format rules matter. This is the benefit of position-based construction state.

Close a vector in two phases

The first phase solves its width using the candidate algorithm in Chapter 11. The second phase writes the chosen representation. For an ordinary heterogeneous vector, that means padding, count, slots, and type bytes. For a typed vector, omit the type bytes. For a fixed numeric tuple, omit the count as well.

When writing an inline scalar's type byte in a heterogeneous vector, the stored width reflects the actual parent slot width. When writing an indirect value's type byte, it preserves the child's width. This is the writer-side counterpart of the reader's two-width rule. [\[1\]](#)

A useful invariant is: after width selection succeeds, the emission phase should not discover a new representability problem. Keep the arithmetic used by the solver and the arithmetic used

by emission visibly equivalent. The companion still checks offset fit during writing so a defect fails loudly rather than silently truncating bytes.

Close a map without losing associations

The map builder accepts pairs. It sorts those pairs by encoded key bytes, rejects duplicates, writes or reuses keys, constructs the typed key vector, and then constructs the value vector with its three-field map prefix.

Avoid sorting a raw interleaved array through an unrelated aggregate pointer in a new implementation merely because it looks compact. A vector of actual pairs makes the object model and association invariant explicit. Only replace that representation after profiling and a careful aliasing review.

The companion does not pool keys or strings. That keeps the initial implementation small and makes fixture layouts easy to inspect. Pooling is a separate optimization that must preserve ownership, offset validity, and mutation semantics. The official builder's flags and pools provide useful source material once the basic writer is correct. [\[1\]](#)

Finish the document

The root has no parent container to tell the reader its slot width. The writer therefore selects and aligns the root slot, writes the root value or backward offset, writes its packed type, and finally writes the actual root slot width. After finishing, the writer is sealed.

There must be exactly one chosen root. A stack-based public builder enforces matching start/end calls and a single remaining root value. The companion uses explicit returned descriptors instead, so its `finish(root)` names the root directly. Both are API choices over the same trailer representation.

Sealing prevents ordinary writes from appending bytes after the trailer and changing which value `GetRoot` would discover. Reusing a builder requires resetting all relevant construction state, not just clearing the byte vector. Pools, duplicate tracking, pending values, and finished state deserve tests of their own.

Exception and allocation behavior

The educational writer does not promise transactional rollback after every allocation failure. An operation may have appended some unreachable construction bytes before throwing. No completed output is returned from the failing operation, but the caller should not assume the builder remains suitable for arbitrary continued use.

A production builder can provide a strong guarantee by recording a checkpoint of byte length and scratch state, then rolling back on failure. Pools complicate rollback because they may have published references to newly written positions. A temporary operation arena or staged pool insertion can keep those updates atomic at the API level.

Do not call a builder “allocation-free” merely because it appends to one vector. Capacity growth, temporary descriptors, sorting storage, and pooling indexes all count. Reserve intelligently, measure allocation count, and distinguish the final buffer size from peak construction memory.

Checkpoint

Exercise 18. Add a builder checkpoint/rollback operation. List every state component that must be restored when an attempted map construction fails after writing some keys but before writing its value vector.

17 Mutation, sharing, and why one outlier matters

What the official mutation methods can do

The pinned C++ Reference exposes experimental mutation methods for numeric values, booleans, and strings. Numeric updates can succeed when the replacement fits the existing representation. The string mutation path requires exactly the existing byte length. These operations do not turn the buffer into a general growable object store. [1]

The companion's `mutate_int` is deliberately narrower: it accepts signed direct or indirect integers, verifies that the supplied mutable owner is the allocation the reference borrows, and returns false when the new value needs a larger width. Failure leaves the value unchanged. Its same-length string mutation also requires valid UTF-8 according to the teaching profile.

A same-width byte update still requires writable storage. A `const` view does not prove the allocation was originally mutable. A buffer backed by read-only memory or a truly `const` object must not be written through a cast. Nor is a multi-byte update automatically safe against concurrent readers.

The scalar outlier example

Consider an untyped vector of 100 signed integers. Ninety-nine values are 1, and one is `INT64_MAX`. If all are inline, every slot needs eight bytes. The type table needs another 100 bytes. The count occupies eight bytes, and the root needs enough offset width to reach back over the vector.

The companion emits **912 bytes** for this layout. When the wide value is stored as an indirect integer before the vector, the other 99 inline values and the offset to the outlier all fit in one-byte slots. The complete document becomes **212 bytes**.

These are generated fixture sizes, not estimated benchmark speedups. The reduction is exactly 700 bytes for these particular layouts. Reading the exceptional value now includes an extra indirection. Building it required an additional out-of-line object. The smaller working set may help many operations, but size reduction alone does not establish lower latency for every access pattern.

Derive the approximate trade-off

Ignoring padding and root differences for a moment, widening an N -element heterogeneous vector from w to W adds approximately $N * (W - w)$ slot bytes, plus any count-width difference. Moving one scalar out of line adds its payload and replaces it with an offset that must fit the narrow parent.

The potential saving is therefore large when N is large and the exceptional values are few. It is small or negative for tiny vectors, many exceptional values, or large offset distances that widen the parent anyway. A real optimizer must solve the resulting layout, not use the approximation as a guarantee.

This is an example of a broader design principle: local compression decisions can have nonlocal effects when containers use a shared stride. The cheapest representation of one value in isolation may not minimize the size of the containing document.

Reserving width is not reserving arbitrary capacity

The official builder has `ForceMinimumBitWidth`, which influences vector/map slot width in the inspected implementation. This can provide room for later numeric changes in those slots. Its use in vector construction should not be generalized into a promise that every standalone root scalar or separately emitted indirect scalar is automatically widened by that setting. [1]

Even a deliberately wide slot does not provide room to append elements. A vector's type table begins immediately after its current slot array, and its count is part of the layout. A string's length and terminator describe its current content. Reserving native allocation capacity in the backing `std::vector` does not create a standard capacity field in the wire format.

Choose the reservation policy according to the expected edits. A few counters can be given wide slots. A frequently growing collection is a different problem and usually deserves a different live representation.

Moving bytes changes some offsets, not all

Let an old relative distance be $D = P - T$, where P is the offset slot and T its target. If an edit moves them by different amounts, the new distance is:

TEXT

```
D_new = D + movement_of_slot - movement_of_target
```

Moving the entire buffer as one unit leaves every internal distance unchanged. Inserting bytes between a target and its referencing slot changes that distance. Moving a subtree as a block can preserve internal references while changing references that cross the subtree boundary.

This is why “relative offsets make relocation easy” needs a scope. Whole-buffer relocation is easy. Arbitrary internal insertion is not automatically easy. Every crossing reference, every width threshold, and every affected container boundary must still be accounted for.

Aliases make mutation observable elsewhere

The companion test writes one string object and places two references to it in a vector. Replacing “Ada” with “Bob” through the first reference makes the second reference read “Bob” too. No bytes moved, no widths changed, and the effect is still broader than one field.

A mutable API must choose whether shared objects have identity semantics or value semantics. Identity semantics make aliases intentionally observe mutations. Value semantics require replacement or copy-on-write before a shared payload is changed. The capstone uses stable identity for containers and treats text values as replacement values in its baseline, avoiding an accidental mixture.

Checkpoint

Exercise 19. Reproduce the 912-byte and 212-byte fixtures. Explain the root width in each, not just the vector payload. Then construct a case where indirection increases total size.

18 Verification is a small security boundary

Begin with the threat model

A buffer from a file, network, plugin, or another language must be treated as an arbitrary byte sequence until validated. It may be truncated, contain invalid widths, describe enormous lengths, point outside itself, repeat the same expensive object many times, or encode a shape that is structurally readable but semantically invalid for your API.

A verifier should state which of these problems it handles. Bounds safety, canonical encoding, UTF-8 validity, sorted map keys, duplicate policy, recursion limits, and application rules are not interchangeable. A function named `VerifyBuffer` is only as strong as its actual checks.

The order of checks matters

Start with the trailer length and root width. Construct a bounded root descriptor. Validate the type code. For an inline value, validate the slot extent and any profile rules for its type. For an indirect value, read the offset using the parent width, check that subtraction is permitted, and only then resolve the target.

Next validate any prefix before reading it. A string needs a length prefix and a content range. A vector needs a count, a checked multiplication by stride, and possibly a type table. A map needs both vectors and the relation between them. A fixed tuple has no count prefix, so its type determines the extent.

The safe arithmetic patterns from Chapter 2 should appear repeatedly. Large unsigned fields are not inherently suspicious; unbounded arithmetic using them is. A verifier must be correct at the maximum representable values, not merely at typical application sizes.

Structural verification versus a strict application profile

The pinned upstream FlexBuffers verifier checks many structural properties and supports depth limits and an optional reuse tracker. Its inspected implementation does not establish every property a strict map/text API might expect. In particular, its map verification does not compare key and value counts or enforce sorted unique keys; its string terminator helper checks that the terminator position is in range rather than checking the byte's value; it does not perform UTF-8 validation. [1]

Those are observations about the pinned code, not a claim that all upstream versions have the same behavior or that the verifier is intended to enforce every possible application convention. The engineering lesson is to read the checks you depend on and add a profile layer where necessary.

The companion's default profile additionally requires valid UTF-8 for text, an actual zero string terminator, equal map counts, strictly increasing unique keys, canonical boolean/null

payloads, and implemented float widths. It rejects reserved type tags. It can allow or reject the deprecated typed-string family, with its documented key-like interpretation.

Budgets are correctness features

A recursive implementation needs a depth budget to prevent stack exhaustion. A work budget prevents a small-looking root from causing unbounded repeated traversal. A byte-scan budget is also valuable because a million references to one very long key can repeatedly trigger expensive scans even when the number of visited nodes is bounded.

The teaching verifier has depth and visit budgets; it does not implement a separate total byte-scan budget or reuse memoization. Its malformed-input tests use bounded input sizes. A production verifier should add those resource controls and report budget exhaustion separately from structural corruption.

A fully iterative traversal can replace recursion with an explicit work stack. That removes reliance on the machine call stack, but it does not remove the need to bound total work and temporary memory. An attacker can exhaust an unbounded heap worklist too.

Sharing and memoization

When the same immutable object is referenced many times, a verifier can remember that it has already validated that interpretation. A cache key must include the buffer identity and enough information to distinguish different interpretations, such as target position, type, and child width. Reusing a result for the same address under a different type is not automatically valid.

A traversal should distinguish an object currently being explored from one whose validation is complete. That distinction detects cycles or overlapping pathological interpretations rather than treating an incomplete result as a proof. Normal builders produce children-before-parents structures, but a verifier must cope with arbitrary bytes, not just normal builders.

Memoization consumes memory. The pinned implementation offers an optional byte-indexed reuse tracker with a memory cost proportional to buffer size. Other designs can use sparse tables or bounded caches. The correct choice depends on sharing density and the allowed validation memory budget. [\[1\]](#)

Verification has a lifetime

A successful result applies to the bytes that were checked. If a caller then changes a length, tag, or offset, later unchecked navigation may be unsafe. If another thread changes payload while it is read, C++ synchronization rules apply regardless of whether metadata stays valid. [\[13\]](#)

An immutable owned snapshot is the simplest durable verification contract. A mutable buffer can support verified access only with explicit mutation rules, versioning, and synchronization.

Do not cache “verified” as a permanent property of an allocation that remains arbitrarily writable.

Checkpoint

Exercise 20. Design an iterative verifier work item and a memoization key. Explain how your design distinguishes a completed shared child from a cycle, and how it limits both visited nodes and scanned text bytes.

PART IV · ENGINEERING FOR QUALITY AND SPEED

19 What “close to official quality” actually requires

Correct output is only the first gate

A writer that emits one readable map has demonstrated a mechanism. A library that can replace a mature implementation needs a much broader body of evidence: complete type coverage within its advertised profile, adversarial input handling, stable API contracts, reproducible builds, cross-platform behavior, interoperability, and sustained testing.

The companion deliberately does not claim that status. It is a base you can understand completely and improve. Its strengths are explicit arithmetic, small independent components, deterministic fixtures, and an executable semantic model. Its current limits include no executed upstream oracle, no long-running production fuzz history, no measured optimized allocator or SIMD engine, and no concurrent mutable API.

This is not a reason to avoid implementing the format. It is a reason to turn “quality” into a set of gates rather than a feeling about how clean the code looks.

Test categories that catch different mistakes

Golden byte tests verify exact examples. They catch reversed offset directions, wrong tags, incorrect trailer placement, and changes in a deliberately fixed encoding policy. Boundary tests exercise signed and unsigned transitions, string lengths, vector counts, and offset-distance thresholds.

Property tests start from a logical model, encode it, decode it, and compare meaning. They should preserve signed versus unsigned distinctions and define float equivalence explicitly. Metamorphic tests make transformations whose effect is known: reorder map insertion order, reuse a child, re-encode a verified document, or widen a permitted storage representation without changing its logical value.

Malformed-input tests mutate trailers, tags, lengths, offsets, key ordering, terminators, and nesting. Allocation-failure tests inject failure at every allocation point and inspect invariants afterward. Stateful mutation tests compare a sequence of operations against a simple reference model, not just the final serialized buffer.

A self-round-trip can hide a shared bug between your writer and reader. An independent oracle is therefore a separate gate, not a redundant one.

Build an interoperability matrix

Test your writer against the official C++ reader and verifier. Test the official C++ writer against your reader. Add a second independent language implementation for the supported types. Compare logical values rather than requiring byte identity, except for fixtures whose exact layout is intentionally prescribed.

Record the exact upstream commit or release and all builder options. Test sharing, indirect numeric values, empty containers, embedded zeros in ordinary strings, the signed and unsigned extremes, different parent/child widths, and different map key/value widths. A “JSON-shaped” corpus alone omits important FlexBuffers types and edge cases.

Legacy forms deserve their own column. A compatibility reader may accept a deprecated form with limited semantics while a modern writer intentionally refuses to emit it. That is a legitimate policy when documented; hiding the distinction behind a generic success result is not.

The companion includes optional C++ and Python oracle smoke-test files. Neither was run for this edition. The C++ test is bidirectional for a small representative document; the Python script reads generated fixtures. Passing them later would still not constitute the full matrix described here.

Sanitizers and coverage-guided fuzzing

AddressSanitizer and UndefinedBehaviorSanitizer help expose memory errors and selected undefined behavior on executed paths. ThreadSanitizer becomes relevant when concurrent code is introduced. None establishes the absence of all defects or the correctness of application semantics. [18–20]

A coverage-guided fuzzer uses instrumented execution feedback to seek new paths. The included `fuzz_verify.cpp` is a bounded entry point for malformed bytes. A 25,000-run libFuzzer smoke test completed for this edition with no reported sanitizer failure. This is stronger than a few hand-picked malformed examples, but it is still a short run against one component on one environment. [21]

The run began with 15 fixtures and ended with 125 retained corpus entries. The report records 233 covered instrumentation counters out of 438 loaded counters. That is not a claim of 53 percent semantic conformance, nor a percentage of the complete project covered. Instrumentation coverage and protocol coverage measure different things.

Stateful testing for the mutable engine

Generate operations such as insert key, replace value, append element, erase element, grow a string value, hold a handle, release a root, collect garbage, freeze, and thaw. After each step, compare the visible logical state with a simple model built from standard containers.

Keep deliberate stale handles and verify that they fail after reclamation and slot reuse. Keep valid aliases and verify that container mutations have the documented shared-identity behavior. Create cycles and verify that export rejects them. Test failure during growth before publication and confirm that the old state remains usable under the promised exception guarantee.

The test generator itself should be small and reviewable. A complex reference model that shares your allocator, handle logic, or width solver can reproduce the same bug rather than detect it.

The actual receipt for this edition

The final deterministic suite contains 20,220 assertions, including 2,000 generated logical documents and their re-encodings, plus 30,000 bounded random byte probes. GCC 14.2 and Clang 17 with address/undefined-behavior sanitizers are the tested compiler configurations. CMake/CTest and the demonstration program were also executed.

The receipts, test source, fuzz entry point, and generated byte fixtures are supplied so these claims are inspectable. A future change should rerun them and expand the corpus before it inherits any claim from this edition.

Checkpoint

Exercise 21. Write the first ten entries of your independent interoperability matrix. Include at least two cases that could pass a self-round-trip while still being incompatible with the official implementation.

20 Hardware-aware optimization without unsafe shortcuts

Count bytes, dependencies, and work

A useful first model of an operation includes bytes touched, dependent loads, branches, allocation work, and synchronization. These quantities are not interchangeable. A compact representation can reduce memory traffic while adding decoding instructions. A pointer-rich representation can simplify decoding while increasing cache and translation pressure.

For a sequential scan, an elementary lower bound is $\text{bytes_read} / \text{sustainable_bandwidth}$, before accounting for computation and dependencies. For a chain of dependent references, bandwidth may not be the main limit because the next address cannot be known until an earlier load completes. These models are not benchmark results; they help select measurements and explain why two layouts behave differently.

A machine's actual cache-line size, memory hierarchy, instruction set, and compiler output should be measured or queried for the deployment target. Do not build a portable wire format around a universal assumption that every relevant line is exactly 64 bytes.

Compression can beat native convenience

A native 16-byte generic cell is convenient for mutation, but it is not automatically compact. A small FlexBuffers value can use a narrow slot and a byte tag. On a read-heavy working set, moving fewer bytes through the cache can outweigh the extra decoding operations.

Typed vectors make this especially interesting. Their type and width can be selected once outside the loop. A native double array may be ideal for repeated arithmetic and updates, while a wire vector of exactly representable floats may use less memory. Conversely, the wire count/stride coupling can widen small integer elements for long vectors. Compare the actual resulting layouts, not slogans about “native” or “compressed.”

The capstone therefore preserves a typed-array path and an immutable wire path rather than declaring one representation universally superior.

Specialize outside the loop

A generic scalar accessor may branch on width for every element. A scan can dispatch once:

TEXT

```
switch vector width:
  1: run a one-byte load loop
  2: run a two-byte load loop
  4: run a four-byte load loop
  8: run an eight-byte load loop
```

The same idea applies to type. Once a typed numeric vector has been validated, the inner loop does not need to rediscover that every element is numeric. This can expose a regular loop to the compiler without introducing a new wire format.

LLVM's documentation describes loop and straight-line vectorization and the diagnostics used to see why a loop was or was not vectorized. Use those diagnostics and inspect generated code before writing architecture-specific intrinsics. A clean, fixed-stride loop can be a better first optimization than a complicated hand-vectorized generic decoder. [16]

SIMD is particularly useful for metadata

Dense control bytes can answer several lookup questions in parallel before touching larger payloads. Abseil's Swiss-table design is an important example: small hash fingerprints filter candidates in a metadata group, after which actual key equality is checked. Empty and deleted states have different probe behavior. [11]

The capstone borrows that principle for a proposed native map, not Abseil's exact internal ABI. The performance opportunity is to avoid fetching full key/value slots for most non-candidates. A matching fingerprint is never sufficient to establish a matching key.

SIMD reads must remain within allocated readable memory. Either probe aligned full groups or explicitly allocate and maintain safe trailing metadata for wraparound loads. The fact that a comparison is vectorized does not exempt its load from bounds and lifetime rules.

Prefetching and batching

A regular scan is often already predictable to the processor. Manual prefetching can add instructions without helping. For irregular graph traversal, prefetching can help only when the target is known sufficiently before it is needed and there is useful independent work to overlap.

Batching independent lookups is often a more general way to expose memory-level parallelism. Instead of following one complete chain before starting the next, resolve the first step for several independent queries, then continue. This changes latency and scheduling behavior, so benchmark both individual-request latency and batch throughput.

A GPU is not an automatic answer for small dynamic maps. Packing, transfer, synchronization, and branching can dominate. Large homogeneous arrays with substantial computation per transferred byte are a more plausible candidate. That is a workload argument, not a claim about one current GPU model.

Allocation, locality, and false sharing

Grouping related small objects can reduce allocator overhead and improve locality. Grouping unrelated frequently written counters on one cache line can instead create false sharing. The Linux kernel documentation explains this write/read and write/write coherence effect and points to tools such as `perf c2c` for diagnosis. [25]

The design implication is selective separation. Keep immutable payload data dense. Avoid placing highly contended ownership counters in the same hot line as read-mostly data. Do not pad every value to a cache line: the resulting memory expansion can be worse than the contention it was intended to prevent.

NUMA placement and large-page policies are further deployment choices, not prerequisites for the core format. Start with locality and measured allocation behavior. Add machine-specific memory policies only when a real working set and deployment topology justify them.

Checkpoint

Exercise 22. Compare a million-element scan in a generic cell vector, a native typed array, and a verified typed FlexBuffer. List the bytes and decisions each iteration needs. Identify which decisions can be hoisted out of the loop.

21 What modern languages help with—and what they do not

C++20 is enough for the core

The companion uses C++20 facilities such as `std::span`, `std::bit_cast`, fixed-width integers, and ordinary RAII containers. None requires a future language feature to express the format safely. A newer language mode can improve ergonomics, but it does not change the wire invariants.

`bit_cast` makes bit-preserving conversion between suitable equal-sized trivially copyable types explicit. It does not validate arbitrary bytes as a pointer, create the lifetime of an unrelated object in a buffer, or repair an out-of-bounds load performed before the cast. [12]

Likewise, a `span` expresses a borrowed range but does not own it. The most valuable modern-C++ improvement is often an API whose types distinguish owned immutable buffers, borrowed verified views, stable logical handles, and temporary mutable borrows.

Allocation resources are policy hooks

Polymorphic memory resources let containers receive allocation policy without embedding one concrete allocator in every public type. A monotonic resource is useful when an entire build or transaction is discarded together. A pool resource can be more appropriate when similarly sized objects are repeatedly allocated and released. [14]

A monotonic allocator is not a general answer to a long-lived mutable document. If individual replacements never reclaim storage until the whole arena is destroyed, steady-state update churn can make resident memory grow without bound. The capstone separates temporary build memory from reusable live storage for this reason.

Allocator choice should remain replaceable until measurements show that a custom slab is worth its maintenance burden. Start with correct lifetimes and allocation accounting. A faster allocation path that loses rollback or reclamation correctness is not an improvement.

Rust changes how borrowing is expressed

Rust can encode many ownership and borrowing constraints directly in types. A reader over a borrowed byte slice can keep derived views tied to that slice's lifetime. A mutable borrow can prevent simultaneous safe-code access that would conflict with mutation.

Raw-pointer construction still requires care. The standard library's `slice::from_raw_parts` contract includes alignment, validity, a single allocation, and lifetime conditions. A buffer being “just bytes” does not permit fabricating arbitrary typed slices without satisfying those requirements. [22]

The corresponding design lesson applies back to C++: treat an exposed typed span as a capability with a limited lifetime, not as a permanent address promised by a stable logical handle. A safe language can help enforce the rule, but it does not choose your representation or export semantics for you.

Managed runtimes still need an ownership boundary

A JVM-facing library can retain a byte buffer and expose bounded accessors instead of allocating a boxed object for every scalar. Java's Foreign Function and Memory API provides memory segments with spatial and temporal bounds and arena-controlled lifetimes; thread-access rules depend on the segment's scope. The JDK 22 documentation is the concrete API baseline referenced here. [\[23\]](#)

The architectural issue is the same whether a binding is written in Java, Kotlin, or another JVM language: avoid a foreign-function call or heap wrapper per numeric element in a hot loop. Batch operations and expose bulk numeric access where lifetime and alignment rules permit it.

A native allocation is not automatically safe merely because a managed wrapper refers to it. The wrapper must retain the owner or enforce a closed-state check. Conversely, pinning or retaining large managed allocations indefinitely can have costs of its own. Keep the boundary explicit and measure it.

JavaScript has a numeric distinction you must preserve

A JavaScript Number does not exactly represent every 64-bit integer. A binding that converts arbitrary FlexBuffers signed or unsigned 64-bit values to Number can silently lose information. The ECMAScript DataView API includes BigInt-based 64-bit operations and explicit endianness parameters for multibyte loads. [\[24\]](#)

A suitable binding can return BigInt for the full integer domain, or return Number only after a checked exactness condition. That decision is part of the public semantic contract. The fact that a value happened to be encoded in an eight-byte slot does not by itself determine whether its numeric value is safe as a Number.

JavaScript objects also do not automatically preserve a blob/string distinction, signed/unsigned intent, fixed tuple representation, or sharing identity. A lossless bridge needs explicit tagged values or a documented normalization policy rather than pretending that plain JSON is the whole FlexBuffers domain.

Bindings should expose bulk capabilities

A useful cross-language interface has operations such as verify buffer, obtain root, resolve path, iterate a map, borrow or copy a typed numeric range, and freeze a mutable document. It should not require repeated tiny crossings merely to scan a vector.

Stable handles are useful at these boundaries because they can be validated after storage moves. A raw address exported to another runtime carries a much stronger lifetime promise. The capstone's handle API and short-lived borrows are designed to make that distinction visible.

Checkpoint

Exercise 23. Specify how a JavaScript binding returns signed 64-bit integers, unsigned 64-bit integers, blobs, and strings with embedded zero bytes. State which conversions are exact and which require an explicit opt-in.

22 Benchmark the workload, not the advertisement

Start with comparable semantics

A benchmark comparing a mutable native map with an immutable buffer must decide what happens when an edit does not fit the buffer. Ignoring failed edits gives the immutable representation an artificial advantage. Rebuilding the buffer after every edit when the real application batches a thousand edits gives it an artificial disadvantage.

Define the actual workflow first: number of reads, number of updates, which updates are structural, how often snapshots are needed, and how long old snapshots remain alive. Then compare implementations that complete the same logical work under the same safety and ownership requirements.

Useful baselines include ordinary standard-container objects, the official FlexBuffers builder plus reader, its limited in-place mutation path when applicable, the companion's semantic laboratory, and the optimized capstone as it is implemented. The proposed optimized engine has no benchmark result in this edition because it is not yet implemented.

A concrete workload matrix

Use small maps with 4 and 16 keys, medium maps with 64 keys, and larger maps with 256 or more keys. Vary key length and common-prefix structure. Include successful and unsuccessful lookup. Distinguish lookup by an existing interned key ID from lookup by an arbitrary string that first has to be hashed and interned.

For vectors, include small tuples, thousands of values, width-boundary counts, and million-element numerical arrays. Vary homogeneous versus heterogeneous values, string lengths, indirect outliers, sharing density, and nesting. Test append, random replace, middle erase, full scan, and freeze after a mutation batch.

Use both read-heavy and write-heavy traces. A realistic mixed trace is more informative than a universal score produced by averaging incomparable microbenchmarks.

Measure the full cost ledger

Report construction time, verification time, lookup/update time, freeze time, thaw time, allocation count, bytes allocated, peak live bytes, retained capacity, and snapshot-retention cost. Resident set size is useful but does not replace internal allocation accounting because allocator caches and unrelated process memory can affect it.

Latency distributions matter for growth and reclamation. Most appends may be cheap while an occasional reallocation copies a large array. An average hides that spike. Measure growth events separately and replay realistic bursts to see their effect on request latency.

A microbenchmark framework can help with repetition, warmup, and result handling. Google Benchmark's user guide also explains the role and limits of optimization barriers such as `DoNotOptimize`. A barrier is not permission to use compile-time-constant inputs and assume the entire intended computation remains. [17]

Prevent common measurement errors

Keep input generation outside the timed region unless generation is part of the workload. Consume results in a way that prevents dead-code elimination. Distinguish checked access from trusted post-verification access rather than quietly comparing one to the other. Record compiler, flags, CPU, memory configuration, operating system, and relevant library revisions.

Warm and cold working sets answer different questions. A tiny repeatedly accessed map can stay in cache; a large shuffled corpus may not. Choose working-set sizes relative to the measured machine rather than calling a fixed number of megabytes “cold” everywhere.

For per-operation tail latency, account for timing overhead and scheduling. A benchmark that times only a large batch and divides by its length reports an average, not the 99th percentile of individual operations. An end-to-end service trace should also distinguish processing time from queueing delay.

Use counters to explain, not decorate

When available, hardware counters can help determine whether an improvement reduced instructions, branch misses, cache misses, or coherence traffic. They should answer a hypothesis. A faster result accompanied by more instructions may be perfectly reasonable if it removes dependent cache misses.

Do not require every optimization to reduce every counter. Do not compare a counter's absolute value across incompatible measurement setups. The useful question is whether the observed change matches the proposed mechanism under controlled conditions.

A simple amortization equation

For a native working representation, model total time as:

TEXT

```
T_native = import_cost
          + reads * native_read_cost
          + updates * native_update_cost
          + snapshots * freeze_cost
```

For a wire-based alternative, use its real read cost and the actual mix of fitting mutations and rebuilds. Keep memory as a separate constraint or an explicitly priced objective; do not add bytes directly to nanoseconds.

A native layer is worthwhile when the savings during repeated work outweigh conversion and snapshot costs while staying within the memory budget. This break-even point can move

dramatically when snapshot frequency changes. It is the central performance question for the capstone.

Checkpoint

Exercise 24. Design a benchmark where rebuilding after every update is appropriate and another where it is not. Explain how the snapshot schedule changes the winning representation even with identical logical data.

23 Optimizations worth earning

Improve the representation before the instruction

A hot loop that repeatedly hashes the same field name may benefit more from an interned key handle than from a faster bitwise hash. A repeated numerical scan may benefit more from a typed span than from a branchless generic-value decoder. A builder that reallocates many times may benefit more from an accurate reserve than from a clever scalar store.

This is an order of investigation, not a prohibition on low-level work. Measure the dominant operation, identify the repeated work it performs, and ask whether the API or representation can remove that work entirely. Only then optimize the remaining inner loop.

Runtime shapes do not require a declared schema

Suppose many maps happen to contain the same keys. A runtime can intern that key population as a shape and store only values plus a shape identity in each object. Field lookup can validate the shape and use a cached slot. The shape was discovered from data rather than supplied as a compile-time schema.

This is a promising extension for repeated record-like objects. It is not the first implementation step. Arbitrary dynamic keys can create unbounded shape diversity, and frequent insertion/removal can cause shape churn. A shape cache therefore needs admission and eviction rules, versioned access paths, and a fallback to the general map.

The capstone keeps shapes optional. It first earns the simpler wins from compact cells, typed vectors, interned keys, and a measured small-map/hash-map transition.

Cache a path only with its assumptions

A path such as `root["settings"]["volume"]` can be compiled into a sequence of key IDs or slot hints. A slot hint is valid only while the corresponding container layout remains compatible. A stable container handle does not imply that a member stays in the same bucket after rehashing.

A useful path cache records the handle, generation, and structural version for each cached step. On mismatch it resolves the step normally and refreshes the cache. A scalar replacement that does not change structure may preserve the path, while an insertion or rehash invalidates it.

Version wraparound and object reuse must be handled deliberately. A generation prevents a recycled object slot from impersonating an old object; a structural version prevents a changed layout from impersonating an old layout. They solve different problems.

Avoid premature universal compression

NaN boxing can make some dynamic values fit in one machine word, but it consumes representational space and interacts with full-width integers, pointer assumptions, NaN payloads, and portability. Tagged pointers similarly depend on address and alignment assumptions that are not the FlexBuffers wire contract.

The capstone chooses a straightforward 16-byte cell as its baseline. That is not the minimum conceivable size. It is enough space for exact 64-bit scalar payloads, an explicit kind, and a validated external handle without borrowing bits from a processor-specific pointer convention.

If profiling later shows cell bandwidth dominates, add a measured specialized representation for the affected region. A typed numeric vector is a simpler and often more powerful compression of repeated type information than forcing every generic cell into an exotic universal encoding.

Make every optimization reversible

Keep a readable scalar path beside SIMD code. Keep the standard-container laboratory beside the custom allocator. Keep the strict verifier beside a trusted fast accessor. Keep a reference semantic model beside adaptive containers.

The point is not to ship all paths in every production configuration. It is to preserve independent ways to check behavior and to make performance work removable when it fails to earn its complexity. A design is easier to improve when optimizations are modular rather than entangled with its meaning.

Checkpoint

Exercise 25. A profile shows that 60 percent of lookup time is spent hashing field-name strings that repeat across requests. Design the smallest API change that removes most of that work, and specify how it prevents a key handle from one document being used in another.

PART V • LIVEFLEX: THE MUTABLE CAPSTONE

24 Capstone: choose the compatibility contract first

The engine we are designing

Call the capstone **LiveFlex**. It is a schemaless in-memory value engine with native mutable containers and standard FlexBuffers import/export. Its primary workload is a working document that receives many small edits and repeated lookups between snapshots. Numeric arrays should support bulk computation without per-element boxing. Structural growth should not rewrite unrelated parts of the document.

The design is for a 64-bit process with a single owner of mutable state. Concurrent readers initially receive immutable snapshots, not unsynchronized access to the mutable heap. Counts and handle indices use explicit bounded integer domains; oversized inputs produce errors rather than implicit truncation. Export has a configurable size and depth budget chosen to match the intended consumers.

The supported logical domain includes null, bool, signed and unsigned 64-bit integers, floating values, UTF-8 strings, keys, blobs, vectors, and string-keyed maps. Homogeneous native arrays are an internal specialization of vectors. Application-specific timestamps or identifiers require an explicit application convention; they are not silently assigned unused wire type codes.

What compatible means here

Every successful export must be an ordinary FlexBuffer under the selected profile. No custom reader, absolute pointer, external arena lookup, or private type code is needed to read that snapshot. Import accepts the supported ordinary wire forms and normalizes them into the live value model.

The live bytes are **not themselves** an ordinary FlexBuffer. A native cell, descriptor, hash index, or spare-capacity region is not passed to `flexbuffers::GetRoot`. Compatibility exists at the explicit freeze/thaw boundary. Exact byte identity, original scalar widths, original type-table choices, and original sharing identity are not preserved unless a separate fidelity mode is implemented.

Container sharing is allowed in live memory. Cycles can be staged by the low-level laboratory API, but an exported root must be acyclic within the supported traversal limits. Export detects a cycle and fails. A stricter application can forbid cycle-creating edits or validate at transaction commit. Do not promise constant-time arbitrary graph insertion and free acyclicity checks at the same time.

Why not just add capacity to a FlexBuffer?

There is no standard capacity field. A sidecar can remember reserved free bytes, but ordinary readers still interpret count, slots, and type tables according to the current wire layout.

For a typed numeric vector with a sufficiently wide count and reserved space, append can indeed be cheap: write the new element and update the count. This is a useful bounded-mutation design. For an untyped vector, however, the type table must begin immediately after the logical slot array. Appending one slot requires moving the existing type bytes out of its way. Even with spare storage, that step is proportional to the current tag count.

An indirect append has another constraint: the new child's bytes must be before the new offset slot. A child appended after an existing parent cannot be reached by a standard backward offset from that parent. An allocator could reserve earlier space, or the parent could be moved/rebuilt later, but either choice introduces explicit capacity or relocation constraints.

Maps add lexical order. Inserting a key can shift key and value slots, requiring their relative offsets to be adjusted. Spare allocation capacity does not remove these ordered-layout obligations. A continuously compatible wire arena is possible for a defined mutation envelope; it is not a universal constant-time editing structure.

A genuine continuously compatible alternative: an append log

Another valid design appends a changed child, then appends rebuilt ancestors and a new root trailer. Old bytes remain in the buffer. The new root can point backward to both unchanged old subtrees and newly emitted children. An ordinary reader using the new published extent can read the latest snapshot.

This **WireLog** design preserves the standard wire grammar after each completed publication. It is attractive for infrequent edits and substantial subtree reuse. Its costs are ancestor/container rebuilding, accumulated unreachable bytes, periodic compaction, and careful ownership when the backing allocation grows. Existing views cannot survive a `std::vector` reallocation merely because internal offsets remain correct.

For a shared child, rebuilding only one path gives replacement semantics along that path, not automatic mutation of every alias. Identity-wide updates require finding and rebuilding all affected references or a different policy. Again, compatibility does not determine mutation semantics by itself.

LiveFlex chooses native editing because its target workload includes repeated arbitrary growth. WireLog remains a serious alternative to benchmark, not a straw man dismissed as impossible.

The non-goals keep the design small

The baseline is not a database, a garbage-collected language VM, a universal lock-free graph, or a new persistent wire standard. It does not include a query language, write-ahead log, distributed identity, or arbitrary user-defined objects. Those can be layered later if a real application requires them.

The central fast paths are simple: update an inline scalar, append within capacity, resolve a stable handle, find an interned key, scan a typed array, and export a bounded acyclic root. Everything else should preserve these paths rather than burden every cell with speculative features.

A proposed public surface

The exact C++ types can evolve, but the operations should make costs visible:

TEXT

Document

```
make_map(), make_vector(), make_numeric_vector(kind)
make_string(bytes), make_blob(bytes)
intern_key(text) -> document-scoped Key
set(map, Key, Value), find(map, Key), erase(map, Key)
append(vector, Value), replace(vector, index, Value)
borrow_numbers(vector, write_guard) -> limited-lifetime span
register_root(Value) -> RootToken
collect()
freeze(root, export_options) -> owning immutable buffer
thaw(verified_buffer, import_options) -> Value
```

A stable handle is a logical identity, not a permanent pointer. A borrowed span is a temporary capability, not a handle. A root token keeps reachable native objects alive; a copied observer cell alone need not. These distinctions are the foundation of the following chapters.

Checkpoint

Exercise 26. Choose between LiveFlex and WireLog for a read-mostly configuration document and for a rapidly growing simulation state. Include snapshot frequency, alias semantics, memory retention, and the cost of updating one element of a large vector.

25 The 16-byte cell and the 32-byte descriptor

Keep ordinary scalars inline

A generic value should not allocate merely to store an integer. The laboratory and proposed engine use a straightforward 16-byte cell:

CPP

```
struct alignas(8) Cell {
    std::uint64_t payload;
    std::uint32_t aux;
    Kind kind;           // one byte
    std::uint8_t flags;
    std::uint16_t reserved;
};
```

The implementation asserts its size, alignment, and relevant field offsets. Signed and unsigned integers occupy the full payload. A floating value stores its binary64 representation. Bool and null use canonical payloads. No width changes are needed when a signed value grows from 127 to 128 or from 32-bit to 64-bit range.

This is a native layout, not a portable serialized structure. It must not be dumped to a file and called FlexBuffers. Its purpose is to make mutation simple and predictable; compression is deferred to export or to specialized native regions.

Cell byte range	Normal use	Short-text use
0-7	Scalar payload or packed handle	First eight content bytes
8-11	Document owner identity for external handles	Next four content bytes
12	Kind	String/key kind
13	Flags	Inline flag plus length
14-15	Reserved, initialized	Reserved, initialized

Short text uses the otherwise available bytes

A string or key of at most twelve bytes can occupy the first twelve bytes directly. Its kind and length remain in the metadata bytes. No terminator is stored in the cell; access uses the recorded length. This avoids an external allocation for many short names and labels.

The constructor must not copy beyond byte 11. It initializes the cell first, writes only the permitted content range, and leaves the kind/flags intact. The companion uses memcpy into the

object representation of a trivially copyable cell, with unsigned integer storage in the overwritten fields.

Returning a view into a by-value temporary cell would dangle. The laboratory therefore returns an owning string from `text(Cell)`. An optimized API can instead return a text-slice object that owns its inline bytes or borrows an external payload under a guard. This API issue is easy to miss when the inline layout looks appealing on paper.

External values use generational handles

For an external object, the payload packs a 32-bit slot index and a 32-bit generation. The `aux` field identifies the owning document. Resolving the handle checks the owner, slot range, live state, generation, and expected kind before exposing the payload.

The index gives stable logical identity across payload growth. The generation detects a stale handle after a freed slot is reused. The owner prevents a numerically identical handle from another document being interpreted in the wrong directory. These are correctness checks, not cryptographic capabilities: a raw memory-corrupting caller is outside this abstraction's safety boundary.

On generation exhaustion, retire the slot rather than wrapping to a value that could match an ancient observer. On document-identity exhaustion, fail or move to a wider identity scheme rather than silently reusing an identity. Such limits are rare but should have a defined outcome.

The proposed descriptor

The optimized design stores external-object metadata in stable descriptor blocks. On the chosen 64-bit ABI, this proposed descriptor fits in 32 bytes:

CPP

```
struct alignas(8) Descriptor {
    std::byte* data;           // 8
    std::uint32_t size;        // 4
    std::uint32_t capacity;    // 4
    std::uint32_t generation;  // 4
    std::uint16_t kind;        // 2
    std::uint16_t flags;       // 2
    std::uint64_t layout_epoch; // 8
};
```

size and capacity are measured in elements for vectors and in bytes for text/blob payloads; the kind determines the unit. A map block has a separately defined bucket layout and occupancy policy. Allocation-class metadata can live in the allocator rather than consuming another word in every descriptor.

`layout_epoch` changes when positions or container edges relevant to cached paths change. A document-wide content revision tracks whether a previously frozen whole-document

snapshot remains current. These are different from the generation: an object can change many times without being freed.

This descriptor is a design proposal, not the layout of `mutable_lab.hpp`. The laboratory uses a slot containing a generation and a `unique_ptr` to a variant node. That simpler representation lets the semantics be tested before custom storage is introduced.

Stable descriptors, movable payloads

When a vector grows, allocate a larger payload, move or copy its live elements, then update the descriptor's pointer and capacity. Every handle still identifies the same descriptor slot. References stored in parent cells do not need to be rewritten.

A pointer or span into the old payload does become invalid. Stable identity and stable address are different promises. The API forbids growth while a conflicting borrow is active and does not expose permanent pointers as its ordinary handle type.

The extra descriptor lookup is a real cost. Direct pointers can remove one level of indirection but make relocation, stale-reference detection, and cross-language validation harder. The capstone pays the lookup for external objects while leaving scalars and short text inline and providing bulk typed borrows to amortize repeated resolution.

Account for memory honestly

A vector of N generic scalars with capacity C needs approximately $16 * C$ payload bytes plus one descriptor and allocator/directory overhead. It does not need a descriptor per scalar. A typed binary64 vector needs approximately $8 * C$ payload bytes plus the same kind of container metadata.

A long string adds an external descriptor and its allocated byte capacity. An interned map key can be stored once in the document's symbol storage and referenced by compact IDs across maps. These distinctions matter far more than reporting “16 bytes per value” as though that number included every allocation and index.

On a machine with 64-byte cache lines, four cells fit in one line when suitably placed. That is a useful illustrative packing calculation, not a universal hardware assumption or a guarantee that one lookup touches only one line.

Checkpoint

Exercise 27. Calculate the approximate payload and descriptor memory for a vector of one million integers in generic cells versus a typed 64-bit array, both with 25 percent spare capacity. State what your estimate excludes.

26 Allocation, roots, reclamation, and compaction

Separate descriptor storage from payload storage

A descriptor directory should not require moving every descriptor when it grows. One simple design allocates fixed-size blocks and keeps pointers to those blocks in a growable directory. With a design block size of 4,096 bytes and 32-byte descriptors, each block contains 128 descriptors. This is an allocator geometry choice, not an assumption about the operating system's page size.

A handle index can be split into a block index and an in-block index. The outer directory can move because callers do not retain pointers into that vector; descriptor blocks themselves stay at stable addresses. The document owner resolves the current directory for each external access.

The laboratory obtains stable node addresses through `unique_ptr` instead. Replacing it with blocks is an optimization behind the same logical handle contract, which is precisely why the handle layer is useful.

Use different allocation policies for different lifetimes

Temporary builder descriptors and transaction scratch data can use bulk-lifetime storage. Long-lived mutable payloads need reuse. Small payloads can use size classes; large arrays can use geometric growth with less aggressive rounding. A reusable standard pool allocator is a reasonable first implementation before custom slabs are justified. [14]

Do not apply independent growth rounding at several layers without accounting for the combined waste. A vector that grows geometrically and then rounds into a much larger slab class can retain far more slack than intended. Derive capacity from the actual allocated byte count, and report the resulting high-water mark.

For numeric arrays, allocate correctly aligned storage and establish the lifetime of the actual element objects. Returning `std::span<double>` from arbitrary serialized bytes is not equivalent to returning it from a native array of live doubles. The bulk API should rest on a real typed allocation, not a convenient reinterpret cast.

Allocation must precede publication

A grow operation should reserve every resource it needs before changing the visible descriptor. Allocate the destination block, construct or copy the live elements, initialize the new element, and only then replace the descriptor's data pointer, size, and capacity. If allocation or construction fails earlier, the old object remains intact.

For trivially copyable cells and numeric payloads, the copying path is simple. For a more general future object type, exception-safe move construction becomes a separate problem. The baseline avoids arbitrary user-defined objects partly to keep this operation predictable.

After publication under exclusive ownership, retire or release the old payload. If snapshots share it, physical reclamation must wait until their ownership ends. “Update the pointer and free the old block” is correct only in the no-borrow, no-snapshot-sharing case.

A copied cell is not automatically a root

The memory manager must know which objects remain reachable from the application. The laboratory makes this explicit: `collect(roots)` receives the complete set of live external roots. A cell held in a local variable but omitted from that set can become a stale observer after collection.

A production interface should make root registration harder to forget. A move-only `RootToken` can register a cell in the document's root set and unregister it on destruction. Internal cells stored in containers are traced through their parent; they do not each need a reference count. Temporary observers are valid only while some registered root or operation guard keeps the target alive.

This avoids putting an atomic ownership counter in every scalar value. It also makes collection a deliberate operation rather than an invisible consequence of every assignment. The price is a root discipline that must be documented and tested.

Mark and sweep with generations

Marking starts from registered roots. Resolve each external handle, mark its descriptor, and traverse child cells only for container kinds. An already marked descriptor is not traversed again. This handles shared subgraphs and unreachable cycles without recursive reference-count leaks.

Sweeping releases unmarked live payloads, marks their slots free, and increments the generation before reuse. A slot whose generation is exhausted is retired. Temporary work memory should be reserved or bounded before sweeping mutates the heap, so an allocation failure does not leave reclamation half-published.

The laboratory uses an explicit worklist and a mark bitmap. Its symbol strings are retained until document destruction; they are not reclaimed by `collect`. The optimized design should either retain that simple bounded-interner policy or introduce a separately tested symbol-lifetime scheme. Reusing a key ID without accounting for existing maps would corrupt lookup meaning.

Cycle policy and collection policy are different

A collector can reclaim an unreachable cycle even though FlexBuffers export cannot represent that cycle as an ordinary finite value tree/DAG under this design. Supporting cycle detection in the collector does not imply cycle support in the interchange contract.

For applications requiring every committed state to be exportable, validate acyclicity at commit or restrict ownership to a tree discipline. A tree discipline can make insertion checks cheaper but changes sharing semantics. The capstone's low-level laboratory allows staging and rejects cycles at freeze; a production API must select and expose its stronger policy deliberately.

Compaction without changing handles

Payload compaction can move live payloads and update their descriptors while preserving handle indices and generations. This is an advantage over embedding raw payload pointers throughout the document. It still requires an exclusive phase or a versioned/snapshot-aware relocation scheme because active borrows point to old addresses.

Descriptor compaction is different. Moving a descriptor to another logical slot would change its handle identity unless an additional forwarding mechanism is introduced. The baseline keeps descriptor slots stable and compacts only payload storage. Free descriptor slots are reused through generations rather than renumbered.

Compaction should be triggered by measured fragmentation and retention, not run after every edit. Its pause time, peak temporary memory, and interaction with snapshots belong in the benchmark ledger.

Checkpoint

Exercise 28. A vector handle survives payload growth but becomes invalid after collection and slot reuse. Explain exactly which metadata changes in each case. Then describe what happens to a raw span obtained before each operation.

27 Fast containers without a declared schema

Generic vectors are the baseline

A generic vector stores contiguous 16-byte cells with a size and capacity in its descriptor. Replacing an existing scalar cell does not move other elements. Appending within capacity writes one cell and increases size. When capacity is exhausted, growth allocates a larger payload and updates the stable descriptor after copying.

Geometric growth gives amortized constant-time append, not constant worst-case latency. A growth event copies the existing live array. Middle insertion and order-preserving erase move a suffix and are linear in its length. These costs should be part of the API documentation and benchmark matrix.

A stable vector handle does not make element indices stable after insertion or erasure. Nor does it make pointers to elements survive growth. Applications needing stable per-element identities should store explicit object handles or choose a different sequence structure; the baseline does not add an allocation and identity to every array element.

Homogeneity can be discovered at runtime

A vector filled with signed integers does not need a tag beside every element. The optimized design can store a homogeneous native lane such as I64, U64, F64, or byte-sized boolean storage. The descriptor carries the lane kind. The public interface can remain schemaless because the specialization is selected from observed values or an explicit runtime request.

The simplest policy is monotonic promotion during ordinary editing. Begin in a compact typed lane when the intended kind is known. Append compatible values without changing representation. On an incompatible insertion, allocate a generic cell vector, convert the existing elements, append the new value, and publish the new representation under the same handle.

Do not automatically convert signed integers to floating point merely because a float arrives. Large integers may lose exactness. Do not merge signed and unsigned domains without checking representability and the promised type semantics. Falling back to tagged cells is often the correct lossless choice.

Automatic demotion after every erase can cause representation thrashing. Make compaction or re-specialization explicit, or require sustained evidence before changing back. This is a small example of hysteresis: use different conditions to enter and leave an optimized state.

A bulk borrow resolves once

A typed numeric operation should resolve the handle, validate the lane, and obtain a bounded span once. Its inner loop then works on actual native numeric objects. The borrow contract prevents growth, reclamation, or incompatible mutation until the borrow ends.

The laboratory exposes `numbers()` for its F64 node and relies on the documented single-owner discipline. A production wrapper should track active borrows in debug builds and make unsafe escapes difficult. A const borrow and a mutable borrow have different permissions; neither is automatically valid across a concurrent writer.

Bit-packed booleans are an optional specialization for memory-sensitive sets or bulk bit operations. They make individual updates read-modify-write operations and complicate parallel access to neighboring bits. Byte booleans are a simpler baseline. Export to the standard boolean-vector representation may expand a packed native bitmap, so include that cost.

Tiny maps should not pay for a large index

A map with only a handful of fields can use a short array of key IDs and cells. Linear search over compact IDs avoids hashing/probing metadata and keeps iteration simple. The precise crossover to a hash table is a benchmark parameter, not a universal constant such as “always eight keys.”

The laboratory uses a linear vector of `(KeyId, Cell)` pairs for every map size. This is intentionally a semantic baseline, not its proposed high-performance large-map representation. It gives a clear reference for insertion, replacement, lookup, and erase.

An optimized implementation can keep separate key-ID and value arrays for dense small maps, avoiding padding between a four-byte ID and a sixteen-byte value. If it instead uses a C++ pair or struct, measure its actual `sizeof`; the sum of member sizes is not necessarily the stride.

A concrete group-probed hash-map design

For larger maps, the proposal uses three arrays in one allocated block: one control byte per bucket, one 32-bit key ID per bucket, and one cell per bucket. Each array is aligned for its own accesses. A control byte stores either a small fingerprint or an empty/deleted state.

The principle is inspired by Swiss-table metadata filtering, but the following is an independent baseline algorithm rather than a claim about Abseil's exact probe sequence. Choose a power-of-two number of fixed-width groups. Hash an interned key ID to a starting group and a seven-bit fingerprint. Search a group's control bytes in parallel or with a scalar fallback, then compare actual key IDs at candidate positions. [\[11\]](#)

The lookup rules are:

TEXT

```
for each group in the bounded probe sequence:
    candidates = occupied controls matching the fingerprint
    for each candidate bucket:
        if stored_key_id == requested_key_id:
            return its value
    if the group contains an empty bucket:
```

TEXT · continued

```
        return missing
return table-invariant failure or missing, by policy
```

Deleted buckets do not terminate a search. An insertion remembers a reusable deleted bucket but continues until it finds the existing key or a genuinely empty stopping group. Otherwise a key already farther along the probe sequence could be inserted a second time.

The descriptor stores live count in *size* and bucket count in *capacity*. A proposed 16-byte map-block header stores the tombstone count and probe-policy metadata. With a bucket count that is a multiple of a 16-byte group, controls begin at block offset 16, key IDs at $16 + M$, and cells at $16 + 5 * M$. A suitably aligned block then occupies $16 + 21 * M$ bytes before allocator rounding. The header is additional to the 32-byte descriptor.

For a first implementation, use aligned full groups and a simple bounded linear group sequence. This avoids out-of-bounds wraparound SIMD loads. More elaborate probe sequences and unaligned metadata windows are later optimizations that must retain complete coverage of the table and safe readable ranges.

Rehashing is a staged operation

Track live count and deleted count separately. Too many tombstones can lengthen probes even when live occupancy is moderate. A rehash can rebuild at the same capacity to remove tombstones or grow when live occupancy requires it. A load-factor threshold such as $7/8$ is a starting experiment, not a guaranteed optimum.

Allocate and initialize the new block first. Reinsert live entries into it. Only after success replace the descriptor's block pointer and capacity. Increment its layout epoch so cached bucket positions are invalidated. The logical map handle remains unchanged.

At occupancy fraction a , this direct-bucket design uses roughly $21 / a$ bytes of control, key-ID, and cell storage per live entry, before alignment and descriptor overhead. At $a = 7/8$, that is about 24 bytes per entry. It excludes interned key bytes and any child payloads. This is a layout calculation, not a measured resident-memory result.

A dense-values-plus-index design can improve full-map iteration, but it adds an index indirection and another set of storage trade-offs. Benchmark it when iteration dominates; do not add it merely because “dense” sounds faster.

Intern keys once, but bound the interner

The document interner maps a UTF-8, zero-free key to a stable ID. A public key handle includes document identity so an ID from one document cannot silently refer to a different name in another. Repeated hot lookups should accept that handle rather than rehash the original string every time.

Store immutable key bytes in stable storage. IDs are not reused while maps can still contain them. For untrusted arbitrary keys, choose a hash policy and resource limits that address adversarial collision and memory-growth behavior. A fast fingerprint is not a cryptographic identity.

The simplest lifetime policy retains symbols until document destruction and caps their total bytes/count. More elaborate symbol reclamation requires tracing every map's IDs or maintaining separate ownership. It should not be smuggled into the initial allocator as an untested side effect.

Strings and blobs use replacement semantics initially

The baseline treats text and blob values as values to replace, while containers have stable shared identity. Assigning a longer string to one map entry does not mutate every other entry that happened to share the old text payload. A small value may become external or vice versa without changing the parent container's identity.

An optimized setter can reuse an external payload only when uniqueness and active-borrow rules make that safe. Without such evidence, allocate the replacement, publish the new cell, and reclaim the old unreachable payload later. Copy-on-write is an ownership algorithm, not an assumption that a pointer “probably has only one user.”

Checkpoint

Exercise 29. Implement the scalar fallback for the group-probed map before adding SIMD. Test collisions, deletion in the middle of a probe chain, duplicate replacement, same-capacity rehash, growth failure, and a cached bucket invalidated by a layout-epoch change.

28 Transactions, borrows, and concurrent snapshots

Start with one mutable owner

The baseline document is single-threaded or externally protected by an exclusive lock. This removes per-cell atomic operations from ordinary mutation and makes allocation, growth, and collection easier to reason about. It does not prevent multiple threads from reading independent immutable snapshots.

A document-level write guard can establish the exclusive phase. Mutable typed borrows exist only inside a compatible guard and prevent operations that would relocate their payloads. In debug builds, active-borrow counters can catch accidental growth or collection while a borrow is live.

The laboratory documents this discipline but does not implement a complete guard type system. Its `std::span<double>` can be misused if retained after growth or destruction. The proposed production API must close that gap before claiming safe general-purpose borrowing.

Single-operation safety before transactions

Each primitive operation should first provide its stated exception guarantee. A scalar replacement can be non-allocating. A growth operation stages new storage before publishing it. A key insertion must not leave a half-interned symbol or a half-filled map bucket visible after failure.

A multi-operation transaction is a stronger requirement. It must restore all visible changes when a later operation fails, not merely keep each individual object internally valid. The laboratory does not provide that guarantee. A failed multi-step import can leave unreachable allocated objects that a later collection reclaims.

For a first transactional implementation, use an undo journal under exclusive ownership. Record an old cell before replacing it. For structural container edits, either record enough inverse information or stage a complete replacement payload while retaining the old one until commit. Reserve journal capacity before making a change whose rollback would otherwise need to allocate.

A practical commit protocol

A transaction stages allocations and records changes. Before publication, validate document-level invariants required by the application, including acyclicity when every committed root must be exportable. On success, increment the document revision and make the new root/version visible. On failure, replay undo records in reverse and release staged allocations.

Reclamation must not destroy the old payload before rollback is no longer possible. Likewise, newly interned symbols must either be rolled back safely or remain as harmless retained interned entries under a documented policy. Retaining unused immutable symbols can preserve logical rollback while weakening a strict memory-rollback guarantee.

Not every application needs transactions. A builder-style edit batch that is discarded wholesale on failure can be simpler. Add all-or-nothing editing when the application requires it, rather than imposing journaling overhead on every scalar update by default.

Publish immutable wire snapshots first

The simplest concurrent-reader path is to freeze a completed root into an owning immutable byte buffer, verify it, and publish shared ownership of that snapshot. Readers acquire a snapshot once per request or operation batch and keep it alive while using borrowed views.

Conceptually:

CPP

```
std::atomic<std::shared_ptr<const Snapshot>> published;

// Writer, after complete construction and validation:
published.store(next_snapshot, std::memory_order_release);

// Reader, retaining ownership for its complete traversal:
auto snapshot = published.load(std::memory_order_acquire);
```

This uses standard atomic shared-ownership facilities. It is not necessarily lock-free, and destruction/deallocation of replaced ownership can have costs outside the atomic update itself. Acquire ownership once per meaningful read session rather than incrementing a shared counter for every field access. [15]

The bytes reachable through the published snapshot must actually remain immutable. An atomic pointer to a structure whose contents are still being modified is not an immutable snapshot.

Why a naive seqlock is not enough in C++

A tempting design reads a version counter, reads ordinary non-atomic payload fields, then retries when the version changed. That can detect inconsistency at an algorithmic level, but it does not automatically make conflicting non-atomic accesses legal under the C++ memory model. A data race is not repaired retroactively by discarding the result. [13]

Use locks, appropriately atomic payloads, or immutable versioned objects with a valid reclamation scheme. Sixteen-byte cells are not universally atomic. Even when a target supports a wide atomic instruction, it does not solve variable-size payload lifetime and consistency by itself.

This is why the baseline deliberately avoids a universal lock-free mutable document. The complexity belongs in a narrowly specified later design, supported by concurrency tests and memory-model reasoning.

Native copy-on-write snapshots need versioned resolution

A future optimization can avoid serializing every snapshot by sharing immutable native payloads. But copying a payload is insufficient if an old snapshot still resolves its handles through the writer's mutable descriptor directory. Updating that shared descriptor would make the old snapshot see the new payload.

A correct design gives each snapshot its own immutable view of the descriptor directory. Directory pages can be shared until modified; a writer clones the affected directory page and payload before publication. A handle is then resolved through the snapshot's directory version, not through a global current directory.

Old generations and payloads remain valid in old snapshot directories until those snapshots are released. A slot reused in the current document can have a newer generation while an old snapshot still owns the previous descriptor version. Physical memory reuse must respect both lifetimes.

This is a substantial extension. It affects collection, directory ownership, interned storage, mutation journaling, and cached paths. The wire-snapshot baseline should be measured before paying for it.

Reclamation and latency

Reference-counted snapshot ownership is simple, but the last reader can inherit a large destruction cost. A controlled retirement queue can move bulk release into a chosen maintenance phase. Epoch-based reclamation can reduce per-read reference-count traffic, but it requires disciplined reader registration and protection against stalled readers retaining unbounded memory.

These are alternatives to evaluate, not features required in the first capstone implementation. The correct policy depends on snapshot size, reader duration, latency requirements, and the acceptable retained-memory bound.

Checkpoint

Exercise 30. Explain why a copy-on-write payload plus one globally mutable handle directory is not sufficient for immutable snapshots. Draw the two directory versions needed when one vector grows while an old reader is still active.

29 Freeze and thaw without changing the standard

Define semantic equivalence first

The baseline import/export contract preserves null, bool, signed versus unsigned integer values, text bytes under the chosen UTF-8 policy, blob bytes, vector order, and map associations. It does not preserve insertion order for maps because the wire map is sorted. It does not promise the original choice between direct and indirect numbers, the original slot widths, fixed versus variable vector form, or sharing identity after import.

Finite numeric values and signed zero require deliberate exactness tests. NaN payload fidelity and exact physical floating width are separate optional requirements; do not infer them from a generic numeric API or ordinary equality. A bit-exact application can require a strict representation mode or use an explicitly defined bit payload.

The deprecated typed-string form is accepted only under its documented limited compatibility interpretation. A new exporter does not generate it. A successful modern export must use only the standard active forms supported by the target profile.

Postorder emission

Freeze walks children before parents. It emits text/blob payloads and nested containers, retains their construction descriptors, then emits the containing vector or map. Finally it emits the root trailer. This is the same ordering principle used by the wire writer in Chapter 16.

For external live objects, maintain a completed-object map from handle to emitted descriptor. A second reference to the same completed object can reuse its wire location. Maintain a separate active-path set to reject cycles. Check depth, work, output-size, and temporary-memory budgets before unbounded growth.

TEXT

```
emit(value):
    if external and in completed:
        return completed[value]
    if external and in active:
        fail with cycle error
    add external value to active
    recursively emit required children
    emit this object's standard representation
    remove external value from active
    remember completed descriptor
    return descriptor
```

The completed key must identify the live object generation, not just a recyclable slot index. The laboratory uses the packed generation/index handle within one document for that purpose.

Native map order is not wire map order

A hash map is iterated according to its storage, not lexical key order. Freeze collects live key/value associations, sorts them by the actual encoded key bytes, and emits the key vector and corresponding values in that order. Sorting by interned numeric key ID would be wrong because ID assignment order is not lexical order.

For a map with K entries, the comparison sort costs roughly $O(K \log K)$ comparisons, with key-byte comparison cost included. Stable shapes or a cached lexical order can reduce repeated work when the key population is unchanged, but a cache must be invalidated on relevant structural edits.

This sorting cost is one of the explicit prices of using a hash-based live map with a sorted wire map. Batching many edits between freezes amortizes it. A workload that freezes after every tiny edit may prefer a different live representation or WireLog.

Typed arrays and width selection

A native 64-bit integer lane can export to a narrower typed vector when every value and the vector count fit the selected width. A binary64 lane can use a narrower floating representation only under the selected exactness policy. The count still participates in the wire width solver.

This is a scan of the array before or during emission. It is not free compression. A caller can choose an export policy that preserves a wide representation when avoiding repeated narrowing analysis matters more than a smaller snapshot. The output remains standard either way if the widths and values are encoded correctly.

The laboratory's F64 node exports as a typed floating vector. Other imported typed vectors may become generic live vectors and later export in a normalized form. Physical form preservation is intentionally outside its baseline contract.

Thawing should not confuse values with identities

The simplest thaw verifies the source and recursively constructs native values. It can duplicate a shared wire child rather than preserving the source's sharing identity. That preserves the logical tree value under the baseline contract, but later mutations of one native copy do not automatically affect another.

An identity-preserving importer can memoize by source buffer, target, type, and width. It must define how inline values, shared immutable text, and containers are treated. A lazy importer can retain immutable wire subtrees and materialize only edited regions, but then the live engine needs a dual traversal path and ownership of the original buffers.

Lazy import is especially attractive for large mostly-read documents. It is a later measured optimization, not a requirement for a clear first implementation. The companion performs eager import and states its normalization behavior explicitly.

Incremental export is not automatically zero-copy

A previously serialized subtree can be copied as a block only when all internal relative references, required prefixes, and alignment conditions remain valid after relocation. References to shared objects outside that block break the simple closed-block assumption. A buffer slice that looks like one subtree may not contain every byte it needs.

An ordinary FlexBuffer also cannot point into multiple unrelated allocations using native pointers. Standard export ultimately needs one coherent byte-addressed representation. Scatter/gather transport does not change what the receiving reader expects its offsets to mean.

The easiest cache is the whole immutable snapshot: when the document revision has not changed, return the same owned snapshot. Per-subtree caches require dependency tracking. A child mutation can invalidate every cached ancestor that embeds that child's serialized bytes, even when the ancestors' own field populations did not change. Shared graphs make this a real dependency problem.

Verification at the export boundary

During development, run the strict verifier on every emitted snapshot and compare it with the native reference model. In qualification, run the independent official readers as well. Once the writer is well tested, a production build may make redundant self-verification optional under an explicit trust policy.

Do not publish partially emitted bytes. A snapshot becomes visible only after its complete trailer and ownership are established. A failed freeze leaves the previous published snapshot intact. That simple publication rule is one of the strongest reasons to keep mutable storage and interchange storage separate.

Checkpoint

Exercise 31. A cached serialized map contains a shared child vector that another map also references. The child changes without either map's key set changing. Which caches are invalid, and why is a key-population version insufficient to decide?

30 Build the capstone in stages, then earn its speed

Run the supplied laboratory

The companion can be built without the official runtime:

```
SH
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build -j2
ctest --test-dir build --output-on-failure
./build/flexbook_demo
./build/flexbook_fixtures fixtures
```

The demonstration constructs a map, creates a typed numeric vector, grows it, changes a number through a native span, replaces a short string with a longer string, and freezes the result. Its recorded output is:

```
TEXT
frozen bytes: 79
name: A longer replacement
samples[0]: 42
```

That output comes from the included code, not from a hypothetical optimized engine. It establishes the essential workflow: native edits followed by a standard-layout snapshot that the teaching verifier and reader can navigate.

Read the three core files in order

`flat_table_lab.hpp` is a compact schema-specific example of front-root navigation, signed vtable displacement, optional field lookup, and forward child references. Its scope is deliberately one table.

`flex_wire.hpp` contains the complete educational wire implementation: type codes, checked arithmetic, scalar loads, UTF-8 validation, the width-solving writer, reference navigation, strict verification, re-encoding, and constrained mutation. Read the helpers before the containers. Then compare the reader and writer formulas line by line.

`mutable_lab.hpp` contains the native cell, document ownership checks, generational slots, interning, standard-container nodes, mutation operations, explicit-root collection, and freeze/thaw. It is short enough to understand as a whole before replacing any storage component.

What is implemented and what is proposed

Capability	Laboratory	Optimized design
16-byte native cells	Implemented and size-checked	Retained
Signed/unsigned/float scalar payloads	Implemented	Retained
Inline text up to 12 bytes	Implemented	Retained with stronger view API
Generational document-scoped handles	Implemented	Retained with opaque public types
Mutable maps/vectors and F64 arrays	Implemented using standard containers	Adaptive storage and grouped map index
Explicit-root mark/sweep	Implemented	RootToken interface and bounded maintenance
Standard-layout freeze/thaw	Implemented; own tests executed	Independent oracle qualification required
Descriptor blocks and reusable slabs	Not implemented	Proposed
SIMD map probes and runtime shapes	Not implemented	Proposed, benchmark-gated
Transactions and concurrent snapshots	Not implemented	Staged design in Chapter 28

This table is part of the project's contract. Do not rename the laboratory “production LiveFlex” merely because it demonstrates the same high-level workflow.

Stage one: qualify the wire layer

Complete the independent interoperability matrix before optimizing the native heap. Add official C++ tests in both directions and a second language reader. Expand the generated corpus to cover every advertised type and width combination. Add allocation-failure and byte-scan-budget tests. Keep all counterexamples as regression fixtures.

Make public construction descriptors opaque and separate verified immutable owners from raw spans. Add an iterative verifier with explicit work and byte budgets when hostile or deeply shared inputs are in scope. Test on the intended architectures and endian environments rather than assuming one x86-64 run establishes portability.

Acceptance is behavioral: valid supported input is read correctly, invalid input is rejected safely, resource limits are reported distinctly, and produced bytes are read by independent implementations. A green build is not that acceptance test.

Stage two: replace one storage component at a time

Introduce stable descriptor blocks while keeping standard payload containers. Compare all stateful tests against the laboratory. Then introduce reusable payload allocation. Then add typed lanes and promotion. Then implement the scalar group-probed map, followed by SIMD only after its probe invariants are tested.

At every step, preserve a way to run the same operation trace through the reference laboratory. A performance improvement that changes aliasing, presence, signedness, or failure behavior is not a valid optimization unless the public contract is intentionally changed.

Use benchmarks to select small-map thresholds, growth factors, slab cutoffs, and metadata group width. Keep the configuration surface small. Parameters that do not materially affect the intended workloads should not become permanent public complexity.

Stage three: add only required ownership features

Introduce root tokens and guarded borrows before general application adoption. Add transactions when multi-operation rollback is required. Add immutable wire-snapshot publication before native copy-on-write snapshots. Add epoch reclamation only if measured reference-count or destruction costs justify it.

Each new feature should have a failure story. What happens when an allocation fails halfway through a rehash? When a root is released during a pending transaction? When a reader retains a snapshot indefinitely? When a generation or version reaches its limit? When an export encounters a cycle or exceeds its budget?

The ability to answer these questions precisely is a stronger sign of implementation quality than an impressive list of data-structure techniques.

Stage four: establish the performance frontier

Publish results for defined workloads and machine configurations, including the official implementation and simple standard-container baselines. Report both speed and memory, and include freeze frequency and retained snapshots. Keep unsuccessful optimization experiments so future maintainers understand why an apparently clever design was rejected.

A good result may be conditional: LiveFlex wins on repeated structural editing and typed numerical updates, while a verified wire buffer wins on sparse reads of compact immutable data. That is not a failure of the capstone. It is the expected result of designing representations for different jobs.

The final mental model

FlatBuffers teaches how a schema can remove repeated type information while vtables preserve field discoverability and evolution. FlexBuffers teaches how a buffer can carry its own types while using widths and relative references to stay compact. LiveFlex teaches where mutability needs identity, capacity, lifetime, and synchronization that a compact wire layout does not supply.

Once those three models are clear, the design choices stop looking arbitrary. You can read the official source as a collection of mechanisms serving explicit constraints—and build your own implementation without confusing a convenient API, a valid byte format, and a fast mutable engine.

Checkpoint

Exercise 32. Choose one proposed optimization and write its acceptance gate: required invariants, reference-model comparisons, adversarial tests, benchmark workloads, and the condition under which you would remove it again.

REFERENCE & COMPLETE SOURCE

APPENDIX A

Type codes and byte equations

Complete FlexBuffers type table for this edition

The numeric codes below are from the pinned C++ source. The packed type byte is $(\text{code} \ll 2) \mid \text{width_code}$, where width codes 0, 1, 2, and 3 mean 1, 2, 4, and 8 bytes respectively. “Indirect” describes the parent slot, not an extra field stored beside every object. [\[1\]](#)

Code	Type	Parent slot	Destination or interpretation
0	Null	Inline	Canonical zero in the strict profile
1	Int	Inline	Signed integer at parent width
2	UInt	Inline	Unsigned integer at parent width
3	Float	Inline	Binary32 or binary64 payload
4	Key	Offset	Zero-terminated key bytes
5	String	Offset	Content; length prefix before target
6	IndirectInt	Offset	Signed scalar at child width
7	IndirectUInt	Offset	Unsigned scalar at child width
8	IndirectFloat	Offset	Floating scalar at child width
9	Map	Offset	Value vector with three-field prefix
10	Vector	Offset	Untyped vector with type table
11	VectorInt	Offset	Variable typed signed vector
12	VectorUInt	Offset	Variable typed unsigned vector
13	VectorFloat	Offset	Variable typed floating vector
14	VectorKey	Offset	Variable typed key vector
15	VectorStringDeprecated	Offset	Legacy form; do not emit in new code
16	VectorInt2	Offset	Fixed signed pair
17	VectorUInt2	Offset	Fixed unsigned pair
18	VectorFloat2	Offset	Fixed floating pair

Code	Type	Parent slot	Destination or interpretation
19	VectorInt3	Offset	Fixed signed triple
20	VectorUInt3	Offset	Fixed unsigned triple
21	VectorFloat3	Offset	Fixed floating triple
22	VectorInt4	Offset	Fixed signed quadruple
23	VectorUInt4	Offset	Fixed unsigned quadruple
24	VectorFloat4	Offset	Fixed floating quadruple
25	Blob	Offset	Raw bytes; length prefix, no terminator
26	Bool	Inline	Canonical 0 or 1 in strict profile
27–35	Unassigned in this snapshot	—	Reject in this implementation
36	VectorBool	Offset	Variable typed boolean vector
37–63	Not supported by this snapshot	—	Reject rather than guess

A type-number range check is not enough to establish support when there are gaps. A future assignment of a previously unused code should cause an explicit capability decision, not accidental interpretation by arithmetic intended for an older family.

The equations worth memorizing

Operation	Equation
Bounded range	$p \leq N \ \&\& \ n \leq N - p$
Alignment padding	$(W - (p \ \& \ (W - 1))) \ \& \ (W - 1)$
FlatBuffers ordinary reference	$\text{target} = \text{slot} + \text{U32}(\text{slot})$
FlatBuffers vtable	$\text{vtable} = \text{table} - \text{S32}(\text{table})$
FlatBuffers field entry	$\text{vtable} + 4 + 2 * \text{field_id}$
FlexBuffers root slot	$N - 2 - \text{bytes}[N - 1]$
FlexBuffers reference	$\text{target} = \text{slot} - \text{U}(\text{slot}, \text{parent_width})$
FlexBuffers length prefix	$\text{target} - \text{child_width}$
FlexBuffers vector element	$V + i * W$
Untyped vector element tag	$V + N * W + i$
Map key-vector offset slot	$V - 3 * W$
Map stored key width	$\text{U}(V - 2 * W, W)$
Fixed numeric tuple length	$2 + (\text{type_code} - 16) / 3$

Every equation involving untrusted values requires the relevant checked arithmetic and bounds conditions before pointer formation. The notation is a compact memory aid, not a complete verifier.

Small golden buffers

Value	Complete hexadecimal bytes
Signed 13	0d 04 01
True	01 68 01
Null	00 00 01
String Ada	03 41 64 61 00 04 14 01
Empty string	00 00 01 14 01
Empty blob	00 00 64 01
Empty untyped vector	00 00 28 01
Untyped [1,2,3]	03 01 02 03 04 04 04 06 28 01
Typed signed [1,2,3]	03 01 02 03 03 2c 01
Fixed signed triple	01 02 03 03 4c 01

These buffers are generated by the companion and checked by its tests. Their validity is grounded in the described wire rules; executed independent-runtime confirmation remains a separate qualification step.

REFERENCE & COMPLETE SOURCE

APPENDIX B

Exercise solutions and implementation hints

Solutions 1–4: foundations

1. The service avoided constructing a conventional object tree merely to read ten fields. It still acquired bytes, copied them into ownership, verified them, and performed access work. Measure acquisition/copy, verification, the ten-field traversal, and any later materialization separately; also measure their sum in the real request path. Retain the verified owner during traversal.
2. 80 is unsigned 128 or signed -128 at one byte. 80 00 is signed positive 128 at two bytes. The sign bit is bit 7 in the first signed interpretation and bit 15 in the second. Width changes the interpretation, even though the least significant byte is the same.
3. First establish $\text{position} \leq 100$. Then $100 - \text{position}$ is a valid remaining extent and comparing the requested length with it cannot overflow. Adding an arbitrary large length to position first can wrap and make an invalid range appear small.
4. One reasonable design makes `VerifiedBuffer` own an immutable allocation. Its views borrow it and cannot outlive it. Moving the owner can preserve views only if the allocation itself stays stable and the contract says so. For independently long-lived views, retain shared ownership explicitly. Do not expose mutable bytes through the verified owner without invalidating its proof.

Solutions 5–10: FlatBuffers

5. The vtable is at $64 - (-16) = 80$. The field is at $64 + 12 = 76$. Both displacements are interpreted relative to the table, but one locates the vtable by signed subtraction and the other locates a field by adding the vtable entry's unsigned displacement.
6. A same-length replacement changes only the three content bytes when the allocation is writable and alias/concurrency rules permit it. A longer replacement needs enough content space, a correct length and terminator, and valid references from every affected slot. Rebuilding is the straightforward general solution. Shared string references make in-place changes visible to every alias.
7. Use a root table for the stable header and optional fields. Use a fixed-layout point struct inside a vector for dense numeric points when its schema is stable. Store dynamic metadata as an explicitly designated FlexBuffers byte payload. Verify the outer table and vector, then verify the inner FlexBuffer under its own profile; retain the outer allocation while borrowing inner bytes.

8. For an eight-byte offset slot the construction formula is $S + 8 - D$, after alignment. D is measured from the end during construction. The emitted distance is measured forward from the final slot. They are different coordinate systems, related by the formula rather than interchangeable values.
9. A new reader sees 5 in old buffers where the field was absent. To preserve old meaning, keep the old default and add a new field or explicit versioned interpretation, or migrate stored data deliberately. Do not assume absence contains a historical copy of the old default value.
10. Extra bytes at the end do not create a field displacement in the table's vtable or a suitable inline slot in its body. Updating shared vtable metadata can also affect other tables. A general structural edit must create a coherent table layout and references, not merely find unused allocation capacity.

Solutions 11–15: FlexBuffers primitives and vectors

11. The final byte says the root slot is one byte. The preceding 0×68 gives type 26 and child width code 0. The slot contains 1, so the value is true. A packed width code of 3 means $1 \ll 3 = 8$; the final trailer instead stores the actual width, where 3 is not one of the permitted values.
12. Root offset: one byte. Vector count: two bytes. String offset: two bytes because it is a vector element slot. String length: one byte because the string tag describes a one-byte prefix. These four answers are the simplest diagnostic for whether the two-width model has been understood.
13. A 256-element variable typed boolean vector needs a two-byte count, so its common element width is at least two. Its slots alone occupy 512 bytes. Under the companion's straightforward layout, the complete document is 518 bytes: two count bytes, 512 element bytes, a two-byte root offset, and two trailer bytes. It is not a bitset.
14. A 300-byte blob has a two-byte count and exactly 300 payload bytes. A 300-element typed unsigned vector of byte-sized values uses two-byte slots because its count is 300, producing 600 payload bytes. In the straightforward complete layouts, the totals are 306 and 606 bytes respectively. The blob's width describes its prefix, not its content stride.
15. The untyped triple is ten bytes. Removing its three element tags produces a seven-byte typed vector. Encoding the length in a fixed tuple type removes one count byte, producing six bytes. The existing typed-vector type family is a defined set of wire codes, not a generic formula that can be extended without coordinating a new protocol assignment and reader support.

Solutions 16–20: maps, builders, mutation, verification

16. Read root width at 20 and map tag at 19. Read the root offset at 18 to reach value data at 14. Recover value count at 13, key-vector width at 12, and its offset at 11 to reach key data at 9. Read key count at 8. The foo key offset at 10 reaches position 4. Its matching index is 1, so read value slot 15 and tag 17. All widths happen to be one in this fixture; that coincidence must not become an implementation assumption.

17. `get_signed()` should reject all non-signed types and distinguish absence from a present null. `convert_to_signed()` can accept an unsigned integer only when it fits, and a float only under explicit integral/range/rounding rules. A missing key remains absence, not zero. A null conversion should be rejected unless a separate application policy intentionally supplies a default.

18. Restore byte length, temporary value/field stacks, open-container state, finished state, and every pool/index entry referring to newly written objects. Restore duplicate/error tracking according to its contract. A simpler implementation stages pool changes until commit or discards the entire failed builder operation scope.

19. Inline: 8 count bytes + 800 slots + 100 tags = 908 bytes before the root. The root distance is 900, requiring two bytes; with its type and width trailer the total is 912. Indirect: 8 scalar bytes + 1 count + 100 slots + 100 tags = 209; the root distance is 200, requiring one byte, giving 212. For a counterexample, make one tiny integer in [1, 2] indirect: the extra scalar payload adds a byte without reducing the already one-byte vector stride.

20. A work item can carry slot, parent width, child width, type, depth, and a continuation state. A memo key for an indirect interpretation includes buffer identity, target, type, and child width. Store “active” separately from “complete.” Charge every visit and every newly scanned text extent against explicit budgets; reuse completed validation where the interpretation and immutable ownership match.

Solutions 21–25: engineering

21. Include signed extremes, unsigned extremes, direct and indirect floats, embedded-zero strings, empty blobs, every typed family, fixed tuples, different root/child widths, and different map key/value widths. A writer and reader that both reverse an offset base or both mistake width codes for actual widths can agree with each other while failing an independent reader.

22. A generic-cell scan touches a tag and payload per cell, typically sixteen bytes of cell storage before external payloads. A native typed array can resolve type and ownership once and read fixed-stride numbers. A verified typed wire vector can resolve type/width once, then use specialized endian-aware loads. Hoist container type, width dispatch, count, and stable bounds outside the element loop where the contract permits it.

23. Return full-range integers as `BigInt`, or use a checked exact-number conversion. Return blobs through a byte-oriented type. Return strings using an explicit byte length and a specified

UTF-8 decoder; embedded zeros are not terminators in the logical string. A plain JSON round-trip is not a lossless bridge for every supported value and type distinction.

24. Immediate publication of every edit makes per-edit rebuilding a relevant baseline. An editor that applies a thousand changes before sending one snapshot should compare one freeze per batch, not a thousand freezes. Conversion overhead can dominate the first workload while native update savings dominate the second.

25. Add `intern_key(text) -> Key` and overload `hot lookup` to accept `Key`. The handle carries document identity and a stable key ID. Validate the identity at the boundary; do not accept an unqualified integer that could mean a different string in another document. Arbitrary-string lookup remains available for cold paths.

Solutions 26–32: the capstone

26. A read-mostly configuration may remain a verified wire buffer or use `WireLog` for occasional replacement snapshots. A rapidly growing working state is a stronger `LiveFlex` candidate. Large vector edits expose rebuild cost; frequent snapshots can erase native-editing gains. Shared-identity mutation and persistent path replacement have different alias behavior, so choose the semantics before comparing speed.

27. With capacity 1,250,000, generic cells need 20,000,000 payload bytes and a typed 64-bit lane needs 10,000,000. Each has a container descriptor, plus directory, allocator, root, and alignment overhead. The estimate excludes child allocations, snapshots, and unused allocator-retained blocks.

28. Growth changes the payload pointer/capacity and relevant layout version but not slot index or generation. Collection releases an unreachable payload and increments its generation before slot reuse. A span into the old payload is invalid after either relocation or reclamation. The logical handle survives only the first operation.

29. Establish scalar probe correctness against a standard map. In particular, a tombstone cannot terminate lookup, and insertion must search for an existing key before reusing a tombstone. Stage rehash into a new block and publish only after success. A layout epoch invalidates cached bucket indices after growth or rearrangement.

30. An old snapshot resolving through the current mutable directory would follow the new payload pointer and observe the update. Give the old and new snapshots distinct immutable directory views, sharing unchanged pages and payloads. The writer clones the affected directory page and payload; ownership retains the old version until its readers finish.

31. Every cached serialized ancestor that embeds the old child representation is invalid, including ancestors reachable through different aliases. An unchanged key set says only that map membership is stable, not that descendants are unchanged. Track content dependencies or rebuild/validate the relevant cache from descendant versions.

32. A valid gate names the behavior that must remain identical, tests adversarial and failure cases, measures the target workload and memory cost, and states a removal criterion. For example, retain SIMD probing only when it improves the intended map-size distribution enough to justify its extra platform code while matching the scalar path on the full stateful corpus.

[REFERENCE & COMPLETE SOURCE](#)[APPENDIX C](#)

Reading the official implementation

A deliberate reading order

Start with the enum and width helpers in `flexbuffers.h`, not with the builder's convenience overloads. Draw the relationship between a type code, its packed byte, and a stored byte width. Then read `Object`, `Sized`, `String`, and `Blob`. Their small interfaces establish the coordinates used everywhere else. [\[1\]](#)

Read `Reference` next. Mark each operation that consumes `parent_width_` and each operation that consumes `byte_width_`. Follow one inline integer, one indirect integer, and one string through their accessors. Notice the difference between an exact physical type test and a permissive conversion. Then trace `GetRoot`, the vector index operators, and map lookup. [\[1\]](#)

For writing, begin with `Builder::Value`, `ElemWidth`, `StoredWidth`, and `StoredPackedType`. Continue through `Align`, `WriteOffset`, `WriteAny`, `CreateVector`, `EndMap`, and `Finish`. The public `Int`, `String`, and lambda-based convenience functions are easier to understand after these mechanisms are clear. Keep a drawing of the growing byte array and the separate temporary value stack. [\[1\]](#)

Read the verifier last, but do not regard it as an optional afterthought. Compare each accessor assumption with the check that is supposed to establish it. In particular, distinguish a bounds check from a semantic check. The companion adds a stricter application profile; that profile is not a claim that every upstream implementation applies identical acceptance rules. [\[1\]](#)

For `FlatBuffers`, use `Table::GetVTable`, `GetOptionalFieldOffset`, `GetField`, and `GetPointer` to reconstruct the read path. Then trace `PushElement`, `ReferTo`, `TrackField`, and `EndTable` in the builder. The public format documentation is a useful guide, but the equations become concrete when followed through an actual field write and subsequent read. [\[2–4\]](#)

Questions to ask beside every function

Identify its preconditions: which bytes are already in bounds, which widths are legal, who owns the memory, and whether concurrent mutation is excluded. Identify its coordinate system: absolute position, end-relative construction offset, parent-relative wire displacement, or native handle. Identify its failure contract: exception, false return, assertion, default value, or undefined use outside the API contract.

Finally, identify its cost. Does it read only metadata, scan a string, allocate a temporary container, sort keys, perform a dependent load, or materialize a result? A function's short source code is not a reliable measure of its runtime work.

Concept navigator

Concept	Where to return
Alignment, endian conversion, object lifetime	Chapters 2–3
Forward references and signed vtable displacement	Chapters 4–5
Omitted fields, defaults, schema evolution	Chapters 4, 8
Backward construction and end-relative offsets	Chapter 7
Root trailer and the two-width rule	Chapters 10–11
Keys, strings, blobs, and embedded zero bytes	Chapter 12
Typed and fixed vectors	Chapter 13
Sorted maps and independent key-vector width	Chapter 14
Reader views and strict conversions	Chapter 15
Candidate-width solving and writer ownership	Chapter 16
Aliasing and constrained wire mutation	Chapter 17
Structural validation and application profiles	Chapter 18
Test gates, sanitizer evidence, interoperability	Chapter 19
SIMD, cache locality, batching, allocation	Chapters 20–23
Wire compatibility versus mutable layout	Chapter 24
Cells, descriptors, handles, generations	Chapters 25–26
Typed lanes, map probing, growth	Chapter 27
Borrows, rollback, snapshots, reclamation	Chapter 28

Concept	Where to return
Standard export, import, and invalidation	Chapter 29
Implementation stages and acceptance gates	Chapter 30

Small glossary

Cell: a fixed-size native value record. In LiveFlex, a cell contains an inline scalar or a handle to a separately managed object. It is not a FlexBuffers wire slot.

Descriptor: a stable native record describing a variable-size object and locating its current payload. Moving a payload need not change the object's logical identity.

Generation: a counter associated with a handle slot, used to distinguish a new occupant from a stale reference to a previous occupant. Generation exhaustion requires an explicit policy.

Packed type: the FlexBuffers byte combining a six-bit type code and a two-bit width code. Its width is not necessarily the width of the parent slot containing the reference.

Parent width: the number of bytes occupied by the current slot. It determines how to read an inline scalar or the offset leading to an indirect object.

Child width: the width associated with an indirect destination, such as its length prefix or vector elements. The book uses this name to make the distinction memorable.

Vtable: the FlatBuffers metadata object that maps stable schema field positions to byte displacements within a table instance. It is not a C++ virtual-function table.

Verification: establishing a stated set of structural or semantic conditions for specified bytes. It does not automatically establish ownership, immutability, authenticity, or application-level meaning.

Freeze: export from a mutable native representation into a complete standard FlexBuffer. It includes representation choices and may require sorting, allocation, traversal, and copying.

Thaw: import into a mutable representation. Eager thaw materializes values; a lazy design may retain verified immutable data and materialize selected portions later.

Snapshot: a stable view of an identified version. Publishing a root atomically does not by itself make every object reachable from that root immutable or safe to reclaim.

Source listings and reproducibility

The next three appendices contain the complete core headers used by the companion. Line numbers are reading aids, not part of the source. Long lines may wrap visually; build from the files in the ZIP rather than copying typeset pages. The tests, fixture generator, demonstration, optional interoperability adapters, and exact execution receipts are included in that ZIP.

The code intentionally favors an auditable semantic baseline over a finished optimized runtime. Its limitations are part of the contract described in Chapters 19 and 30. An implementation can be valuable for learning and differential testing without being ready to accept hostile production traffic.

REFERENCE & COMPLETE SOURCE

APPENDIX D

Complete checked FlexBuffers laboratory

File: `include/flex_wire.hpp`. This is original educational C++20 code, not a copy of the official header. It supplies the byte primitives, writer, borrowed reader, strict-profile verifier, copy operation, and constrained mutation helpers discussed in the book.

```
include/flex_wire.hpp

1 // Original educational implementation for the accompanying book.
2 // SPDX-License-Identifier: MIT
3 // C++20. No FlatBuffers runtime dependency. Not production-certified.
4 #pragma once
5 #include <algorithm>
6 #include <array>
7 #include <bit>
8 #include <cmath>
9 #include <cstdint>
10 #include <cstdint>
11 #include <cstring>
12 #include <limits>
13 #include <optional>
14 #include <span>
15 #include <stdexcept>
16 #include <string>
17 #include <string_view>
18 #include <utility>
19 #include <vector>
20
21 namespace flexlab {
22 using Bytes = std::vector<std::uint8_t>;
23 using ByteSpan = std::span<const std::uint8_t>;
24 struct Error : std::runtime_error { using std::runtime_error::runtime_error; };
25 inline void require(bool ok, const char* what) { if (!ok) throw Error(what); }
26 static_assert(sizeof(double) == 8 && sizeof(float) == 4);
27 static_assert(std::numeric_limits<double>::is_iec559);
28 static_assert(std::numeric_limits<float>::is_iec559);
29
30 enum class Type : std::uint8_t {
31     Null=0, Int=1, UInt=2, Float=3, Key=4, String=5,
32     IndirectInt=6, IndirectUInt=7, IndirectFloat=8, Map=9, Vector=10,
33     VectorInt=11, VectorUInt=12, VectorFloat=13, VectorKey=14,
34     VectorStringDeprecated=15,
35     Int2=16, UInt2=17, Float2=18, Int3=19, UInt3=20, Float3=21,
36     Int4=22, UInt4=23, Float4=24, Blob=25, Bool=26, VectorBool=36
37 };
38 inline unsigned id(Type t) { return static_cast<unsigned>(t); }
39 inline bool inline_type(Type t) { return id(t) <= 3 || t == Type::Bool; }
40 inline bool typed(Type t) { return (id(t)>=11 && id(t)<=15) || t==Type::VectorBool; }
41 inline bool fixed(Type t) { return id(t)>=16 && id(t)<=24; }
42 inline bool valid_type(Type t) { return id(t)<=26 || t==Type::VectorBool; }
43 inline bool width_ok(unsigned w) { return w==1 || w==2 || w==4 || w==8; }
44 inline unsigned width_code(unsigned w) {
45     require(width_ok(w), "invalid byte width");
46     return w==1 ? 0 : w==2 ? 1 : w==4 ? 2 : 3;
47 }
```

include/flex_wire.hpp · continued

```

47 }
48 inline std::uint8_t packed(Type t, unsigned w) {
49     return static_cast<std::uint8_t>((id(t)<<2) | width_code(w));
50 }
51 inline unsigned uint_width(std::uint64_t x) {
52     return x<=0xff ? 1 : x<=0xffff ? 2 : x<=0xffffffffULL ? 4 : 8;
53 }
54 inline unsigned int_width(std::int64_t x) {
55     if (x>=-128 && x<=127) return 1;
56     if (x>=-32768 && x<=32767) return 2;
57     if (x>=INT32_MIN && x<=INT32_MAX) return 4;
58     return 8;
59 }
60 inline bool fits_unsigned(std::uint64_t x, unsigned w) {
61     return w==8 || x < (std::uint64_t{1} << (8*w));
62 }
63 inline std::size_t add(std::size_t a, std::size_t b) {
64     require(b<=SIZE_MAX-a, "size addition overflow"); return a+b;
65 }
66 inline std::size_t mul(std::size_t a, std::size_t b) {
67     require(a==0 || b<=SIZE_MAX/a, "size multiplication overflow"); return a*b;
68 }
69 inline std::size_t align_up(std::size_t n, unsigned w) {
70     require(width_ok(w), "invalid alignment");
71     return add(n, (w-(n & (w-1))) & (w-1));
72 }
73 inline void range(ByteSpan b, std::size_t p, std::size_t n) {
74     require(p<=b.size() && n<=b.size()-p, "byte range out of bounds");
75 }
76 // This slow, explicit baseline never creates an unaligned typed pointer.
77 inline std::uint64_t load_u(ByteSpan b, std::size_t p, unsigned w) {
78     require(width_ok(w), "invalid load width"); range(b,p,w);
79     std::uint64_t u=0;
80     for (unsigned j=0; j<w; ++j) u |= std::uint64_t{b[p+j]} << (8*j);
81     return u;
82 }
83 inline std::int64_t load_i(ByteSpan b, std::size_t p, unsigned w) {
84     auto u=load_u(b,p,w);
85     if (w==8) return std::bit_cast<std::int64_t>(u);
86     const unsigned bits=8*w;
87     if (u & (std::uint64_t{1} << (bits-1)))
88         return static_cast<std::int64_t>(u) - static_cast<std::int64_t>(std::uint64_t{1}<<bit
• s);
89     return static_cast<std::int64_t>(u);
90 }
91 inline double load_f(ByteSpan b, std::size_t p, unsigned w) {
92     require(w==4 || w==8, "float must be binary32 or binary64");
93     if (w==4) return std::bit_cast<float>(static_cast<std::uint32_t>(load_u(b,p,4)));
94     return std::bit_cast<double>(load_u(b,p,8));
95 }
96 inline void store_u(Bytes& b, std::uint64_t u, unsigned w) {
97     require(width_ok(w), "invalid store width");
98     for (unsigned j=0; j<w; ++j) b.push_back(static_cast<std::uint8_t>(u>>(8*j)));
99 }
100 inline bool utf8(std::string_view s) {
101     for (std::size_t i=0; i<s.size(); i++) {
102         const auto a=static_cast<unsigned char>(s[i++]);
103         if (a<0x80) continue;

```

include/flex_wire.hpp · continued

```

104     unsigned more=0; std::uint32_t cp=0, minimum=0;
105     if (a>=0xc2 && a<=0xdf) { more=1; cp=a&31; minimum=0x80; }
106     else if (a>=0xe0 && a<=0xef) { more=2; cp=a&15; minimum=0x800; }
107     else if (a>=0xf0 && a<=0xf4) { more=3; cp=a&7; minimum=0x10000; }
108     else return false;
109     if (more>s.size()-i) return false;
110     while (more--) {
111         auto q=static_cast<unsigned char>(s[i++]);
112         if ((q&0xc0)!=0x80) return false;
113         cp=(cp<<6)|(q&63);
114     }
115     if (cp<minimum || cp>0x10ffff || (cp>=0xd800 && cp<=0xdfff)) return false;
116 }
117 return true;
118 }
119 inline int byte_compare(std::string_view a, std::string_view b) {
120     const auto n=std::min(a.size(),b.size());
121     const auto c=n ? std::memcmp(a.data(),b.data(),n) : 0;
122     if (c) return c;
123     return a.size()<b.size() ? -1 : a.size()>b.size() ? 1 : 0;
124 }
125
126 struct Encoded {
127     Type type=Type::Null;
128     unsigned width=1; // scalar minimum or child storage width
129     std::uint64_t payload=0; // scalar bits, or absolute child position
130     const void* owner=nullptr; // referenced children belong to one writer
131 };
132
133 class Writer {
134     Bytes b_;
135     bool sealed_=false;
136     void writable() const { require(!sealed_, "writer already finished"); }
137     void owned(const Encoded& e) const {
138         require(inline_type(e.type) || e.owner==this, "cross-writer child reference");
139         require(valid_type(e.type) && width_ok(e.width), "invalid encoded value");
140     }
141     void pad(unsigned w) { b_.resize(align_up(b_.size(),w),0); }
142     void write_value(const Encoded& e, unsigned w) {
143         owned(e);
144         if (!inline_type(e.type)) {
145             require(e.payload<=b_.size(), "forward FlexBuffers offset");
146             const auto d=static_cast<std::uint64_t>(b_.size())-e.payload;
147             require(fits_unsigned(d,w), "offset does not fit"); store_u(b_,d,w);
148         } else if (e.type==Type::Float) {
149             const auto d=std::bit_cast<double>(e.payload);
150             require(w==4 || w==8, "invalid float slot width");
151             store_u(b_, w==4 ? std::bit_cast<std::uint32_t>(static_cast<float>(d)) :
152 • e.payload,w);
153         } else store_u(b_,e.payload,w);
154     }
155     unsigned vector_width(std::span<const Encoded> v, unsigned prefix,
156                           bool has_length, const Encoded* keys) const {
157         for (unsigned w : {1U,2U,4U,8U}) {
158             if (has_length && !fits_unsigned(v.size(),w)) continue;
159             bool ok=true;
160             const auto base=align_up(b_.size(),w);
161             if (keys) {

```

include/flex_wire.hpp · continued

```

161         owned(*keys);
162         ok=keys->payload<=base && fits_unsigned(base-keys->payload,w)
163         && fits_unsigned(keys->width,w);
164     }
165     for (std::size_t i=0; ok && i<v.size(); ++i) {
166         const auto& e=v[i]; owned(e);
167         if (inline_type(e.type)) ok=e.width<=w;
168         else {
169             const auto p=add(base,mul(add(prefix,i),w));
170             ok=e.payload<=p && fits_unsigned(p-e.payload,w);
171         }
172     }
173     if (ok) return w;
174 }
175 throw Error("no representable vector width");
176 }
177 Encoded sequence(std::span<const Encoded> v, Type output,
178                 bool has_types, bool has_length, const Encoded* keys=nullptr) {
179     writable();
180     const unsigned prefix=keys ? 3 : has_length ? 1 : 0;
181     const auto w=vector_width(v,prefix,has_length,keys);
182     pad(w);
183     if (keys) { write_value(*keys,w); store_u(b_,keys->width,w); }
184     if (has_length) store_u(b_,v.size(),w);
185     const auto p=b_.size();
186     for (const auto& e:v) write_value(e,w);
187     if (has_types) for (const auto& e:v)
188         b_.push_back(packed(e.type,inline_type(e.type) ? w : e.width));
189     return {output,w,p,this};
190 }
191 public:
192     explicit Writer(std::size_t reserve=256) { b_.reserve(reserve); }
193     Writer(const Writer&)=delete;
194     Writer& operator=(const Writer&)=delete;
195     Writer(Writer&&)=delete;
196     Writer& operator=(Writer&&)=delete;
197     static Encoded null() { return {}; }
198     static Encoded boolean(bool b) { return {Type::Bool,1,b ? 1ULL : 0ULL,nullptr}; }
199     static Encoded integer(std::int64_t x) {
200         return {Type::Int,int_width(x),std::bit_cast<std::uint64_t>(x),nullptr};
201     }
202     static Encoded uint(std::uint64_t x) { return {Type::UInt,uint_width(x),x,nullptr}; }
203     static Encoded real(double x, bool force64=false) {
204         const auto f=static_cast<float>(x);
205         const bool exact=!std::isnan(x) && static_cast<double>(f)==x;
206         return {Type::Float,!force64 && exact ? 4U :
207             • 8U,std::bit_cast<std::uint64_t>(x),nullptr};
208     }
209     Encoded indirect(Encoded scalar) {
210         writable(); owned(scalar);
211         require(scalar.type==Type::Int || scalar.type==Type::UInt ||
212             • scalar.type==Type::Float,
213             "only numeric scalars can be indirect");
214         pad(scalar.width); const auto p=b_.size(); write_value(scalar,scalar.width);
215         return {static_cast<Type>(id(scalar.type)+5),scalar.width,p,this};
216     }
217     Encoded key(std::string_view s) {
218         writable(); require(s.find('\0')==s.npos, "key contains NUL");

```

include/flex_wire.hpp · continued

```

217     require(utf8(s), "key is not UTF-8"); const auto p=b_.size();
218     b_.insert(b_.end(),s.begin(),s.end()); b_.push_back(0);
219     return {Type::Key,1,p,this};
220 }
221 Encoded string(std::string_view s) {
222     writable(); require(utf8(s), "string is not UTF-8");
223     const auto w=uint_width(s.size()); pad(w); store_u(b_,s.size(),w);
224     const auto p=b_.size(); b_.insert(b_.end(),s.begin(),s.end()); b_.push_back(0);
225     return {Type::String,w,p,this};
226 }
227 Encoded blob(ByteSpan data) {
228     writable(); const auto w=uint_width(data.size()); pad(w); store_u(b_,data.size(),w);
229     const auto p=b_.size(); b_.insert(b_.end(),data.begin(),data.end());
230     return {Type::Blob,w,p,this};
231 }
232 Encoded vector(std::span<const Encoded> v) { return sequence(v,Type::Vector,true,true);
233 }
234 Encoded typed_vector(std::span<const Encoded> v, Type element, unsigned fixed_n=0) {
235     require(element==Type::Int || element==Type::UInt || element==Type::Float ||
236         element==Type::Key || element==Type::Bool, "unsupported typed vector
237     element");
238     for (const auto& e:v) require(e.type==element, "heterogeneous typed vector");
239     Type result;
240     if (fixed_n) {
241         require(fixed_n>=2 && fixed_n<=4 && v.size()==fixed_n, "bad fixed vector length");
242         require(id(element)>=1 && id(element)<=3, "bad fixed vector element type");
243         result=static_cast<Type>(16+3*(fixed_n-2)+id(element)-1);
244     } else result=element==Type::Bool ? Type::VectorBool :
245     static_cast<Type>(id(element)+10);
246     return sequence(v,result,false,fixed_n==0);
247 }
248 Encoded map(std::vector<std::pair<std::string,Encoded>> fields) {
249     writable();
250     std::sort(fields.begin(),fields.end(),[](const auto& a,const auto& b) {
251         return byte_compare(a.first,b.first)<0;
252     });
253     std::vector<Encoded> keys, vals; keys.reserve(fields.size());
254     vals.reserve(fields.size());
255     for (std::size_t i=0; i<fields.size(); ++i) {
256         if (i) require(fields[i-1].first!=fields[i].first, "duplicate map key");
257         owned(fields[i].second); keys.push_back(key(fields[i].first));
258     }
259     vals.push_back(fields[i].second);
260     const auto kv=typed_vector(keys,Type::Key);
261     return sequence(vals,Type::Map,true,true,&kv);
262 }
263 const Bytes& finish(const Encoded& root) {
264     writable(); owned(root);
265     unsigned w=1;
266     for (w=1; w<=8; w*=2) {
267         const auto p=align_up(b_.size(),w);
268         if (inline_type(root.type) ? root.width<=w :
269             root.payload<=p && fits_unsigned(p-root.payload,w)) break;
270     }
271     require(w<=8,"root offset overflow"); pad(w); write_value(root,w);
272     b_.push_back(packed(root.type,inline_type(root.type) ? w : root.width));
273     b_.push_back(static_cast<std::uint8_t>(w)); sealed_=true; return b_;
274 }

```

include/flex_wire.hpp · continued

```

270     const Bytes& buffer() const { require(sealed_,"unfinished buffer"); return b_; }
271 };
272
273 struct Ref {
274     ByteSpan bytes;
275     std::size_t slot=0;
276     unsigned parent_width=1;
277     unsigned child_width=1;
278     Type type=Type::Null;
279     static Ref root(ByteSpan b) {
280         require(b.size()>=3,"truncated root trailer");
281         const auto w=b.back(); require(width_ok(w),"invalid root width");
282         require(w<=b.size()-2,"truncated root slot");
283         const auto t=b[b.size()-2];
284         return {b,b.size()-2-w,w,1U<<(t&3),static_cast<Type>(t>>2)};
285     }
286     std::size_t target() const {
287         require(!inline_type(type),"inline value has no target");
288         const auto d=load_u(bytes,slot,parent_width);
289         require(d<=slot,"backward offset underflow"); return
    • slot-static_cast<std::size_t>(d);
290     }
291     std::int64_t as_int() const {
292         require(type==Type::Int || type==Type::IndirectInt,"not a signed integer");
293         return type==Type::Int ? load_i(bytes,slot,parent_width) :
    • load_i(bytes,target(),child_width);
294     }
295     std::uint64_t as_uint() const {
296         require(type==Type::UInt || type==Type::IndirectUInt,"not an unsigned integer");
297         return type==Type::UInt ? load_u(bytes,slot,parent_width) :
    • load_u(bytes,target(),child_width);
298     }
299     double as_float() const {
300         require(type==Type::Float || type==Type::IndirectFloat,"not a floating-point value");
301         return type==Type::Float ? load_f(bytes,slot,parent_width) :
    • load_f(bytes,target(),child_width);
302     }
303     bool as_bool() const {
304         require(type==Type::Bool,"not a boolean"); return load_u(bytes,slot,parent_width)!=0;
305     }
306     std::size_t length() const {
307         if (fixed(type)) return 2+(id(type)-16)/3;
308         require(type==Type::Vector || type==Type::Map || typed(type) ||
309             type==Type::String || type==Type::Blob,"type has no length prefix");
310         const auto p=target(); require(child_width<=p,"missing length prefix");
311         const auto n=load_u(bytes,p-child_width,child_width);
312         require(n<=SIZE_MAX,"length does not fit size_t"); return
    • static_cast<std::size_t>(n);
313     }
314     std::string_view text() const {
315         require(type==Type::String || type==Type::Key,"not text");
316         const auto p=target();
317         if (type==Type::String) {
318             const auto n=length(); range(bytes,p,add(n,1));
319             require(bytes[p+n]==0,"missing string terminator");
320             return {reinterpret_cast<const char*>(bytes.data()+p),n};
321         }
322         std::size_t end=p;

```

include/flex_wire.hpp · continued

```

323     while (end<bytes.size() && bytes[end]!=0) ++end;
324     require(end<bytes.size(),"unterminated key");
325     return {reinterpret_cast<const char*>(bytes.data()+p),end-p};
326 }
327 ByteSpan blob() const {
328     require(type==Type::Blob,"not a blob");
329     const auto p=target(), n=length(); range(bytes,p,n); return bytes.subspan(p,n);
330 }
331 Ref at(std::size_t index) const {
332     require(type==Type::Vector || type==Type::Map || typed(type) || fixed(type),"not a
    • vector");
333     const auto n=length(); require(index<n,"vector index out of range");
334     const auto p=target(), extent=mul(n,child_width);
335     range(bytes,p,extent);
336     const auto loc=add(p,mul(index,child_width));
337     if (type==Type::Vector || type==Type::Map) {
338         const auto tags=add(p,extent); range(bytes,tags,n);
339         const auto t=bytes[tags+index];
340         return {bytes,loc,child_width,1U<<(t&3),static_cast<Type>(t>>2)};
341     }
342     Type element;
343     if (fixed(type)) element=static_cast<Type>(1+(id(type)-16)%3);
344     else if (type==Type::VectorBool) element=Type::Bool;
345     else if (type==Type::VectorStringDeprecated) element=Type::Key;
346     else element=static_cast<Type>(id(type)-10);
347     return {bytes,loc,child_width,1,element};
348 }
349 Ref keys() const {
350     require(type==Type::Map,"not a map"); const auto p=target();
351     const auto pref=mul(child_width,3); require(pref<=p,"missing map prefix");
352     const auto q=p-pref;
353     const auto kw=load_u(bytes,q+child_width,child_width);
354     require(width_ok(static_cast<unsigned>(kw)) && kw<=8,"invalid key-vector width");
355     return {bytes,q,child_width,static_cast<unsigned>(kw),Type::VectorKey};
356 }
357 std::optional<Ref> find(std::string_view key) const {
358     const auto k=keys(); require(k.length()==length(),"key/value length mismatch");
359     std::size_t lo=0, hi=length();
360     while (lo<hi) {
361         const auto m=lo+(hi-lo)/2; const auto cmp=byte_compare(k.at(m).text(),key);
362         if (cmp<0) lo=m+1; else hi=m;
363     }
364     if (lo<length() && byte_compare(k.at(lo).text(),key)==0) return at(lo);
365     return std::nullopt;
366 }
367 };
368
369 struct VerifyOptions {
370     std::size_t max_depth=64;
371     std::size_t max_visits=1'000'000;
372     bool check_utf8=true;
373     bool canonical_bool_and_null=true;
374     bool allow_deprecated=true;
375 };
376 // A bounded recursive teaching verifier. Deliberately no reuse memoization.
377 // Semantic checks are stronger than the upstream structural verifier.
378 inline void verify_ref(const Ref& r, const VerifyOptions& o,
379     std::size_t depth, std::size_t& visits) {

```

include/flex_wire.hpp · continued

```

380     require(depth<=o.max_depth,"nesting limit exceeded");
381     require(visits<o.max_visits,"verification work limit exceeded"); ++visits;
382     require(width_ok(r.parent_width) && width_ok(r.child_width),"invalid width");
383     require(valid_type(r.type),"unknown or reserved type tag");
384     range(r.bytes,r.slot,r.parent_width);
385     if (inline_type(r.type)) {
386         if (r.type==Type::Float) require(r.parent_width==4 || r.parent_width==8,"invalid
    • inline float width");
387         if (o.canonical_bool_and_null && r.type==Type::Bool)
388             require(load_u(r.bytes,r.slot,r.parent_width)<=1,"noncanonical boolean");
389         if (o.canonical_bool_and_null && r.type==Type::Null)
390             require(load_u(r.bytes,r.slot,r.parent_width)==0,"noncanonical null");
391         return;
392     }
393     const auto p=r.target();
394     require((p & (r.child_width-1))==0,"misaligned child position");
395     switch (r.type) {
396         case Type::IndirectInt: case Type::IndirectUInt:
397             range(r.bytes,p,r.child_width); return;
398         case Type::IndirectFloat:
399             require(r.child_width==4 || r.child_width==8,"invalid indirect float width");
400             range(r.bytes,p,r.child_width); return;
401         case Type::String: case Type::Key: {
402             const auto s=r.text(); require(!o.check_utf8 || utf8(s),"invalid UTF-8"); return;
403         }
404         case Type::Blob: (void)r.blob(); return;
405         default: break;
406     }
407     require(r.type==Type::Map || r.type==Type::Vector || typed(r.type) ||
    • fixed(r.type),"not a supported container");
408     require(o.allow_deprecated || r.type!=Type::VectorStringDeprecated,"deprecated vector
    • disabled");
409     const auto n=r.length(); range(r.bytes,p,mul(n,r.child_width));
410     if (r.type==Type::Vector || r.type==Type::Map)
411         range(r.bytes,add(p,mul(n,r.child_width)),n);
412     // Charge every scalar too: inputs cannot hide unbounded work in a typed array.
413     require(n<=o.max_visits-visits,"verification work limit exceeded");
414     for (std::size_t i=0; i<n; ++i) verify_ref(r.at(i),o,depth+1,visits);
415     if (r.type==Type::Map) {
416         const auto k=r.keys(); require(k.length()==n,"map key/value count mismatch");
417         verify_ref(k,o,depth+1,visits);
418         for (std::size_t i=1; i<n; ++i)
419             require(byte_compare(k.at(i-1).text(),k.at(i).text())<0,"map keys unsorted or
    • duplicate");
420     }
421 }
422 inline Ref verify(ByteSpan b, VerifyOptions options={}) {
423     auto r=Ref::root(b); std::size_t visits=0; verify_ref(r,options,0,visits); return r;
424 }
425
426 // Re-encode supported values. Sharing and representation widths are not retained.
427 inline Encoded copy_to(Writer& w, const Ref& r, std::size_t depth=0) {
428     require(depth<=64,"copy nesting limit exceeded");
429     switch (r.type) {
430         case Type::Null: return Writer::null();
431         case Type::Bool: return Writer::boolean(r.as_bool());
432         case Type::Int: return Writer::integer(r.as_int());
433         case Type::UInt: return Writer::uint(r.as_uint());

```

include/flex_wire.hpp · continued

```

434     case Type::Float: return Writer::real(r.as_float());
435     case Type::IndirectInt: return w.indirect(Writer::integer(r.as_int()));
436     case Type::IndirectUInt: return w.indirect(Writer::uint(r.as_uint()));
437     case Type::IndirectFloat: return w.indirect(Writer::real(r.as_float()));
438     case Type::String: return w.string(r.text());
439     case Type::Key: return w.key(r.text());
440     case Type::Blob: return w.blob(r.blob());
441     case Type::Map: {
442         std::vector<std::pair<std::string,Encoded>> fields; const auto k=r.keys();
443         for (std::size_t i=0; i<r.length(); ++i)
444             fields.emplace_back(std::string(k.at(i).text()),copy_to(w,r.at(i),depth+1));
445         return w.map(std::move(fields));
446     }
447     default: {
448         require(r.type==Type::Vector || typed(r.type) || fixed(r.type),"unsupported copy
• type");
449         std::vector<Encoded> values; values.reserve(r.length());
450         for (std::size_t i=0; i<r.length(); ++i)
451             values.push_back(copy_to(w,r.at(i),depth+1));
452         if (r.type==Type::Vector) return w.vector(values);
453         Type element=Type::Key;
454         if (fixed(r.type)) element=static_cast<Type>(1+(id(r.type)-16)%3);
455         else if (r.type==Type::VectorBool) element=Type::Bool;
456         else if (r.type!=Type::VectorStringDeprecated)
457             element=static_cast<Type>(id(r.type)-10);
458         return w.typed_vector(values,element,fixed(r.type) ?
• static_cast<unsigned>(r.length()) : 0);
459     }
460 }
461 // Same-layout mutation only. Aliases observe the same changed bytes.
462 inline bool mutate_int(Bytes& owner, const Ref& r, std::int64_t value) {
463     require(r.bytes.data()==owner.data() && r.bytes.size()==owner.size(),"wrong mutable
• owner");
464     require(r.type==Type::Int || r.type==Type::IndirectInt,"not a signed integer");
465     const auto w=r.type==Type::Int ? r.parent_width : r.child_width;
466     const auto p=r.type==Type::Int ? r.slot : r.target();
467     if (int_width(value)>w) return false;
468     range(owner,p,w); const auto bits=std::bit_cast<std::uint64_t>(value);
469     for (unsigned j=0; j<w; ++j) owner[p+j]=static_cast<std::uint8_t>(bits>>(8*j));
470     return true;
471 }
472 inline bool mutate_same_length_string(Bytes& owner, const Ref& r, std::string_view value)
473     {
474     require(r.bytes.data()==owner.data() && r.bytes.size()==owner.size(),"wrong mutable
• owner");
475     require(r.type==Type::String && utf8(value),"not a valid UTF-8 string mutation");
476     if (r.length()!=value.size()) return false;
477     const auto p=r.target(); range(owner,p,value.size());
478     if (!value.empty()) std::memmove(owner.data()+p,value.data(),value.size());
479     return true;
480 }
481 } // namespace flexlab

```

REFERENCE & COMPLETE SOURCE

APPENDIX E

Complete mutable semantic laboratory

File: `include/mutable_lab.hpp`. This implementation supplies cells, document-scoped generational handles, standard-container payloads, explicit-root collection, and standard-format freeze/thaw. The optimized allocation, map, transaction, and concurrency mechanisms proposed for LiveFlex are not implemented here.

```
include/mutable_lab.hpp

1  // Executable semantic laboratory for the book's mutable capstone.
2  // SPDX-License-Identifier: MIT
3  // This is NOT the proposed slab/SIMD production implementation.
4  #pragma once
5  #include "flex_wire.hpp"
6  #include <atomic>
7  #include <memory>
8  #include <unordered_map>
9  #include <unordered_set>
10 #include <variant>
11 #include <type_traits>
12
13 namespace flexlab::live {
14 enum class Kind : std::uint8_t { Null, Bool, Int, UInt, Float, String, Key, Blob, Vector,
  • Map, F64 };
15 struct alignas(8) Cell {
16     std::uint64_t payload=0;
17     std::uint32_t aux=0;
18     Kind kind=Kind::Null;
19     std::uint8_t flags=0;
20     std::uint16_t reserved=0;
21 };
22 static_assert(sizeof(Cell)==16 && alignof(Cell)==8);
23 static_assert(offsetof(Cell,aux)==8 && offsetof(Cell,kind)==12);
24 static_assert(std::is_trivially_copyable_v<Cell>);
25 inline bool small_text(const Cell& c) { return (c.kind==Kind::String ||
  • c.kind==Kind::Key) && (c.flags&0x80); }
26 inline bool external(const Cell& c) {
27     return c.kind>=Kind::String && !small_text(c);
28 }
29 inline Cell null() { return {}; }
30 inline Cell boolean(bool x) { return {x ? 1ULL : 0ULL,0,Kind::Bool,0,0}; }
31 inline Cell integer(std::int64_t x) { return
  • {std::bit_cast<std::uint64_t>(x),0,Kind::Int,0,0}; }
32 inline Cell uint(std::uint64_t x) { return {x,0,Kind::UInt,0,0}; }
33 inline Cell real(double x) { return {std::bit_cast<std::uint64_t>(x),0,Kind::Float,0,0};
  • }
34 inline std::int64_t as_int(Cell c) { require(c.kind==Kind::Int,"not an integer cell");
  • return std::bit_cast<std::int64_t>(c.payload); }
35 inline double as_float(Cell c) { require(c.kind==Kind::Float,"not a float cell"); return
  • std::bit_cast<double>(c.payload); }
36 struct TransparentHash {
37     using is_transparent=void;
38     std::size_t operator()(std::string_view x) const { return
```

include/mutable_lab.hpp · continued

```

    • std::hash<std::string_view>{}(x); }
39 };
40 class Document {
41     struct Text { std::string data; };
42     struct Blob { Bytes data; };
43     struct Vector { std::vector<Cell> data; };
44     struct Map { std::vector<std::pair<std::uint32_t,Cell>> data; };
45     struct F64 { std::vector<double> data; };
46     using Node=std::variant<Text,Blob,Vector,Map,F64>;
47     struct Slot {
48         std::uint32_t generation=1;
49         std::unique_ptr<Node> node;
50     };
51     inline static std::atomic<std::uint64_t> next_owner_{1};
52     const std::uint32_t owner_;
53     std::vector<Slot> slots_;
54     std::vector<std::uint32_t> free_;
55     std::unordered_map<std::string,std::uint32_t,TransparentHash,std::equal_to<>> symbols_;
56     std::vector<std::string> names_;
57     static std::uint32_t new_owner() {
58         const auto x=next_owner_.fetch_add(1,std::memory_order_relaxed);
59         require(x<=UINT32_MAX,"document identity space exhausted"); return
    • static_cast<std::uint32_t>(x);
60     }
61     void own(Cell c) const {
62         if (!external(c)) return;
63         require(c.aux==owner_,"cross-document handle");
64         const auto i=static_cast<std::uint32_t>(c.payload),
    • g=static_cast<std::uint32_t>(c.payload>>32);
65         require(i<slots_.size() && slots_[i].node && slots_[i].generation==g,"stale handle");
66     }
67     Node& node(Cell c) { own(c); require(external(c),"not an external cell"); return
    • *slots_[static_cast<std::uint32_t>(c.payload)].node; }
68     const Node& node(Cell c) const { own(c); require(external(c),"not an external cell");
    • return *slots_[static_cast<std::uint32_t>(c.payload)].node; }
69     template<class T> T& get(Cell c, Kind kind) {
70         require(c.kind==kind,"cell kind mismatch"); return std::get<T>(node(c));
71     }
72     template<class T> const T& get(Cell c, Kind kind) const {
73         require(c.kind==kind,"cell kind mismatch"); return std::get<T>(node(c));
74     }
75     Cell allocate(Kind kind, Node n) {
76         auto ptr=std::make_unique<Node>(std::move(n)); // allocate before publishing
77         std::uint32_t i;
78         if (free_.empty()) {
79             require(slots_.size()<UINT32_MAX,"slot space exhausted");
80             i=static_cast<std::uint32_t>(slots_.size()); slots_.push_back({});
81         } else { i=free_.back(); free_.pop_back(); }
82         slots_[i].node=std::move(ptr);
83         return { (std::uint64_t{slots_[i].generation}<<32)|i,owner_,kind,0,0 };
84     }
85     std::uint32_t intern(std::string_view key) {
86         require(key.find('\0')==key.npos && utf8(key),"invalid map key");
87         if (const auto it=symbols_.find(key); it!=symbols_.end()) return it->second;
88         require(names_.size()<UINT32_MAX,"key identity space exhausted");
89         const auto id=static_cast<std::uint32_t>(names_.size());
90         names_.emplace_back(key);
91         try { symbols_.emplace(names_.back(),id); }

```

include/mutable_lab.hpp · continued

```

92     catch (...) { names_.pop_back(); throw; }
93     return id;
94 }
95 template<class F> void children(Cell c, F&& f) const {
96     if (c.kind==Kind::Vector) for (auto v:get<Vector>(c,Kind::Vector).data) f(v);
97     if (c.kind==Kind::Map) for (const auto& kv:get<Map>(c,Kind::Map).data) f(kv.second);
98 }
99 Encoded freeze_one(Writer& w, Cell c, std::unordered_map<std::uint64_t,Encoded>& done,
100                  std::unordered_set<std::uint64_t>& active, std::size_t depth) const
101 • {
102     require(depth<=64,"freeze nesting limit exceeded"); own(c);
103     if (external(c)) {
104         if (const auto it=done.find(c.payload); it!=done.end()) return it->second;
105         require(active.insert(c.payload).second,"cycle cannot be exported to FlexBuffers");
106     }
107     Encoded result;
108     switch (c.kind) {
109     case Kind::Null: result=Writer::null(); break;
110     case Kind::Bool: result=Writer::boolean(c.payload!=0); break;
111     case Kind::Int: result=Writer::integer(as_int(c)); break;
112     case Kind::UInt: result=Writer::uint(c.payload); break;
113     case Kind::Float: result=Writer::real(as_float(c)); break;
114     case Kind::String: result=w.string(text(c)); break;
115     case Kind::Key: result=w.key(text(c)); break;
116     case Kind::Blob: result=w.blob(get<Blob>(c,Kind::Blob).data); break;
117     case Kind::Vector: {
118         std::vector<Encoded> v;
119         for (auto item:get<Vector>(c,Kind::Vector).data)
120             • v.push_back(freeze_one(w,item,done,active,depth+1));
121         result=w.vector(v); break;
122     }
123     case Kind::Map: {
124         std::vector<std::pair<std::string,Encoded>> fields;
125         for (const auto& [k,value]:get<Map>(c,Kind::Map).data)
126             fields.emplace_back(names_[k],freeze_one(w,value,done,active,depth+1));
127         result=w.map(std::move(fields)); break;
128     }
129     case Kind::F64: {
130         std::vector<Encoded> v;
131         for (auto x:get<F64>(c,Kind::F64).data) v.push_back(Writer::real(x));
132         result=w.typed_vector(v,Type::Float); break;
133     }
134     }
135     if (external(c)) { active.erase(c.payload); done.emplace(c.payload,result); }
136     return result;
137 }
138 Cell thaw_one(const Ref& r, std::size_t depth) {
139     require(depth<=64,"thaw nesting limit exceeded");
140     switch (r.type) {
141     case Type::Null: return null();
142     case Type::Bool: return boolean(r.as_bool());
143     case Type::Int: case Type::IndirectInt: return integer(r.as_int());
144     case Type::UInt: case Type::IndirectUInt: return uint(r.as_uint());
145     case Type::Float: case Type::IndirectFloat: return real(r.as_float());
146     case Type::String: return string(r.text());
147     case Type::Key: return key(r.text());
148     case Type::Blob: return blob(r.blob());
149     case Type::Map: {

```

include/mutable_lab.hpp · continued

```

148     auto out=map(); const auto k=r.keys();
149     for (std::size_t i=0; i<r.length(); ++i)
    • set(out,k.at(i).text(),thaw_one(r.at(i),depth+1));
150     return out;
151 }
152 default: {
153     require(r.type==Type::Vector || typed(r.type) || fixed(r.type),"unsupported
    • import");
154     if (r.type==Type::VectorFloat) {
155         std::vector<double> values; values.reserve(r.length());
156         for (std::size_t i=0; i<r.length(); ++i) values.push_back(r.at(i).as_float());
157         return f64(std::move(values));
158     }
159     auto out=vector();
160     for (std::size_t i=0; i<r.length(); ++i) push(out,thaw_one(r.at(i),depth+1));
161     return out;
162 }
163 }
164 }
165 public:
166 Document() : owner_(new_owner()) {}
167 Document(const Document&)=delete;
168 Document& operator=(const Document&)=delete;
169 Cell string(std::string_view s, bool is_key=false) {
170     require(utf8(s) && (!is_key || s.find('\0')==s.npos),"invalid text");
171     const auto kind=is_key ? Kind::Key : Kind::String;
172     if (s.size()<=12) {
173         Cell c; c.kind=kind; c.flags=static_cast<std::uint8_t>(0x80|s.size());
174         if (!s.empty()) std::memcpy(&c,s.data(),s.size());
175         return c;
176     }
177     return allocate(kind,Text{std::string(s)});
178 }
179 Cell key(std::string_view s) { return string(s,true); }
180 // Returns an owning string, so even an inline temporary is safe.
181 std::string text(Cell c) const {
182     require(c.kind==Kind::String || c.kind==Kind::Key,"not a text cell");
183     if (small_text(c)) return {reinterpret_cast<const
    • char*>(&c),static_cast<std::size_t>(c.flags&15)};
184     return std::get<Text>(node(c)).data;
185 }
186 Cell blob(ByteSpan b) { return allocate(Kind::Blob,Blob{Bytes(b.begin(),b.end())}); }
187 Cell vector() { return allocate(Kind::Vector,Vector{}); }
188 Cell map() { return allocate(Kind::Map,Map{}); }
189 Cell f64(std::vector<double> data={}) { return
    • allocate(Kind::F64,F64{std::move(data)}); }
190 void push(Cell c, Cell value) { own(value);
    • get<Vector>(c,Kind::Vector).data.push_back(value); }
191 Cell at(Cell c, std::size_t i) const { return get<Vector>(c,Kind::Vector).data.at(i); }
192 void replace(Cell c, std::size_t i, Cell v) { own(v);
    • get<Vector>(c,Kind::Vector).data.at(i)=v; }
193 void erase_at(Cell c, std::size_t i) {
194     auto& v=get<Vector>(c,Kind::Vector).data; require(i<v.size(),"bad erase index");
195     v.erase(v.begin()+static_cast<std::ptrdiff_t>(i));
196 }
197 void set(Cell map_cell, std::string_view k, Cell value) {
198     own(value); auto& entries=get<Map>(map_cell,Kind::Map).data;
199     const auto key=intern(k);

```

include/mutable_lab.hpp · continued

```

200     for (auto& [id,v]:entries) if (id==key) { v=value; return; }
201     entries.emplace_back(key,value);
202 }
203 std::optional<Cell> find(Cell map_cell, std::string_view k) const {
204     const auto& entries=get<Map>(map_cell,Kind::Map).data;
205     const auto it=symbols_.find(k); if (it==symbols_.end()) return std::nullopt;
206     for (const auto& [id,v]:entries) if (id==it->second) return v;
207     return std::nullopt;
208 }
209 bool erase(Cell map_cell, std::string_view k) {
210     auto& entries=get<Map>(map_cell,Kind::Map).data;
211     const auto key=symbols_.find(k); if (key==symbols_.end()) return false;
212     for (std::size_t i=0; i<entries.size(); ++i) if (entries[i].first==key->second) {
213         entries.erase(entries.begin()+static_cast<std::ptrdiff_t>(i)); return true;
214     }
215     return false;
216 }
217 void push_number(Cell c, double v) { get<F64>(c,Kind::F64).data.push_back(v); }
218 std::span<double> numbers(Cell c) { return get<F64>(c,Kind::F64).data; }
219 Bytes freeze(Cell root) const {
220     Writer w; std::unordered_map<std::uint64_t,Encoded> done;
221     std::unordered_set<std::uint64_t> active;
222     const auto r=freeze_one(w,root,done,active,0); return w.finish(r);
223 }
224 Cell thaw(ByteSpan bytes) { const auto r=verify(bytes); return thaw_one(r,0); }
225 std::size_t collect(std::span<const Cell> roots) {
226     std::vector<bool> mark(slots_.size(),false);
227     std::vector<Cell> work(roots.begin(),roots.end());
228     while (!work.empty()) {
229         const auto c=work.back(); work.pop_back(); own(c); if (!external(c)) continue;
230         const auto i=static_cast<std::uint32_t>(c.payload); if (mark[i]) continue;
231         mark[i]=true; children(c,[&](Cell child) { work.push_back(child); });
232     }
233     free_.reserve(add(free_.size(),slots_.size()));
234     std::size_t freed=0;
235     for (std::size_t i=0; i<slots_.size(); ++i) if (slots_[i].node && !mark[i]) {
236         slots_[i].node.reset(); ++freed;
237         if (slots_[i].generation!=UINT32_MAX) {
238             ++slots_[i].generation; free_.push_back(static_cast<std::uint32_t>(i));
239         } // Retire exhausted slots rather than wrap the generation.
240     }
241     return freed;
242 }
243 };
244 } // namespace flexlab::live

```

REFERENCE & COMPLETE SOURCE

APPENDIX F

Complete FlatBuffers table laboratory

File: include/flat_table_lab.hpp. This deliberately small implementation constructs and reads the worked Sample table. It is not a schema compiler, general FlatBuffers builder, or complete FlatBuffers verifier.

```
include/flat_table_lab.hpp

1 // A checked, schema-specific FlatBuffers table lesson, NOT a general compiler.
2 // SPDX-License-Identifier: MIT
3 #pragma once
4 #include "flex_wire.hpp"
5 namespace flexlab::flat {
6 // table Sample { count:int = 0; name:string; } root_type Sample;
7 inline Bytes sample(std::int32_t count, std::string_view name) {
8     require(name.size() <= UINT32_MAX, "string too long");
9     Bytes b;
10    store_u(b, 12, 4); // root table at 12
11    store_u(b, 8, 2); store_u(b, 12, 2); // vtable size, inline object size
12    store_u(b, 4, 2); store_u(b, 8, 2); // count and name positions in table
13    store_u(b, 8, 4); // table 12 minus displacement 8 -> vtable 4
14    store_u(b, std::bit_cast<std::uint32_t>(count), 4);
15    store_u(b, 4, 4); // name offset at 20 -> length at 24
16    store_u(b, name.size(), 4);
17    b.insert(b.end(), name.begin(), name.end()); b.push_back(0);
18    b.resize(align_up(b.size(), 4), 0); return b;
19 }
20 class Sample {
21     ByteSpan b_;
22     std::size_t table_, vt_;
23     std::size_t field(unsigned index, unsigned width) const {
24         const auto entry = 4 + 2 * index;
25         const auto vt_size = load_u(b_, vt_, 2);
26         if (entry + 2 > vt_size) return 0;
27         const auto off = load_u(b_, vt_ + entry, 2); if (!off) return 0;
28         const auto object_size = load_u(b_, vt_ + 2, 2);
29         require(off >= 4 && off <= object_size && width <= object_size - off, "field outside inline
    • object");
30         const auto p = add(table_, static_cast<std::size_t>(off)); range(b_, p, width); return p;
31     }
32 public:
33     explicit Sample(ByteSpan b) : b_(b), table_(0), vt_(0) {
34         table_ = static_cast<std::size_t>(load_u(b_, 0, 4)); range(b_, table_, 4);
35         const auto d = load_i(b_, table_, 4);
36         if (d >= 0) { require(static_cast<std::uint64_t>(d) <= table_, "vtable underflow");
    • vt_ = table_ - static_cast<std::size_t>(d); }
37         else vt_ = add(table_, static_cast<std::size_t>(-d));
38         const auto len = load_u(b_, vt_, 2);
39         require(len >= 4 && len % 2 == 0, "bad vtable size");
    • range(b_, vt_, static_cast<std::size_t>(len));
40         const auto object = load_u(b_, vt_ + 2, 2); require(object >= 4, "bad inline object size");
41         range(b_, table_, static_cast<std::size_t>(object));
42     }
43     std::int32_t count() const {
```

```
include/flat_table_lab.hpp · continued
```

```
44     const auto p=field(0,4); return p ? static_cast<std::int32_t>(load_i(b_,p,4)) : 0;
45 }
46 std::optional<std::string_view> name() const {
47     const auto p=field(1,4); if (!p) return std::nullopt;
48     const auto q=add(p,static_cast<std::size_t>(load_u(b_,p,4)));
49     const auto n=static_cast<std::size_t>(load_u(b_,q,4));
50     const auto text=add(q,4); range(b_,text,add(n,1));
51     require(b_[text+n]==0,"missing FlatBuffers string terminator");
52     return std::string_view(reinterpret_cast<const char*>(b_.data()+text),n);
53 }
54 };
55 } // namespace flexlab::flat
```

[REFERENCE & COMPLETE SOURCE](#)[APPENDIX G](#)

Sources and edition notes

The numbered citations in this book refer to the primary sources below. The three implementation files are pinned to commit `b8431fbcd7a5c71817f314e18b332c0648554efa`, dated 14 September 2026. Other links are documentation locations consulted for this edition on 18 September 2026; rolling documentation can change. The Java reference deliberately uses an explicit JDK 22 API baseline. C++ draft links explain the language rules discussed, not a claim of complete support in every compiler.

Wire-format explanations and implementation observations are distinguished from the original LiveFlex design. Test receipts describe only executions performed for the supplied companion. The book does not claim upstream-runtime interoperability, production hardening, a completed optimized allocator, or benchmark superiority.

[1] Google FlatBuffers: FlexBuffers C++ implementation

Pinned source; type codes, layout, reader, builder, mutations, verifier.

[Open primary source](#)

[2] Google FlatBuffers: Table implementation

Pinned source; vtable lookup, accessors, mutation, verification helpers.

[Open primary source](#)

[3] Google FlatBuffers: FlatBufferBuilder implementation

Pinned source; reverse construction, offset conversion, vtables, 64-bit-region mechanisms.

[Open primary source](#)

[4] FlatBuffers documentation: Internals

Wire-format overview, scalar layout, offsets, tables, structs and vectors.

[Open primary source](#)

[5] FlatBuffers documentation: FlexBuffers

Schemaless format overview and API context.

[Open primary source](#)

[6] FlatBuffers documentation: Schema

Schema types, declarations and attributes.

[Open primary source](#)

[7] FlatBuffers documentation: Evolution

Compatibility rules and schema-evolution discipline.

[Open primary source](#)

[8] FlatBuffers documentation: C++

Generated C++ usage, verification and mutation context.

[Open primary source](#)

[9] FlatBuffers documentation: flatc

Compiler and code-generation options.

[Open primary source](#)

[10] FlatBuffers documentation: Annotation

Inspecting and annotating binary layouts.

[Open primary source](#)

[11] Abseil: Swiss Tables design notes

Primary design notes for metadata-first hash probing. LiveFlex's proposal is separately specified, not a copy of Abseil's complete implementation.

[Open primary source](#)

[12] C++ working draft: bit.cast

Requirements for representation-preserving bit casts.

[Open primary source](#)

[13] C++ working draft: Data races

Data races, happens-before and undefined behavior.

[Open primary source](#)

[14] C++ working draft: Memory resources

Polymorphic memory resources and resource lifetimes.

[Open primary source](#)

[15] C++ working draft: Atomic smart pointers

Atomic shared-pointer operations and implementation-defined lock-free status.

[Open primary source](#)

[16] LLVM: Auto-vectorization documentation

Loop and SLP vectorizers, diagnostics and constraints.

[Open primary source](#)

[17] Google Benchmark: User guide

Repetitions, optimization barriers, timing and measurement tools.

[Open primary source](#)

[18] Clang: AddressSanitizer

Memory-error instrumentation and execution requirements.

[Open primary source](#)

[19] Clang: UndefinedBehaviorSanitizer

Undefined-behavior instrumentation and its scope.

[Open primary source](#)

[20] Clang: ThreadSanitizer

Data-race instrumentation. No ThreadSanitizer run is claimed for this single-threaded laboratory.

[Open primary source](#)

[21] LLVM: LibFuzzer

Coverage-guided in-process fuzzing, targets, corpora and limits.

[Open primary source](#)

[22] Rust standard library: slice::from_raw_parts

Validity, alignment, allocation and lifetime conditions for borrowed raw slices.

[Open primary source](#)

[23] Java SE 22 API: MemorySegment

A specific Foreign Function & Memory API baseline, not a claim that JDK 22 is the latest release.

[Open primary source](#)

[24] ECMAScript specification: DataView objects

Explicit-endian byte access and BigInt scalar operations.

[Open primary source](#)

[25] Linux kernel documentation: False sharing

False-sharing behavior and investigation techniques.

[Open primary source](#)

Edition and reproducibility

First edition, 18 September 2026. The source commit above fixes the upstream implementation baseline. The accompanying archive contains the original source, fixtures, optional oracle programs, compiler/test receipts, and the editable manuscript. No upstream implementation files or font files are redistributed in the archive.

Original companion code is provided under the MIT license included in the archive. Upstream FlatBuffers code is separately governed by its own license. This book paraphrases and analyzes the implementation; the full source appendices reproduce the original companion, not the upstream header.

The strongest next verification step is to execute and extend the independent C++ oracle, then run stateful differential testing and sustained coverage-guided fuzzing on every intended target platform. Performance claims should be published only after the workload-normalized benchmarks in Chapter 22 have been run.