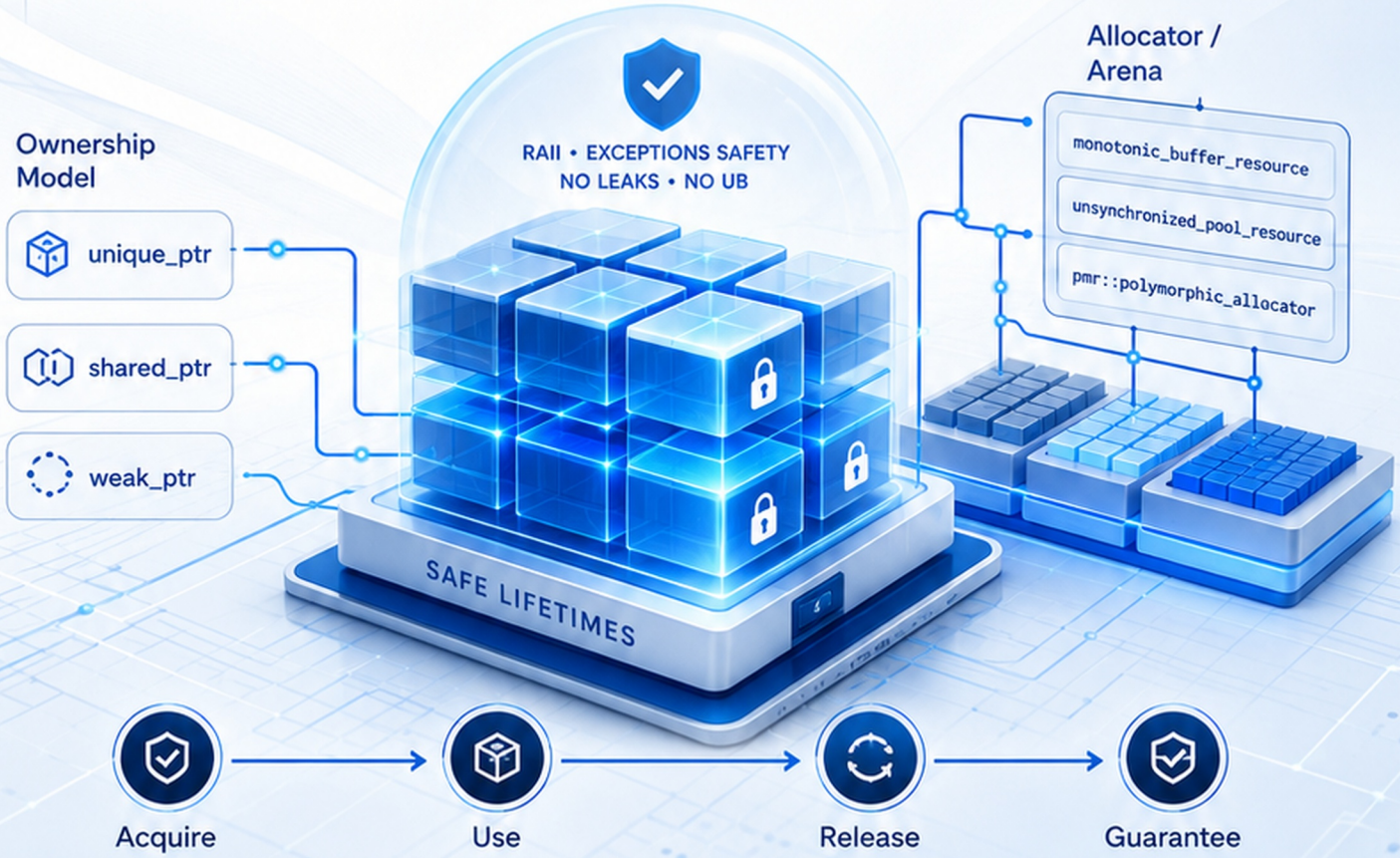
Third Edition

Advanced Memory Management in Modern C++

Safe, disciplined, and high-performance techniques for C++23 and C++26

Aligned with the latest C++ Core Guidelines



Prepared by
Ayman Alheraki

AI-assisted with full human review and verification

Advanced Memory Management in Modern C++

Third Edition - C++23 / C++26

Safe, disciplined, and high-performance techniques

Prepared by Ayman Alheraki

AI-assisted in research, drafting, and design; reviewed and verified for this edition.

August 2026

Technical note: the C++ Core Guidelines are a living community project led by Bjarne Stroustrup and Herb Sutter; they are not the normative ISO C++ standard.

Publication Note

Third Edition - August 2026

This edition is a complete reconstruction of the earlier memory-management book. It preserves the original purpose—help C++ programmers write safer and more efficient systems—but reorganizes the subject around an enforceable memory-safety discipline rather than a catalog of features.

The text distinguishes guarantees encoded by types and RAII, conditions checked by contracts/hardened libraries/tools, and residual unsafe boundaries requiring focused review. C++23 and C++26 are integrated where they materially improve lifetime, ownership, interoperability, bounded storage, or reclamation.

Examples are presented as real selectable PDF text with syntax highlighting, not as code screenshots, so they can be copied into an editor. C++26 support is incremental across compilers and standard libraries, so feature-test macros and CI on actual toolchains are part of the recommended practice.

Editorial position

The strongest defense of modern C++ is not to deny its unsafe capabilities, but to make unsafe capabilities exceptional, visible, and continuously checked.

Author's Introduction

From the Second Edition to a Safety Discipline

In the second edition, I wrote that memory management is not merely one technique among many, but a foundation for the performance and stability of modern applications. That edition emphasized RAII, smart pointers, leak prevention, and the evolution of Modern C++ through C++23. This third edition keeps that purpose but raises the bar.

The goal now is not only to explain tools. It is to show how a C++ team can establish a strict discipline: ownership explicit, borrowing shorter than ownership, sequence bounds traveling with data, cleanup automatic, low-level lifetime manipulation isolated, and every risky boundary checked by tools and review.

Why this matters

Modern C++ has enough language and library support to make safe code the normal path. The hard part is consistency.

Author's Introduction - continued

C++23 and C++26 change what good memory engineering looks like

C++23 brings std::out_ptr/std::inout_ptr for C interop, std::start_lifetime_as for specialized explicit-lifetime work, allocate_at_least for allocator-aware storage, and standard scope guards for lexical cleanup and rollback.

C++26 goes further with contracts, standard-library hardening, std::inplace_vector, std::indirect, std::polymorphic, owner_hash/owner_equal, hazard pointers, RCU, and targeted lifetime/alignment utilities.

My aim is that the reader finishes with a practical policy—not just knowledge—and can build C++ software whose memory behavior is predictable, reviewable, measurable, and defensible.

Author note

The book targets intermediate-to-advanced C++ developers, systems programmers, performance/embedded engineers, reviewers, and technical leads.

What “Memory-Safe C++” Means Here

A precise claim instead of marketing language

This book does not claim that arbitrary C++ is automatically memory-safe. C++ deliberately exposes low-level operations that can violate bounds, lifetime, type, and concurrency rules.

Here, “memory-safe C++” means a deliberately constrained engineering profile: RAII resources, non-owning raw pointers/references, explicit dynamic ownership, bounds-aware interfaces, controlled invalidation, isolated unsafe operations, defined concurrent reclamation, and mandatory static/dynamic verification.

This matches the direction of the C++ Core Guidelines safety profiles and uses C++26 hardening/contracts as additional defense in depth.

Important limitation

No discipline can promise correctness against compiler defects, hardware faults, malicious external binaries, or arbitrary unchecked unsafe code. A safety claim must name its boundary.

How to Use This Book

A compact field guide rather than a slow tutorial

Each chapter opens with a concise model, followed by four one-page Discipline Cards. Every card contains one principle, a short explanation, engineering rules, risk/application context, a copyable syntax-highlighted code pattern or visual model, and a review gate.

Read Parts I–II to establish a baseline; Parts III–IV for allocation, interop, and concurrency; Part V for C++26; Part VI to turn technical knowledge into measurable engineering policy.

Suggested workflow

Read -> adopt one rule -> automate one check -> migrate one unsafe boundary -> measure -> repeat.

C++23 Upgrade Map

Memory-relevant facilities to adopt deliberately

For application code, the most immediately useful additions are smart-pointer adaptors for C APIs and standard scope guards. Low-level authors gain explicit lifetime-start utilities and allocate_at_least.

Adopt features because they remove custom code, clarify invariants, or improve verification—not merely because the language mode is newer.

C++23 focus

out_ptr / inout_ptr - start_lifetime_as - allocate_at_least - scope guards - mdspan and modern view-based design.

C++26 Upgrade Map

The safety and memory-management step-change

C++26 adds contracts, standard-library hardening, fixed-capacity inplace_vector, new value-like dynamic ownership wrappers, shared-owner hashing/equality, hazard pointers, RCU, and targeted lifetime/alignment utilities.

Implementation support arrives incrementally. Verify actual compiler + standard-library capability with feature-test macros and CI.

C++26 focus

Contracts + hardening; standardized reclamation; bounded storage; safer value-like dynamic ownership.

Core Guidelines Alignment

Latest strong recommendations used as the policy backbone

The C++ Core Guidelines are a living project edited by Bjarne Stroustrup and Herb Sutter. The browsable document retrieved for this edition is dated June 14, 2026. It emphasizes static type safety, resource safety, RAII, non-owning raw pointers, scoped objects, avoidance of explicit new/delete, smart-pointer ownership, span interfaces, and tool-supported type/bounds/lifetime profiles.

This edition uses those recommendations as a baseline and extends them with C++23 and C++26 facilities. The architectural principle remains primary: make invalid ownership and lifetime relationships difficult to express.

Terminology correction

The C++ Core Guidelines are not the ISO C++ standard; they are a living practical guideline project for using modern ISO C++ effectively.

Contents

Contents

Contents

Contents

PART 1

FOUNDATIONS

Memory safety, lifetime, ownership, and RAII

1

01

MEMORY DISCIPLINE CHAPTER

The Memory-Safety Discipline

Replace folklore with an explicit safety model: ownership, lifetime, bounds, initialization, concurrency, and verification.

DISCIPLINE CARD 1.1

Threat Model: What Can Actually Go Wrong

PRINCIPLE **Name the failure classes before choosing the defense.**

Why it matters

Leaks are only one category. Include use-after-free, out-of-bounds access, double destruction, invalid object lifetime, misalignment, uninitialized reads, iterator invalidation, data races, and ownership cycles.

Engineering rules

- Classify every pointer-like value as owner or borrower.
- Keep size with data at interfaces.
- Make lifetime boundaries visible.
- Assign a tool or type-system defense to every high-risk class.

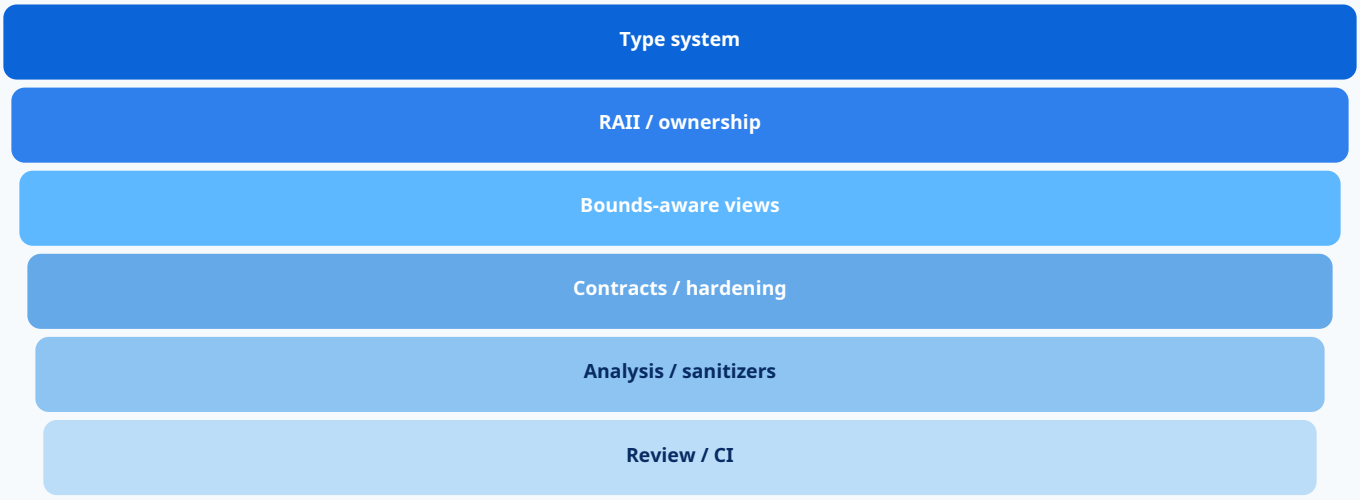
RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Name the failure classes before choosing the defense. Classify every pointer-like value as owner or borrower. Keep size with data at interfaces.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

Review cannot approve memory-sensitive code unless the failure class and prevention mechanism are both obvious.

Core Guidelines P.4, P.8, R.1

DISCIPLINE CARD 1.2

No Honest Shortcut to Absolute Safety

PRINCIPLE **Constrain the language and enforce the constraint.**

Why it matters

Arbitrary C++ is not automatically memory-safe. A disciplined profile can eliminate broad bug classes when RAII, bounds-aware abstractions, isolated unsafe code, and mandatory verification work together. Residual risk remains at low-level casts, FFI, concurrency, unchecked preconditions, and external code.

Engineering rules

- Call unsafe code unsafe and isolate it.
- Do not claim safety from smart pointers alone.
- Treat data races as memory-safety relevant.
- Require analysis, sanitizer, test, and review evidence.

RISK IF IGNORED

Typical failure: an address travels without its valid extent, so an otherwise small indexing mistake becomes an out-of-bounds read or write.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Constrain the language and enforce the constraint. Call unsafe code unsafe and isolate it. Do not claim safety from smart pointers alone.

REVIEW CHECKLIST

- Call unsafe code unsafe and isolate it.
- Do not claim safety from smart pointers alone.
- Treat data races as memory-safety relevant.
- Require analysis, sanitizer, test, and review evidence.

DESIGN CONSEQUENCE

Constrain the language and enforce the constraint. Make the rule visible in the type, API, scope, or build policy so a later refactor cannot silently remove it.

REVIEW GATE

Every safety claim must state both enforced rules and the remaining unsafe boundary.

Core Guidelines safety profiles; C++26 hardening/contracts

DISCIPLINE CARD 1.3

The Six-Layer Defense

PRINCIPLE **Layer prevention and detection so one missed defense does not become a silent exploit.**

Why it matters

Use types to remove invalid states, RAII to pair acquire/release, views to preserve bounds, contracts and hardened libraries for violated preconditions, sanitizers for dynamic faults, and process controls to prevent regression.

Engineering rules

- Prefer prevention over detection.
- Prefer compile-time checking when practical.
- Use runtime checks where static proof is unavailable.
- Keep production diagnostics actionable.

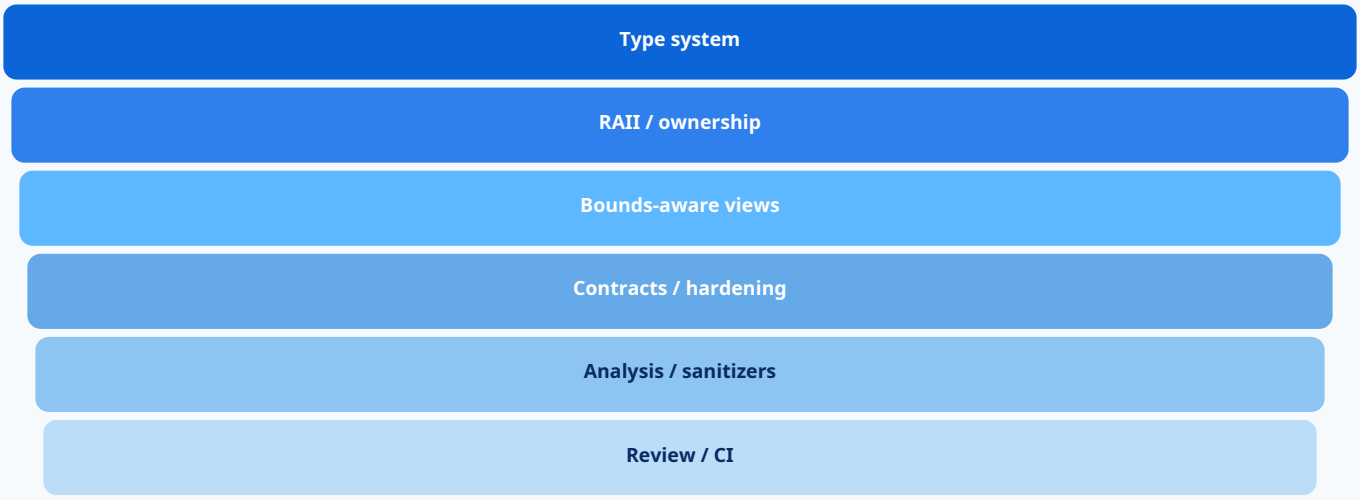
RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Layer prevention and detection so one missed defense does not become a silent exploit. Prefer prevention over detection. Prefer compile-time checking when practical.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

Critical components should have at least two independent defenses for their highest-risk failure modes.

Core Guidelines P.5-P.7, P.12-P.13

DISCIPLINE CARD 1.4

Default-Safe / Explicit-Unsafe

PRINCIPLE **Make the safe path shorter than the unsafe path.**

Why it matters

Application code should not routinely own through raw pointers, perform pointer arithmetic, or scatter allocation/deallocation. Low-level operations belong in named adapters, allocators, resource handles, and platform layers with explicit invariants.

Engineering rules

- Value semantics first.
- Unique ownership second.
- Shared ownership only for genuine shared lifetime.
- Raw pointers/references are observations by default.

RISK IF IGNORED

Typical failure: storage, construction, alignment, or deallocation contracts are mixed, causing leaks, fragmentation, invalid lifetimes, or mismatched release.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Make the safe path shorter than the unsafe path. Value semantics first. Unique ownership second.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```cpp
struct Image { /* value-like */ };
Image load_image(Path p);
std::unique_ptr<Node> make_node();
void render(const Image&);        // borrow, non-null
void inspect(const Node*);        // borrow, nullable
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Make the safe path shorter than the unsafe path. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

If ownership or nullability is not obvious from the API shape, redesign the API.

Core Guidelines R.3, R.4, R.20-R.21

02

MEMORY DISCIPLINE CHAPTER

Storage, Object Lifetime, and Undefined Behavior

Understand the distinction between storage and a live object; most advanced memory bugs hide in that gap.

DISCIPLINE CARD 2.1

Storage Duration Is Not Object Lifetime

PRINCIPLE **Bytes can exist while no T object exists there.**

Why it matters

Automatic, static, thread, and dynamic storage duration describe storage. Object lifetime is a separate language fact. This distinction matters in arenas, placement construction, serialization buffers, unions, and explicit-lifetime facilities.

Engineering rules

- Do not infer object lifetime from address existence.
- Use normal construction in ordinary code.
- Reserve manual lifetime control for infrastructure.
- State size, alignment, and lifetime preconditions for raw storage.

RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Bytes can exist while no T object exists there. Do not infer object lifetime from address existence. Use normal construction in ordinary code.

MODEL / VISUAL

Storage lifetime != object lifetime

HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

Any function accepting raw storage must document what object, if any, is alive in it.

C++ object lifetime; C++23 start_lifetime_as

DISCIPLINE CARD 2.2

Lifetime Begins and Ends Precisely

PRINCIPLE **The valid-access window is narrower than the storage window.**

Why it matters

Construction begins most lifetimes; destruction or storage reuse ends them. A reference can dangle even when the numeric address still looks plausible, which is common after vector reallocation or scope exit.

Engineering rules

- Do not keep borrows across relocating mutations.
- Do not use an object after explicit destruction.
- Keep manual lifetime code local.
- Re-acquire pointers after invalidating operations.

RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. The valid-access window is narrower than the storage window. Do not keep borrows across relocating mutations. Do not use an object after explicit destruction.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```cpp
std::vector<int> v{1,2,3};
int* p = &v[0];
v.push_back(4);      // may reallocate
// *p = 9;           // potentially dangling
p = &v[0];           // re-acquire if still needed
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

The valid-access window is narrower than the storage window. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

For every long-lived borrow, identify the event that invalidates it.

Lifetime safety profile; container invalidation

DISCIPLINE CARD 2.3

Undefined Behavior Is a Safety Boundary

PRINCIPLE **Do not reason from observed results after UB.**

Why it matters

Optimizers may assume undefined behavior never occurs. Invalid memory access, null dereference, misalignment, data races, and invalid lifetime can therefore produce effects far from the faulty line.

Engineering rules

- Treat sanitizer findings as release blockers.
- Avoid representation tricks unless explicitly permitted.
- Keep casts narrow and justified.
- Never depend on a UB experiment that happened to work.

RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Do not reason from observed results after UB. Treat sanitizer findings as release blockers. Avoid representation tricks unless explicitly permitted.

REVIEW CHECKLIST

- Treat sanitizer findings as release blockers.
- Avoid representation tricks unless explicitly permitted.
- Keep casts narrow and justified.
- Never depend on a UB experiment that happened to work.

DESIGN CONSEQUENCE

Do not reason from observed results after UB. Make the rule visible in the type, API, scope, or build policy so a later refactor cannot silently remove it.

REVIEW GATE

A sanitizer report is evidence of a bug even when an unsanitized build appears stable.

Core Guidelines P.4; UBSan docs

DISCIPLINE CARD 2.4

Initialization Discipline

PRINCIPLE **Create objects in valid, deterministic states.**

Why it matters

Initialization at declaration, constructors that establish invariants, value initialization, and container-managed construction remove a large class of accidental uninitialized reads. Raw storage is different: allocated bytes are not automatically constructed objects.

Engineering rules

- Initialize at declaration.
- Use constructors to establish invariants.
- Do not let partially initialized objects escape.
- Run MemorySanitizer where supported.

RISK IF IGNORED

Typical failure: storage, construction, alignment, or deallocation contracts are mixed, causing leaks, fragmentation, invalid lifetimes, or mismatched release.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Create objects in valid, deterministic states. Initialize at declaration. Use constructors to establish invariants.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
struct Packet {
    std::uint32_t id{};
    std::array<std::byte,64> payload{};
};
Packet p{};
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Create objects in valid, deterministic states. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

A public type should not expose a accidental “not initialized yet” state.

Core Guidelines ES.20; MemorySanitizer

03

MEMORY DISCIPLINE CHAPTER

Ownership Architecture

Use types and interfaces to make ownership explicit, local, and mechanically reviewable.

DISCIPLINE CARD 3.1

Raw Pointers and References Are Non-Owning

PRINCIPLE **Treat T* and T& as observations unless a low-level owner annotation says otherwise.**

Why it matters

This convention removes ambiguity. Owning resources should live in values, containers, unique_ptr/shared_ptr, or custom RAII handles. References naturally express required borrowing; pointers can express nullable borrowing.

Engineering rules

- T& = borrowed, non-null.
- T* = borrowed, possibly null.
- unique_ptr = exclusive dynamic owner.
- shared_ptr = shared lifetime, not convenience.

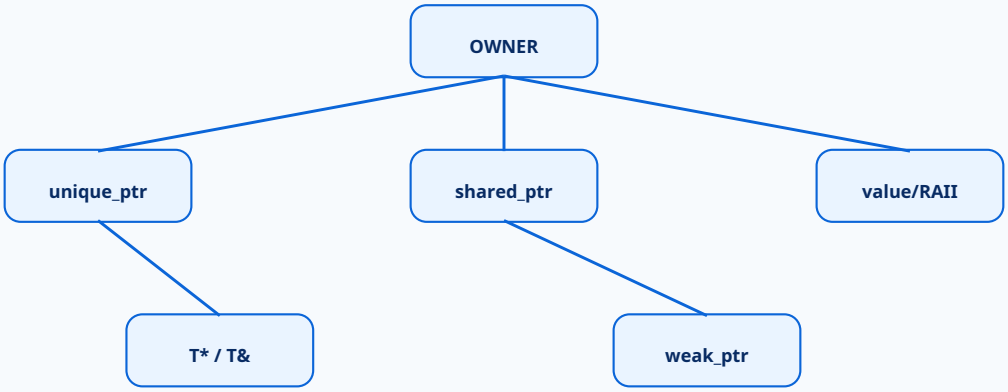
RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Treat T* and T& as observations unless a low-level owner annotation says otherwise. T& = borrowed, non-null. T* = borrowed, possibly null.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

Do not delete through a plain T* parameter received from another component.

Core Guidelines R.3-R.4, R.20

DISCIPLINE CARD 3.2

Return by Value When You Can

PRINCIPLE **Value semantics are the simplest ownership model.**

Why it matters

Return-value optimization and move semantics make value-returning factories efficient for many types. Dynamic allocation should be introduced for identity, polymorphism, stable address, lifetime decoupling, or allocator strategy—not habit.

Engineering rules

- Prefer T make_T() over T* make_T().
- Use unique_ptr when dynamic lifetime is part of the contract.
- Avoid “caller must delete” APIs.
- Measure before replacing values with shared ownership.

RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Value semantics are the simplest ownership model. Prefer T make_T() over T* make_T(). Use unique_ptr when dynamic lifetime is part of the contract.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
Widget make_widget(Config cfg) {
    Widget w{cfg};
    return w;
}
std::unique_ptr<Base> make_plugin();
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Value semantics are the simplest ownership model. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

A raw owning factory return requires a legacy/ABI justification.

Core Guidelines F.20, R.3

DISCIPLINE CARD 3.3

Ownership Transfer Must Be Visible

PRINCIPLE **Use owning parameter types only when ownership semantics matter.**

Why it matters

Passing unique_ptr by value says the callee consumes ownership. Passing unique_ptr& says the callee may reseat it. If the function only uses the pointee, accept T& or T*.

Engineering rules

- unique_ptr<T> by value consumes.
- unique_ptr<T>& reseats.
- T&/T* merely use.
- Do not take shared_ptr just to access the object.

RISK IF IGNORED

Typical failure: the code leaves ownership, lifetime, or cleanup implicit. Under error paths or future refactoring, that implicit assumption becomes a memory defect.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Use owning parameter types only when ownership semantics matter. unique_ptr<T> by value consumes. unique_ptr<T>& reseats.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
void install(std::unique_ptr<Device> d);
void reset(std::unique_ptr<Device>& d);
void ping(Device& d);
void maybe_ping(Device* d);
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Use owning parameter types only when ownership semantics matter. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

A smart-pointer parameter is justified only if the function uses its lifetime semantics.

Core Guidelines R.30-R.36

DISCIPLINE CARD 3.4

Ownership Graphs: Prefer Trees

PRINCIPLE **Exclusive ownership is naturally acyclic; shared ownership needs an explicit cycle policy.**

Why it matters

Use unique_ptr for parent-to-child ownership and raw/reference back-links when lifetime is nested. True shared lifetime creates a graph; weak_ptr should break cycles.

Engineering rules

- Start with unique_ptr.
- Name the independent owners before using shared_ptr.
- Use weak_ptr for non-owning edges in shared graphs.
- Document the root and teardown path.

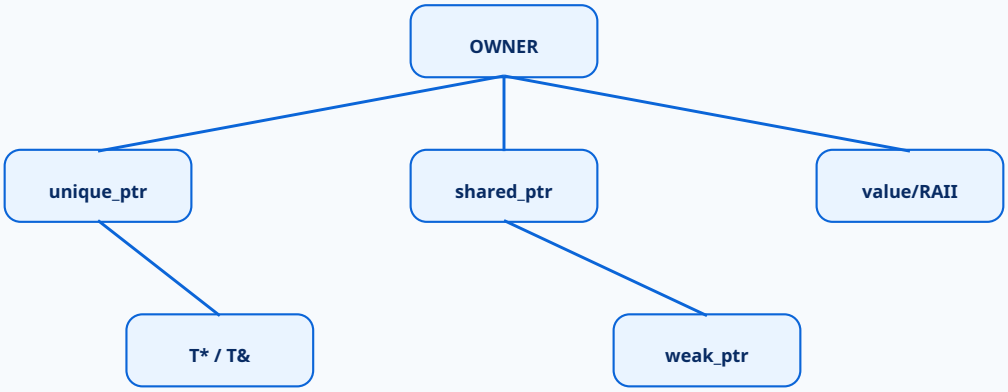
RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Exclusive ownership is naturally acyclic; shared ownership needs an explicit cycle policy. Start with unique_ptr. Name the independent owners before using shared_ptr.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

Every shared-ownership graph requires a cycle-breaking strategy.

Core Guidelines R.21, R.24

04

MEMORY DISCIPLINE CHAPTER

RAII, Rule of Zero, and Resource Handles

Turn cleanup from a control-flow obligation into a type invariant.

DISCIPLINE CARD 4.1

RAII Is the Primary Resource-Safety Mechanism

PRINCIPLE **Acquire a resource into an object whose destructor releases it.**

Why it matters

RAII handles normal return, early return, and exception unwinding uniformly. The pattern applies to memory, files, sockets, mutexes, mappings, GPU handles, and transactions.

Engineering rules

- Bind acquisition to a manager immediately.
- Prefer standard handles when available.
- Keep destructors non-throwing.
- Use RAII members so partial construction is safe.

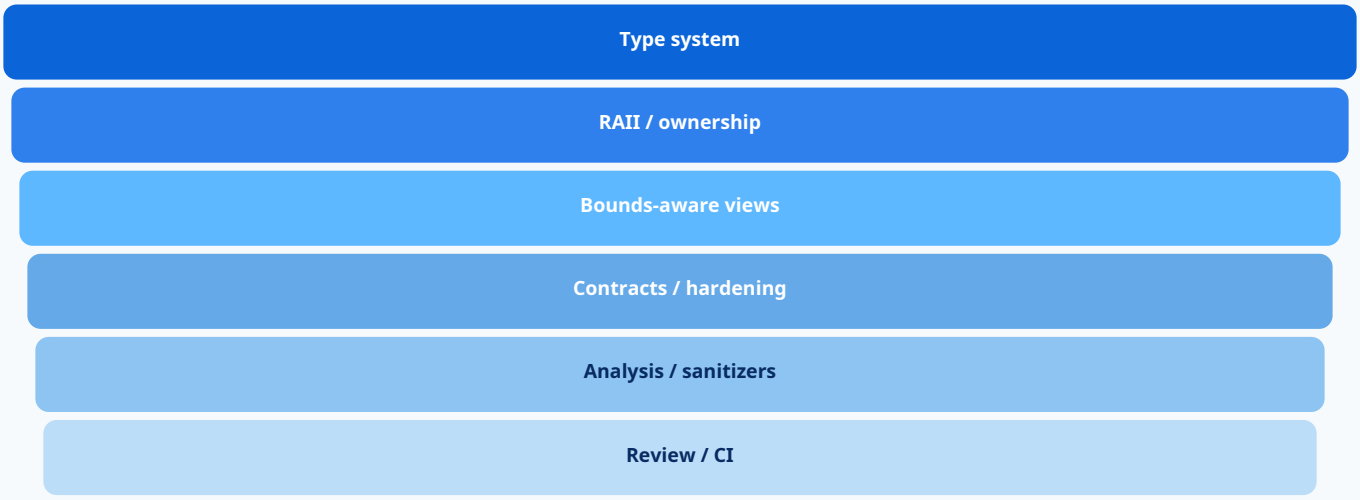
RISK IF IGNORED

Typical failure: an early exit leaves resources or invariants half-updated. The cure is to make rollback and cleanup lexical and automatic.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Acquire a resource into an object whose destructor releases it. Bind acquisition to a manager immediately. Prefer standard handles when available.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

If cleanup depends on remembering a matching call later, encapsulate the pair.

Core Guidelines R.1, R.12

DISCIPLINE CARD 4.2

Rule of Zero First

PRINCIPLE **Let members own resources so your class needs no special cleanup code.**

Why it matters

Values, containers, strings, smart pointers, and RAII wrappers compose correct behavior. A custom destructor/copy/move implementation is often a signal that a member should become a resource handle.

Engineering rules

- Prefer value-like members.
- Avoid owning raw pointer members.
- Use =default when it clarifies intent.
- Delete copy only when semantics require uniqueness.

RISK IF IGNORED

Typical failure: the code leaves ownership, lifetime, or cleanup implicit. Under error paths or future refactoring, that implicit assumption becomes a memory defect.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Let members own resources so your class needs no special cleanup code. Prefer value-like members. Avoid owning raw pointer members.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
class Session {
    std::string name_;
    std::vector<std::byte> buffer_;
    std::unique_ptr<Transport> transport_;
};
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Let members own resources so your class needs no special cleanup code. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

Before writing a destructor, ask which member should own the resource.

Core Guidelines C.20-C.21

DISCIPLINE CARD 4.3

Custom RAII Handles for C and OS Resources

PRINCIPLE **Wrap a foreign handle once; do not spread its release protocol through the codebase.**

Why it matters

A small move-only wrapper can encode the invalid sentinel, close operation, ownership transfer, and destructor. This is often safer than a smart pointer when the “pointer” is actually an integer or opaque token.

Engineering rules

- Destructor releases only when valid.
- Copy is deleted unless duplication is meaningful.
- Move transfers and invalidates the source.
- Expose raw handle only via explicit native_handle/get.

RISK IF IGNORED

Typical failure: ownership conventions differ across the boundary, causing leaks, double release, or exceptions escaping into an ABI that cannot handle them.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Wrap a foreign handle once; do not spread its release protocol through the codebase. Destructor releases only when valid. Copy is deleted unless duplication is meaningful.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
class File {
    std::FILE* f_{};
public:
    ~File(){ if(f_) std::fclose(f_); }
    File(const File&) = delete;
    File& operator=(const File&) = delete;
};
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Wrap a foreign handle once; do not spread its release protocol through the codebase. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

The raw handle must never outlive the wrapper that owns it.

Core Guidelines R.1

DISCIPLINE CARD 4.4

Move-Only Types Make Ownership Linear

PRINCIPLE **A move-only owner can have exactly one active owner at a time.**

Why it matters

unique_ptr shows the pattern. Custom resource handles should usually transfer ownership through noexcept moves and leave the source empty but destructible.

Engineering rules

- Delete copy for unique resources.
- Move = transfer + source invalidation.
- Moved-from state remains valid to destroy/assign.
- Declare noexcept only when true.

RISK IF IGNORED

Typical failure: the code leaves ownership, lifetime, or cleanup implicit. Under error paths or future refactoring, that implicit assumption becomes a memory defect.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. A move-only owner can have exactly one active owner at a time. Delete copy for unique resources. Move = transfer + source invalidation.

REVIEW CHECKLIST

- Delete copy for unique resources.
- Move = transfer + source invalidation.
- Moved-from state remains valid to destroy/assign.
- Declare noexcept only when true.

DESIGN CONSEQUENCE

A move-only owner can have exactly one active owner at a time. Make the rule visible in the type, API, scope, or build policy so a later refactor cannot silently remove it.

REVIEW GATE

A moved-from owner must be safe to destroy and reassign.

Core Guidelines C.64-C.66

PART 2

SAFE INTERFACES

Borrowing, views, containers, and smart pointers

2

05

MEMORY DISCIPLINE CHAPTER

Pointers, References, and Nullability

Make pointer meaning simple: single object, non-owning, and null only when the interface says null is valid.

DISCIPLINE CARD 5.1

Prefer References for Required Borrowing

PRINCIPLE **Use T& when a valid object is required.**

Why it matters

A reference removes null from the state space and communicates non-ownership. const T& is a strong default for read-only access to non-trivial objects when copying is unnecessary.

Engineering rules

- T& for required mutable borrow.
- const T& for required read-only borrow.
- Never return a reference to a local.
- Document lifetime coupling for returned subobject references.

RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Use T& when a valid object is required. T& for required mutable borrow. const T& for required read-only borrow.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
void update(Config& cfg);
void log(const Config& cfg);
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Use T& when a valid object is required. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

If callers must check null but null has no semantic meaning, redesign the parameter.

Core Guidelines R.4, F.44

DISCIPLINE CARD 5.2

Use Pointers for Nullable Borrowing

PRINCIPLE **A raw T* usually means maybe-object, borrowed.**

Why it matters

The callee should not delete it, infer an array, or keep it beyond the promised lifetime. If ownership is involved, use an owning type; if a sequence is involved, use a bounds-aware view.

Engineering rules

- Check null at the boundary.
- Do not infer array bounds from T*.
- Do not store borrows beyond owner lifetime.
- Avoid routine pointer arithmetic.

RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. A raw T* usually means maybe-object, borrowed. Check null at the boundary. Do not infer array bounds from T*.

REVIEW CHECKLIST

- Check null at the boundary.
- Do not infer array bounds from T*.
- Do not store borrows beyond owner lifetime.
- Avoid routine pointer arithmetic.

DESIGN CONSEQUENCE

A raw T* usually means maybe-object, borrowed. Make the rule visible in the type, API, scope, or build policy so a later refactor cannot silently remove it.

REVIEW GATE

A raw pointer parameter must not silently become an owner.

Core Guidelines R.2-R.3, ES.42

DISCIPLINE CARD 5.3

Encode Non-Null Requirements

PRINCIPLE **Use a reference or a not_null-style wrapper instead of comments that say “must not be null.”**

Why it matters

When pointer syntax is required but null is forbidden, an explicit non-null type makes the invariant visible and tool-friendly. C++26 contracts can reinforce the boundary, but they do not replace ownership types.

Engineering rules

- Prefer references where they fit.
- Use not_null when pointer behavior is needed.
- Contracts state conditions, not ownership.
- Remove redundant null checks after the invariant is established.

RISK IF IGNORED

Typical failure: an address travels without its valid extent, so an otherwise small indexing mistake becomes an out-of-bounds read or write.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Use a reference or a not_null-style wrapper instead of comments that say “must not be null.” Prefer references where they fit. Use not_null when pointer behavior is needed.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
void consume(gsl::not_null<const Packet*> p);
// C++26 can additionally add a precondition.
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++2c. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Use a reference or a not_null-style wrapper instead of comments that say “must not be null.” The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

Do not accept a nullable type and rely only on prose to forbid null.

GSL not_null; C++26 contracts

DISCIPLINE CARD 5.4

Ban Pointer Arithmetic in Application Code

PRINCIPLE **Use ranges, iterators, span, and indexes that retain bounds.**

Why it matters

Pointer arithmetic is legitimate inside audited codecs, containers, allocators, and drivers, but it should not be the default traversal technique in application code.

Engineering rules

- No p++ loops over unknown storage in product logic.
- Use span for contiguous ranges.
- Use iterators/ranges for generic traversal.
- Isolate unavoidable arithmetic in low-level helpers.

RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Use ranges, iterators, span, and indexes that retain bounds. No p++ loops over unknown storage in product logic. Use span for contiguous ranges.

REVIEW CHECKLIST

- No p++ loops over unknown storage in product logic.
- Use span for contiguous ranges.
- Use iterators/ranges for generic traversal.
- Isolate unavoidable arithmetic in low-level helpers.

DESIGN CONSEQUENCE

Use ranges, iterators, span, and indexes that retain bounds. Make the rule visible in the type, API, scope, or build policy so a later refactor cannot silently remove it.

REVIEW GATE

Raw pointer arithmetic outside infrastructure requires a specific review justification.

Core Guidelines ES.42; bounds profile

06

MEMORY DISCIPLINE CHAPTER

Views, span, string_view, and Borrowed Ranges

Views improve bounds and copying costs only when their lifetime relationship is understood.

DISCIPLINE CARD 6.1

std::span Keeps Address and Extent Together

PRINCIPLE **Pass contiguous sequences as a view that carries size.**

Why it matters

span removes the pointer-plus-length split and makes subranges explicit. It is still non-owning, so the source must remain alive and structurally valid for the whole use interval.

Engineering rules

- Prefer span<T> over T* + size.
- Use span<const T> for read-only sequences.
- Do not let a span outlive its storage.
- Use subspan rather than pointer arithmetic.

RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Pass contiguous sequences as a view that carries size. Prefer span<T> over T* + size. Use span<const T> for read-only sequences.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
void checksum(std::span<const std::byte> bytes);
std::array<std::byte,128> buf{};
checksum(buf);
checksum(std::span{buf}.first(32));
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Pass contiguous sequences as a view that carries size. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

A span solves bounds representation, not ownership. Both must be correct.

Core Guidelines R.14, ES.42

DISCIPLINE CARD 6.2

std::string_view Is Borrowed Text

PRINCIPLE **string_view never extends character lifetime.**

Why it matters

It is excellent for read-only parameters and parsing, but dangerous as a long-lived member unless an owner with longer lifetime is guaranteed. A view into a temporary string can dangle immediately.

Engineering rules

- Use string_view for non-owning parameters.
- Store string when ownership is needed.
- Re-acquire after mutations that may reallocate.
- Audit stored views for owner lifetime.

RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. string_view never extends character lifetime. Use string_view for non-owning parameters. Store string when ownership is needed.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
void parse(std::string_view text);
std::string owned = make_name();
std::string_view v = owned;
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

string_view never extends character lifetime. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

A stored string_view requires a documented, longer-lived owner.

string_view lifetime rules

DISCIPLINE CARD 6.3

Borrowed Ranges Carry Lifetime Information

PRINCIPLE **Do not assume iterators from temporaries remain valid.**

Why it matters

The ranges model uses borrowed_range and dangling to prevent some unsafe iterator results from temporary ranges. The same idea should guide your APIs: do not return hidden references into objects that can already be dead.

Engineering rules

- Understand whether a view owns, caches, or borrows.
- Avoid storing iterators across mutation.
- Prefer value results when iterator lifetime is fragile.
- Constrain APIs that could return into temporaries.

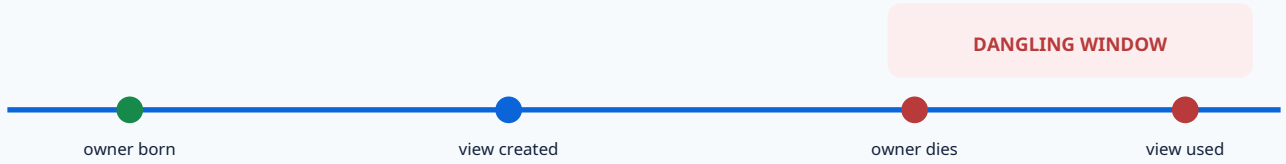
RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Do not assume iterators from temporaries remain valid. Understand whether a view owns, caches, or borrows. Avoid storing iterators across mutation.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

If an API can return into a dead temporary, redesign or constrain it.

C++ ranges lifetime model

DISCIPLINE CARD 6.4

Draw the Borrow Timeline

PRINCIPLE **When lifetime is unclear, draw owner birth, view creation, invalidation, and last use.**

Why it matters

A simple timeline exposes stale iterators, vector pointers, substrings, mapped-region views, and C API buffers. The unsafe interval begins when the owner dies or structurally invalidates the view.

Engineering rules

- Mark owner birth/death.
- Mark invalidating mutations.
- Mark borrow creation/last use.
- Shorten the borrow interval when possible.

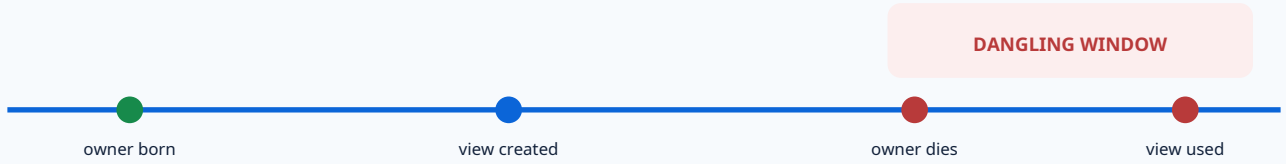
RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. When lifetime is unclear, draw owner birth, view creation, invalidation, and last use. Mark owner birth/death. Mark invalidating mutations.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

The owner must remain alive and structurally valid for the entire observation interval.

Core Guidelines lifetime profile

07

MEMORY DISCIPLINE CHAPTER

Containers, Bounds, and Invalidation

Standard containers remove manual allocation, but safe use still requires bounds and invalidation discipline.

DISCIPLINE CARD 7.1

Know the Invalidation Contract

PRINCIPLE **Container mutation can invalidate pointers, references, and iterators.**

Why it matters

vector reallocation invalidates all element addresses; erase invalidates positions at/after the erased element. Node-based containers have different contracts. The safe rule is to check the actual container guarantees instead of relying on folklore.

Engineering rules

- Do not keep vector element pointers across growth.
- Use stable IDs or indexes when appropriate.
- Re-acquire after invalidating mutation.
- Choose a container whose stability fits the algorithm.

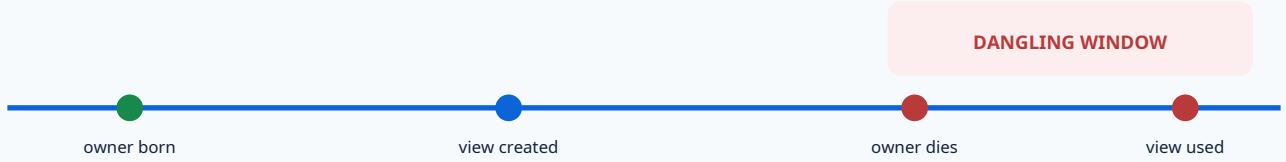
RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Container mutation can invalidate pointers, references, and iterators. Do not keep vector element pointers across growth. Use stable IDs or indexes when appropriate.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

Any stored iterator/reference must list the mutations that invalidate it.

C++ container invalidation rules

DISCIPLINE CARD 7.2

Bounds: Prefer Checkable APIs

PRINCIPLE **Out-of-range access is a memory-safety failure.**

Why it matters

Use algorithms and range-for to reduce explicit indexing. Use .at() where runtime checking is appropriate. C++26 hardened libraries add defense in depth for selected preconditions, but unchecked access is still an invalid call if its precondition is violated.

Engineering rules

- Validate untrusted indexes.
- Use .at() at trust boundaries where appropriate.
- Prefer algorithms that avoid manual indexes.
- Do not remove checks before measuring.

RISK IF IGNORED

Typical failure: an address travels without its valid extent, so an otherwise small indexing mistake becomes an out-of-bounds read or write.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Out-of-range access is a memory-safety failure. Validate untrusted indexes. Use .at() at trust boundaries where appropriate.

REVIEW CHECKLIST

- Validate untrusted indexes.
- Use .at() at trust boundaries where appropriate.
- Prefer algorithms that avoid manual indexes.
- Do not remove checks before measuring.

DESIGN CONSEQUENCE

Out-of-range access is a memory-safety failure. Make the rule visible in the type, API, scope, or build policy so a later refactor cannot silently remove it.

REVIEW GATE

Externally derived indexes must be validated before unchecked access.

C++26 library hardening; bounds profile

DISCIPLINE CARD 7.3

reserve Controls Growth, Not Correctness

PRINCIPLE **reserve can reduce reallocations but does not prove references stay valid forever.**

Why it matters

If correctness depends on capacity never being exceeded, encode the maximum or choose a fixed-capacity type. reserve is a performance/stability window, not a lifetime guarantee.

Engineering rules

- Reserve from known/estimated growth.
- Do not store long-lived pointers just because reserve ran.
- Use fixed capacity for hard bounds.
- Treat rebuild/shrink operations as invalidation points.

RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. reserve can reduce reallocations but does not prove references stay valid forever. Reserve from known/estimated growth. Do not store long-lived pointers just because reserve ran.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
std::vector<Event> events;
events.reserve(expected_max);
for (auto&& e : input) events.push_back(e);
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

reserve can reduce reallocations but does not prove references stay valid forever. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

If capacity overflow would break correctness, capacity must be a checked invariant.

std::vector capacity rules

DISCIPLINE CARD 7.4

C++26 std::inplace_vector

PRINCIPLE **Use dynamic size with fixed capacity when heap allocation is undesirable and a maximum is meaningful.**

Why it matters

inplace_vector stores elements in the object, remains contiguous, and has compile-time capacity. It fits embedded, real-time, packet, small-buffer, and low-latency designs.

Engineering rules

- Pick capacity from system bounds.
- Handle exhaustion explicitly.
- Remember insert/erase can still invalidate positions.
- Prefer it over manual array + size bookkeeping.

RISK IF IGNORED

Typical failure: an address travels without its valid extent, so an otherwise small indexing mistake becomes an out-of-bounds read or write.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Use dynamic size with fixed capacity when heap allocation is undesirable and a maximum is meaningful. Pick capacity from system bounds. Handle exhaustion explicitly.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
#include <inplace_vector>
std::inplace_vector<Message,64> q;
if (q.try_push_back(msg) == nullptr) handle_overflow();
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++2c. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Use dynamic size with fixed capacity when heap allocation is undesirable and a maximum is meaningful. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

The capacity is part of the system contract, not a magic number.

C++26 std::inplace_vector

08

MEMORY DISCIPLINE CHAPTER

Smart Pointers: Deep Ownership Semantics

Use smart pointers to encode ownership precisely; do not use them as a generic replacement for every raw pointer.

DISCIPLINE CARD 8.1

unique_ptr Is the Default Dynamic Owner

PRINCIPLE **Exclusive ownership is simple, deterministic, and low-overhead.**

Why it matters

unique_ptr destroys its object with the owner and can carry a custom deleter. Prefer make_unique for ordinary objects. Use unique_ptr<T[]> only when a dynamic array is truly the right abstraction; vector is usually clearer.

Engineering rules

- Prefer make_unique.
- Pass by value to transfer ownership.
- Use custom deleters for matching foreign release APIs.
- release() must immediately transfer ownership elsewhere.

RISK IF IGNORED

Typical failure: an address travels without its valid extent, so an otherwise small indexing mistake becomes an out-of-bounds read or write.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Exclusive ownership is simple, deterministic, and low-overhead. Prefer make_unique. Pass by value to transfer ownership.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
auto p = std::make_unique<Job>(cfg);
run(*p);
submit(std::move(p));
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Exclusive ownership is simple, deterministic, and low-overhead. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

A released raw pointer must acquire a new owner in the same narrow operation.

Core Guidelines R.20-R.23

DISCIPLINE CARD 8.2

shared_ptr: Understand the Control Block

PRINCIPLE **Shared ownership has a real runtime and semantic cost.**

Why it matters

The control block tracks strong/weak ownership and holds deleter/allocator metadata. make_shared often co-allocates object and control block, reducing allocations but changing retention characteristics.

Engineering rules

- Use shared_ptr only for shared lifetime.
- Prefer make_shared when its trade-offs fit.
- Borrow T&/T* when ownership is not needed.
- Measure reference-count traffic in hot paths.

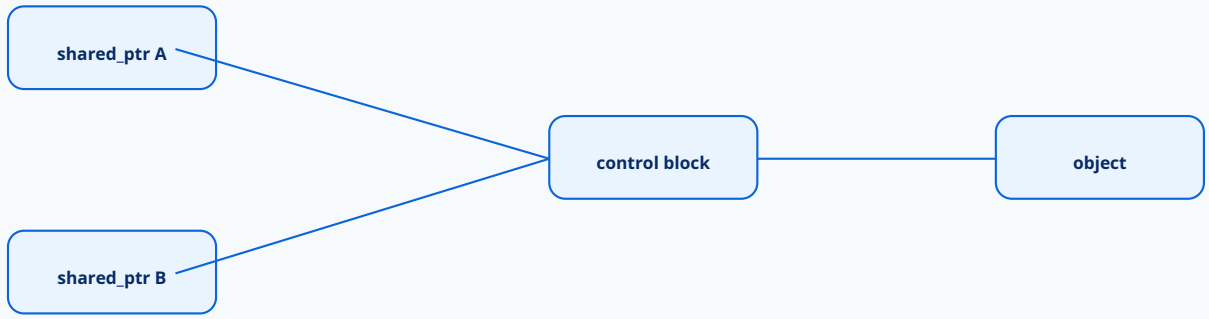
RISK IF IGNORED

Typical failure: ownership is wider than intended, reference cycles keep resources alive, or shared lifetime is confused with mere access.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Shared ownership has a real runtime and semantic cost. Use shared_ptr only for shared lifetime. Prefer make_shared when its trade-offs fit.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

Before adding shared_ptr, name the independent owners that require shared lifetime.

Core Guidelines R.21-R.22

DISCIPLINE CARD 8.3

weak_ptr Breaks Shared Cycles

PRINCIPLE **Observe a shared object without extending its lifetime.**

Why it matters

weak_ptr is the normal back-edge in shared graphs. lock() obtains a temporary shared owner. enable_shared_from_this is safe only when the object already participates in a shared control block; never construct shared_ptr(this).

Engineering rules

- Use weak_ptr for non-owning shared-graph edges.
- Handle expiration after lock().
- Never write shared_ptr(this).
- Use shared_from_this only when self-lifetime sharing is real.

RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Observe a shared object without extending its lifetime. Use weak_ptr for non-owning shared-graph edges. Handle expiration after lock().

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
if (auto p = weak_owner.lock()) p->notify();
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Observe a shared object without extending its lifetime. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

A graph with shared_ptr edges must be reviewed for cycles.

Core Guidelines R.24

DISCIPLINE CARD 8.4

C++26 Owner Identity

PRINCIPLE **Use owner_hash/owner_equal when the key is shared ownership identity, not raw address.**

Why it matters

Two shared_ptr values can point at different subobjects while sharing one control block. C++26 owner-based hashing/equality lets unordered containers key by that lifetime identity.

Engineering rules

- Choose address identity or owner identity deliberately.
- Use owner_hash and owner_equal together.
- Do not mix relations in one container.
- Guard partial C++26 support with feature macros.

RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Use owner_hash/owner_equal when the key is shared ownership identity, not raw address. Choose address identity or owner identity deliberately. Use owner_hash and owner_equal together.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
#if __cpp_lib_smart_ptr_owner_equality >= 202306L
using Set = std::unordered_set<std::shared_ptr<Node>,
  std::owner_hash, std::owner_equal>;
#endif
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++2c. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Use owner_hash/owner_equal when the key is shared ownership identity, not raw address. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

The chosen equality relation must match the data structure's semantic identity.

C++26 owner_hash/owner_equal

PART 3

ALLOCATION & LOW-LEVEL MEMORY

Allocation strategy, PMR, arenas, lifetime, and alignment

3

09

MEMORY DISCIPLINE CHAPTER

Allocation Discipline and Failure

Treat allocation as an architectural choice, not an incidental expression scattered across code.

DISCIPLINE CARD 9.1

Avoid Naked new/delete in Application Code

PRINCIPLE **An explicit allocation result should immediately enter a manager.**

Why it matters

Containers, make_unique/make_shared, and RAII handles make leaks and exceptional exits much harder. Explicit new/delete belongs mainly inside allocators, resource managers, and low-level infrastructure.

Engineering rules

- Ban naked new/delete in product logic.
- Never mix malloc/free with new/delete pairs.
- Prefer scoped objects when practical.
- Keep explicit allocation inside resource-management components.

RISK IF IGNORED

Typical failure: storage, construction, alignment, or deallocation contracts are mixed, causing leaks, fragmentation, invalid lifetimes, or mismatched release.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. An explicit allocation result should immediately enter a manager. Ban naked new/delete in product logic. Never mix malloc/free with new/delete pairs.

REVIEW CHECKLIST

- Ban naked new/delete in product logic.
- Never mix malloc/free with new/delete pairs.
- Prefer scoped objects when practical.
- Keep explicit allocation inside resource-management components.

DESIGN CONSEQUENCE

An explicit allocation result should immediately enter a manager. Make the rule visible in the type, API, scope, or build policy so a later refactor cannot silently remove it.

REVIEW GATE

Every explicit allocation site must live in a component whose job is resource management.

Core Guidelines R.5, R.10-R.13

DISCIPLINE CARD 9.2

Placement Construction Is Not Allocation

PRINCIPLE **Separate raw storage from construction of an object in that storage.**

Why it matters

Low-level containers and arenas may obtain storage once and then construct/destroy objects within it. construct_at/destroy_at make this intent clearer, but alignment, exception safety, and live-slot tracking remain your responsibility.

Engineering rules

- Use only in infrastructure that owns raw storage.
- Track which slots contain live objects.
- Destroy exactly what was constructed.
- Never expose unconstructed storage as a live T.

RISK IF IGNORED

Typical failure: storage, construction, alignment, or deallocation contracts are mixed, causing leaks, fragmentation, invalid lifetimes, or mismatched release.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Separate raw storage from construction of an object in that storage. Use only in infrastructure that owns raw storage. Track which slots contain live objects.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
T* p = std::allocator_traits<Alloc>::allocate(a,1);
try {
  std::construct_at(p,args...);
  std::destroy_at(p);
  std::allocator_traits<Alloc>::deallocate(a,p,1);
} catch(...) {
  std::allocator_traits<Alloc>::deallocate(a,p,1);
  throw;
}
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Separate raw storage from construction of an object in that storage. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

A raw-storage component must have an explicit live-slot invariant.

construct_at/destroy_at; allocator model

DISCIPLINE CARD 9.3

C++23 allocate_at_least

PRINCIPLE **Let an allocator report useful extra capacity when it can.**

Why it matters

allocate_at_least returns storage for at least the requested count and reports the actual count. It is primarily a low-level container/allocator optimization that can exploit allocator size classes and reduce reallocations.

Engineering rules

- Use through allocator-aware code.
- Store the returned count.
- Construct only the objects whose lifetimes should begin.
- Deallocate according to the API contract.

RISK IF IGNORED

Typical failure: storage, construction, alignment, or deallocation contracts are mixed, causing leaks, fragmentation, invalid lifetimes, or mismatched release.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Let an allocator report useful extra capacity when it can. Use through allocator-aware code. Store the returned count.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
std::allocator<T> a;
auto r = a.allocate_at_least(n); // r.count >= n
// construct selected objects in r.ptr ...
a.deallocate(r.ptr, r.count);
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Let an allocator report useful extra capacity when it can. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

If you cannot preserve the returned count and lifetime state, do not use allocate_at_least.

C++23 allocate_at_least

DISCIPLINE CARD 9.4

Allocation Failure Is Part of the Contract

PRINCIPLE **Decide whether failure propagates, degrades, retries, or is prevented by design.**

Why it matters

General-purpose code often allows bad_alloc to reach a recovery boundary. Real-time/embedded code may preallocate. nothrow new only changes reporting to nullptr; it does not solve resource exhaustion.

Engineering rules

- Define failure policy per subsystem.
- Do not catch bad_alloc just to continue with broken invariants.
- Preallocate for hard real-time phases.
- Log context at recovery boundaries.

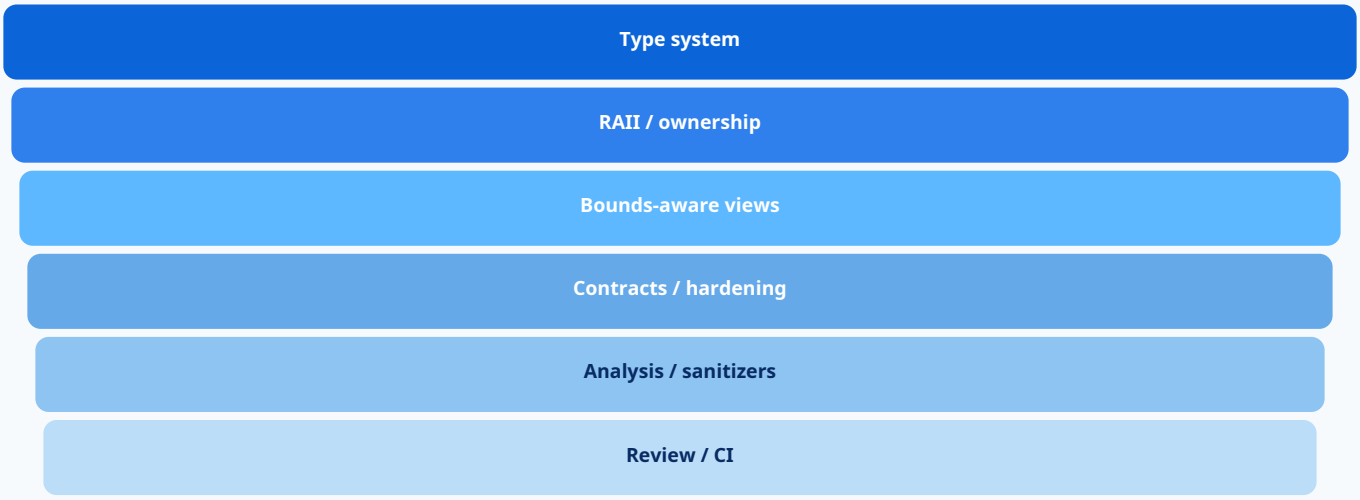
RISK IF IGNORED

Typical failure: an address travels without its valid extent, so an otherwise small indexing mistake becomes an out-of-bounds read or write.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Decide whether failure propagates, degrades, retries, or is prevented by design. Define failure policy per subsystem. Do not catch bad_alloc just to continue with broken invariants.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

No allocation site may have an implicit or contradictory failure policy.

C++ allocation semantics

10

MEMORY DISCIPLINE CHAPTER

Allocators and std::pmr Memory Resources

Move allocation strategy behind a controlled interface so data structures can remain generic.

DISCIPLINE CARD 10.1

Allocator-Aware Design Needs a Reason

PRINCIPLE **Custom allocation is justified by measured locality, determinism, bulk release, instrumentation, or integration needs.**

Why it matters

Allocator behavior affects propagation on copy/move/swap and can subtly change container behavior. Most applications should reach for standard containers and pmr before inventing a new allocator type.

Engineering rules

- Measure allocation frequency and size classes first.
- Use allocator_traits in generic low-level code.
- Test unequal-allocator copy/move/swap cases.
- Do not optimize allocation by intuition alone.

RISK IF IGNORED

Typical failure: storage, construction, alignment, or deallocation contracts are mixed, causing leaks, fragmentation, invalid lifetimes, or mismatched release.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Custom allocation is justified by measured locality, determinism, bulk release, instrumentation, or integration needs. Measure allocation frequency and size classes first. Use allocator_traits in generic low-level code.

REVIEW CHECKLIST

- Measure allocation frequency and size classes first.
- Use allocator_traits in generic low-level code.
- Test unequal-allocator copy/move/swap cases.
- Do not optimize allocation by intuition alone.

DESIGN CONSEQUENCE

Custom allocation is justified by measured locality, determinism, bulk release, instrumentation, or integration needs. Make the rule visible in the type, API, scope, or build policy so a later refactor cannot silently remove it.

REVIEW GATE

A custom allocator must state the measured problem it solves.

std::allocator_traits; Core Guidelines P.9

DISCIPLINE CARD 10.2

std::pmr Decouples Container Logic from Allocation Strategy

PRINCIPLE **A polymorphic_allocator lets related objects share a runtime memory policy.**

Why it matters

pmr is valuable at subsystem boundaries where many containers/objects should draw from one resource. The central safety rule is lifetime: the memory_resource must outlive every allocation and every container that may deallocate through it.

Engineering rules

- Resource lifetime must dominate client lifetime.
- Pass memory_resource* explicitly at construction boundaries.
- Avoid hidden global default-resource changes.
- Use null_memory_resource when fallback is forbidden.

RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. A polymorphic_allocator lets related objects share a runtime memory policy. Resource lifetime must dominate client lifetime. Pass memory_resource* explicitly at construction boundaries.

MODEL / VISUAL

HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

A pmr container must never outlive its memory_resource.

std::pmr memory_resource/polymorphic_allocator

DISCIPLINE CARD 10.3

monotonic_buffer_resource: Fast Phase Allocation

PRINCIPLE **Use monotonic allocation when many objects die together.**

Why it matters

Individual deallocations are ignored; memory is reclaimed when the resource is released/destroyed. It fits parse trees, frame scratch memory, requests, compilers, and batch construction.

Engineering rules

- Match resource lifetime to the phase.
- Use a local initial buffer for predictable small workloads.
- Choose the upstream resource deliberately.
- Do not mix unrelated object lifetimes in one monotonic arena.

RISK IF IGNORED

Typical failure: an address travels without its valid extent, so an otherwise small indexing mistake becomes an out-of-bounds read or write.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Use monotonic allocation when many objects die together. Match resource lifetime to the phase. Use a local initial buffer for predictable small workloads.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
std::byte buf[16*1024];
std::pmr::monotonic_buffer_resource arena{buf,sizeof buf};
std::pmr::vector<Node> nodes{&arena};
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Use monotonic allocation when many objects die together. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

Every allocation from a monotonic resource must belong to the same logical release phase.

std::pmr::monotonic_buffer_resource

DISCIPLINE CARD 10.4

Pool Resources: Size-Class Reuse

PRINCIPLE **Use pools for repeated allocations of similar sizes.**

Why it matters

unsynchronized_pool_resource avoids locking for thread-confined use; synchronized_pool_resource supports shared use. Both can retain blocks for reuse and delegate large requests upstream.

Engineering rules

- Use unsynchronized pools only when truly thread-confined.
- Use synchronized pools for shared access.
- Tune pool_options after profiling.
- Remember the pool resource owns retained storage.

RISK IF IGNORED

Typical failure: storage, construction, alignment, or deallocation contracts are mixed, causing leaks, fragmentation, invalid lifetimes, or mismatched release.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Use pools for repeated allocations of similar sizes. Use unsynchronized pools only when truly thread-confined. Use synchronized pools for shared access.

MODEL / VISUAL

HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

Synchronization policy follows actual sharing, not fear.

std::pmr pool resources

11

MEMORY DISCIPLINE CHAPTER

Arenas, Pools, Slabs, and Reclamation Patterns

Use specialized memory strategies when object lifetime topology is simpler than general-purpose heap lifetime.

DISCIPLINE CARD 11.1

Arena Allocation: One Lifetime Domain

PRINCIPLE **Trade per-object deallocation for fast allocation and one bulk reset.**

Why it matters

Arenas reduce allocator metadata traffic and often improve locality when many allocations share a phase lifetime. The danger is escape: any pointer/view that survives reset becomes invalid. Non-trivial destructors also require deliberate handling.

Engineering rules

- Bind arena to request/frame/phase.
- Do not let arena-backed views escape.
- Track destructors when needed.
- Instrument peak usage and fallback.

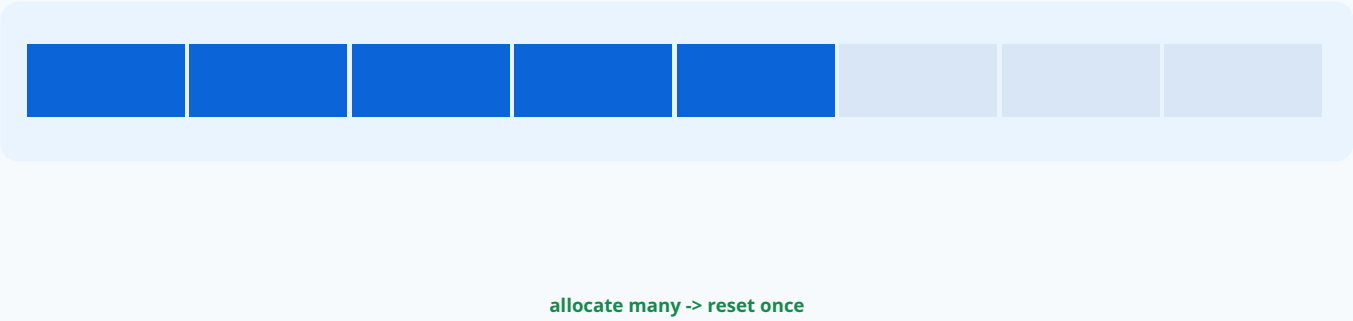
RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Trade per-object deallocation for fast allocation and one bulk reset. Bind arena to request/frame/phase. Do not let arena-backed views escape.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

No arena-backed pointer may cross the arena reset boundary.

Arena pattern; monotonic_buffer_resource

DISCIPLINE CARD 11.2

Fixed-Block Pools: Determinism Through Uniformity

PRINCIPLE **Preallocate equal-size slots when object size and maximum occupancy are known.**

Why it matters

Fixed pools can deliver predictable constant-time-ish allocation with low overhead and controlled fragmentation. They fit embedded queues, packets, entities, and control objects.

Engineering rules

- Validate ownership on debug deallocation.
- Test over-aligned types.
- Use generation counters when stale handles are possible.
- Define pool exhaustion behavior.

RISK IF IGNORED

Typical failure: storage, construction, alignment, or deallocation contracts are mixed, causing leaks, fragmentation, invalid lifetimes, or mismatched release.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Preallocate equal-size slots when object size and maximum occupancy are known. Validate ownership on debug deallocation. Test over-aligned types.

REVIEW CHECKLIST

- Validate ownership on debug deallocation.
- Test over-aligned types.
- Use generation counters when stale handles are possible.
- Define pool exhaustion behavior.

DESIGN CONSEQUENCE

Preallocate equal-size slots when object size and maximum occupancy are known. Make the rule visible in the type, API, scope, or build policy so a later refactor cannot silently remove it.

REVIEW GATE

Pool exhaustion is a tested state, not an impossible assumption.

Memory pool design

DISCIPLINE CARD 11.3

Free Lists Are Easy to Corrupt

PRINCIPLE **A fast free list can turn a double free into allocator metadata corruption.**

Why it matters

When free blocks carry linkage, invalid insertion, double return, or concurrent reclamation bugs can poison the structure. Debug builds should add range checks, state tags, canaries, or shadow metadata.

Engineering rules

- Never expose free-list internals to untrusted code.
- Detect double return in instrumented builds.
- Separate atomic unlink from safe reclamation.
- Fuzz allocate/free/reuse sequences.

RISK IF IGNORED

Typical failure: storage, construction, alignment, or deallocation contracts are mixed, causing leaks, fragmentation, invalid lifetimes, or mismatched release.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. A fast free list can turn a double free into allocator metadata corruption. Never expose free-list internals to untrusted code. Detect double return in instrumented builds.

REVIEW CHECKLIST

- Never expose free-list internals to untrusted code.
- Detect double return in instrumented builds.
- Separate atomic unlink from safe reclamation.
- Fuzz allocate/free/reuse sequences.

DESIGN CONSEQUENCE

A fast free list can turn a double free into allocator metadata corruption. Make the rule visible in the type, API, scope, or build policy so a later refactor cannot silently remove it.

REVIEW GATE

A lock-free free list is not correct until reclamation is proven.

Allocator engineering; C++26 reclamation

DISCIPLINE CARD 11.4

Memory Strategy Needs Telemetry

PRINCIPLE **Optimization without allocation telemetry is guesswork.**

Why it matters

At minimum record allocation count, bytes, peak live bytes, arena high-water mark, pool high-water mark, fallback count, and size distribution. Correlate with latency in performance-sensitive systems.

Engineering rules

- Measure before and after custom strategies.
- Expose high-water marks in tests.
- Fail “no allocation” tests when they allocate.
- Keep production telemetry low-overhead.

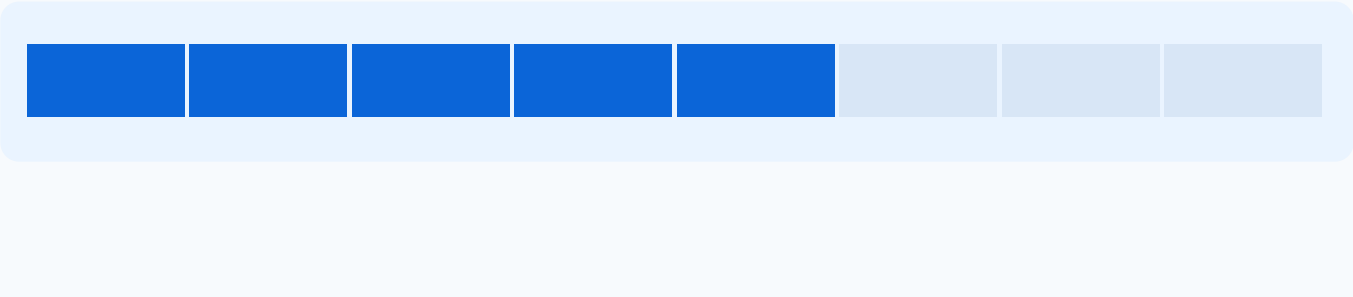
RISK IF IGNORED

Typical failure: storage, construction, alignment, or deallocation contracts are mixed, causing leaks, fragmentation, invalid lifetimes, or mismatched release.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Optimization without allocation telemetry is guesswork. Measure before and after custom strategies. Expose high-water marks in tests.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

Every custom allocator ships with allocation-count and peak-usage observability.

Performance engineering

12

MEMORY DISCIPLINE CHAPTER

Explicit Lifetime, Representation, and Alignment

Low-level memory code must obey lifetime, aliasing, alignment, and representation rules simultaneously.

DISCIPLINE CARD 12.1

C++23 std::start_lifetime_as

PRINCIPLE **Use explicit lifetime-start utilities only when suitable storage intentionally becomes an implicit-lifetime object.**

Why it matters

These APIs target low-level serialization, shared memory, allocators, and similar infrastructure. They are not a replacement for normal construction and have strict type/representation requirements.

Engineering rules

- Check type requirements.
- Ensure storage size and alignment.
- Use only when representation semantics are understood.
- Prefer normal construction whenever practical.

RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Use explicit lifetime-start utilities only when suitable storage intentionally becomes an implicit-lifetime object. Check type requirements. Ensure storage size and alignment.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
alignas(Header) std::array<std::byte,sizeof(Header)> bytes{};
Header* h = std::start_lifetime_as<Header>(bytes.data());
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Use explicit lifetime-start utilities only when suitable storage intentionally becomes an implicit-lifetime object. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

Every start_lifetime_as call needs a comment explaining why ordinary construction is not used.

C++23 std::start_lifetime_as

DISCIPLINE CARD 12.2

Use bit_cast/memcpy for Representation Transfers

PRINCIPLE **Do not use reinterpret_cast when the real intent is to copy object representation.**

Why it matters

bit_cast provides a constrained same-size value representation conversion for trivially copyable types. memcpy is the traditional byte-copy primitive. Native layout is still not automatically a portable wire format because of padding and endianness.

Engineering rules

- Use bit_cast for representation-preserving value conversion.
- Use explicit serialization for protocols.
- Do not assume struct padding is protocol layout.
- Avoid type-punning through unrelated pointers.

RISK IF IGNORED

Typical failure: the code leaves ownership, lifetime, or cleanup implicit. Under error paths or future refactoring, that implicit assumption becomes a memory defect.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Do not use reinterpret_cast when the real intent is to copy object representation. Use bit_cast for representation-preserving value conversion. Use explicit serialization for protocols.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```cpp
std::uint32_t bits = 0x3f800000u;
float one = std::bit_cast<float>(bits);
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Do not use reinterpret_cast when the real intent is to copy object representation. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

Native object representation is not a portable format unless the format defines it.

std::bit_cast; object representation

DISCIPLINE CARD 12.3

Alignment Is a Precondition

PRINCIPLE **Misaligned object access can be undefined even if hardware can physically perform it.**

Why it matters

Use alignas, aligned allocation, typed storage, and std::align. assume_aligned is an optimization promise and becomes dangerous if its precondition is false. C++26 is_sufficiently_aligned can help test a pointer before such an assumption.

Engineering rules

- Preserve alignment in allocator interfaces.
- Test over-aligned types.
- Check before assume_aligned.
- Do not assume arbitrary byte buffers satisfy T alignment.

RISK IF IGNORED

Typical failure: storage, construction, alignment, or deallocation contracts are mixed, causing leaks, fragmentation, invalid lifetimes, or mismatched release.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Misaligned object access can be undefined even if hardware can physically perform it. Preserve alignment in allocator interfaces. Test over-aligned types.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
#if __cpp_lib_is_sufficiently_aligned >= 202411L
if (std::is_sufficiently_aligned<64>(p))
    use_simd(std::assume_aligned<64>(p));
#endif
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++2c. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Misaligned object access can be undefined even if hardware can physically perform it. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

Every custom allocator must be tested with over-aligned types.

C++26 is_sufficiently_aligned

DISCIPLINE CARD 12.4

C++26 std::is_within_lifetime

PRINCIPLE **Treat it as a specialized lifetime utility, not a universal dangling-pointer detector.**

Why it matters

is_within_lifetime is designed for narrow lifetime queries with important constant-evaluation context. General runtime lifetime safety still comes from ownership architecture, analysis, and sanitizers.

Engineering rules

- Do not use it as general runtime lifetime tracking.
- Keep lifetime rules simple enough for tools.
- Prefer designs with unambiguous ownership.
- Use the facility only where its specified context applies.

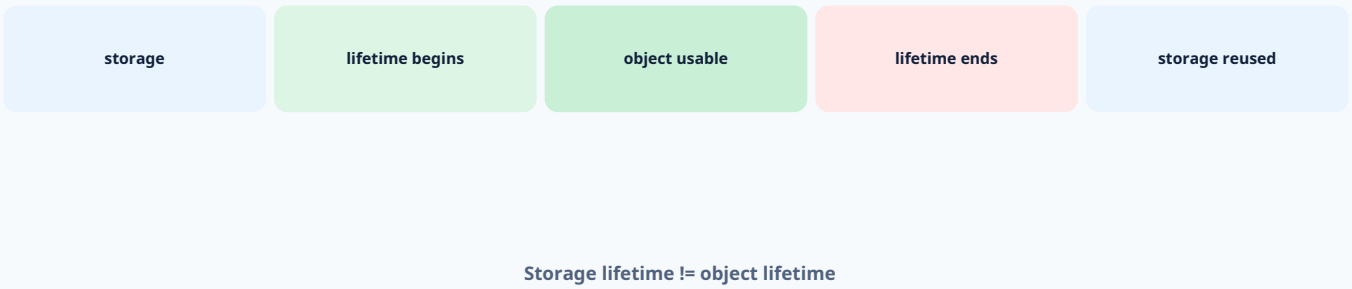
RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Treat it as a specialized lifetime utility, not a universal dangling-pointer detector. Do not use it as general runtime lifetime tracking. Keep lifetime rules simple enough for tools.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

If you need pervasive lifetime probes, redesign ownership instead.

C++26 is_within_lifetime

PART 4

INTEROP & CONCURRENCY

C boundaries, exception safety, threads, and atomics

4

13

MEMORY DISCIPLINE CHAPTER

C Interop and C++23 Smart-Pointer Adaptors

Foreign APIs often expose T** or void**; bridge them without abandoning RAII.

DISCIPLINE CARD 13.1

C++23 std::out_ptr Bridges T** Output

PRINCIPLE **Adapt a smart owner to a C API that writes a newly created pointer.**

Why it matters

out_ptr creates a temporary adaptor that receives the foreign pointer and resets the smart pointer at the end of the full expression. It removes the manual raw-temp + reset sequence.

Engineering rules

- Use the correct foreign deleter.
- Keep the adaptor temporary.
- Check the C function success contract.
- Do not assume a shared_ptr can reset without the right deleter.

RISK IF IGNORED

Typical failure: ownership conventions differ across the boundary, causing leaks, double release, or exceptions escaping into an ABI that cannot handle them.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Adapt a smart owner to a C API that writes a newly created pointer. Use the correct foreign deleter. Keep the adaptor temporary.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
using CPtr = std::unique_ptr<c_obj,CDeleter>;
CPtr p;
if (int ec = c_create(std::out_ptr(p)); ec != 0)
    return error(ec);
use(*p);
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Adapt a smart owner to a C API that writes a newly created pointer. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

The foreign allocation and C++ smart owner must agree on the release function.

C++23 std::out_ptr/out_ptr_t

DISCIPLINE CARD 13.2

std::inout_ptr for Read-and-Replace APIs

PRINCIPLE **Use inout_ptr only when the C function reads the old pointer and may replace it.**

Why it matters

The wrapper exposes the current pointer and then re-adopts the resulting pointer. The exact foreign semantics matter: whether the old object is freed, mutated, replaced, or preserved on failure.

Engineering rules

- Read the C ownership contract first.
- Encode the exact deleter.
- Define success and failure ownership.
- Prefer a safer custom wrapper when semantics are ambiguous.

RISK IF IGNORED

Typical failure: ownership conventions differ across the boundary, causing leaks, double release, or exceptions escaping into an ABI that cannot handle them.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Use inout_ptr only when the C function reads the old pointer and may replace it. Read the C ownership contract first. Encode the exact deleter.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
// Use only if the foreign API has matching in/out ownership:
// c_refresh(std::inout_ptr(p));
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Use inout_ptr only when the C function reads the old pointer and may replace it. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

The wrapper must state what happens to the old resource on both success and failure.

C++23 std::inout_ptr/inout_ptr_t

DISCIPLINE CARD 13.3

Wrap Foreign Handles Immediately

PRINCIPLE **Keep C ownership rules at the boundary, not in domain code.**

Why it matters

A C API may use malloc/free, custom destroy functions, file descriptors, or opaque handles. Convert each protocol into one RAII type and expose safe C++ operations beyond the adapter.

Engineering rules

- One wrapper per resource protocol.
- No raw foreign owner enters domain logic.
- Translate error codes at the boundary.
- Keep callback/ABI exception policy explicit.

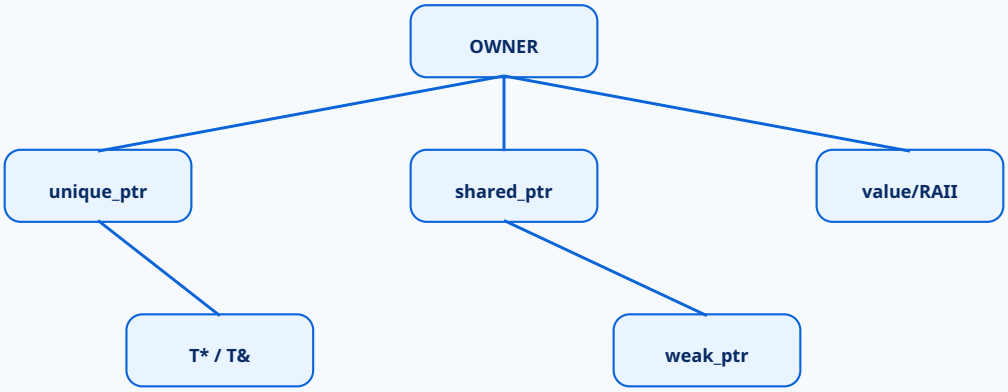
RISK IF IGNORED

Typical failure: an address travels without its valid extent, so an otherwise small indexing mistake becomes an out-of-bounds read or write.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Keep C ownership rules at the boundary, not in domain code. One wrapper per resource protocol. No raw foreign owner enters domain logic.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

Domain code should not need to know which C function releases the resource.

Core Guidelines R.1, R.12

DISCIPLINE CARD 13.4

Do Not Let Exceptions Cross a C ABI

PRINCIPLE **Catch C++ exceptions before returning through extern "C" entry points.**

Why it matters

Pure C interfaces do not define C++ exception propagation. Preserve RAII cleanup inside the C++ frame, then translate failure to the ABI's error protocol. The same applies to callbacks invoked by C libraries.

Engineering rules

- Catch and translate inside the boundary.
- Mark boundary functions noexcept when appropriate.
- Never throw through a C callback.
- Log enough context for diagnosis.

RISK IF IGNORED

Typical failure: an early exit leaves resources or invariants half-updated. The cure is to make rollback and cleanup lexical and automatic.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Catch C++ exceptions before returning through extern "C" entry points. Catch and translate inside the boundary. Mark boundary functions noexcept when appropriate.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
extern "C" int api_call(ctx* c) noexcept {
  try { return do_work(c) ? 0 : 1; }
  catch (...) { log_current_exception(); return -1; }
}
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Catch C++ exceptions before returning through extern "C" entry points. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

Every extern "C" entry point needs an explicit exception policy.

C/C++ interop practice

14

MEMORY DISCIPLINE CHAPTER

Exception Safety and Scope Guards

Resource safety is not complete until failure paths preserve invariants and cleanup.

DISCIPLINE CARD 14.1

Know the Guarantee You Provide

PRINCIPLE **Classify operations as no-throw, strong, basic, or no useful guarantee.**

Why it matters

RAII ensures cleanup, but state mutation still needs a failure contract. The strong guarantee leaves observable state unchanged on failure; the basic guarantee preserves invariants and prevents leaks.

Engineering rules

- Document guarantees for mutation-heavy public operations.
- Construct new state before committing.
- Use RAII for every temporary resource.
- Avoid multi-object partial mutation before throwing work.

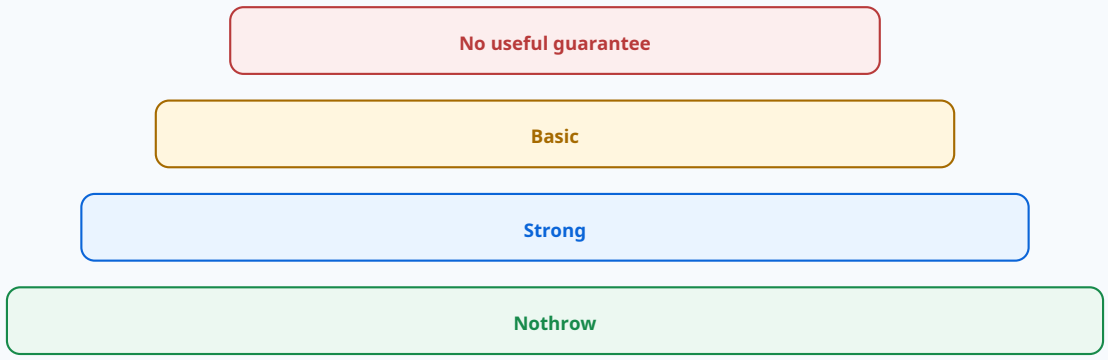
RISK IF IGNORED

Typical failure: the code leaves ownership, lifetime, or cleanup implicit. Under error paths or future refactoring, that implicit assumption becomes a memory defect.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Classify operations as no-throw, strong, basic, or no useful guarantee. Document guarantees for mutation-heavy public operations. Construct new state before committing.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

A public mutation that can throw must state its post-failure state.

Exception safety principles

DISCIPLINE CARD 14.2

C++23 scope_exit / scope_fail / scope_success

PRINCIPLE **Use scope guards for lexical cleanup or rollback that does not deserve a reusable type.**

Why it matters

scope_exit always runs, scope_fail runs on exceptional exit, and scope_success runs on successful exit. They complement RAII, especially for transaction rollback and short-lived paired operations.

Engineering rules

- Prefer a real RAII type for reusable resources.
- Keep guard callbacks small and non-throwing.
- Capture only objects that outlive the guard.
- Use scope_fail for rollback, not general exception handling.

RISK IF IGNORED

Typical failure: an early exit leaves resources or invariants half-updated. The cure is to make rollback and cleanup lexical and automatic.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Use scope guards for lexical cleanup or rollback that does not deserve a reusable type. Prefer a real RAII type for reusable resources. Keep guard callbacks small and non-throwing.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
std::scope_fail rollback([&] noexcept {
  db.restore(snapshot);
});
apply_changes();
commit();
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Use scope guards for lexical cleanup or rollback that does not deserve a reusable type. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

A guard callback must not access an object already destroyed before the guard.

C++23 <scope>

DISCIPLINE CARD 14.3

noexcept Move Enables Better Container Guarantees

PRINCIPLE **A truly non-throwing move lets containers relocate efficiently and safely.**

Why it matters

vector-like containers can prefer move during growth when move is noexcept; otherwise they may copy to preserve guarantees. Move-only RAII handles should usually move without throwing because they transfer simple ownership state.

Engineering rules

- Declare noexcept only when true.
- Use move_if_noexcept in generic relocation.
- Prefer noexcept-movable members.
- Test with throwing element types.

RISK IF IGNORED

Typical failure: the code leaves ownership, lifetime, or cleanup implicit. Under error paths or future refactoring, that implicit assumption becomes a memory defect.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. A truly non-throwing move lets containers relocate efficiently and safely. Declare noexcept only when true. Use move_if_noexcept in generic relocation.

REVIEW CHECKLIST

- Declare noexcept only when true.
- Use move_if_noexcept in generic relocation.
- Prefer noexcept-movable members.
- Test with throwing element types.

DESIGN CONSEQUENCE

A truly non-throwing move lets containers relocate efficiently and safely. Make the rule visible in the type, API, scope, or build policy so a later refactor cannot silently remove it.

REVIEW GATE

A false noexcept is a termination bug; missing noexcept can be a performance/guarantee cost.

Core Guidelines C.66

DISCIPLINE CARD 14.4

Commit Last: The Transaction Pattern

PRINCIPLE **Build and validate temporary state, then publish it in one small step.**

Why it matters

This naturally supports the strong guarantee. Temporary resources clean themselves on failure; only the final swap/move changes the live object.

Engineering rules

- Prepare before mutate.
- Validate before publish.
- Commit with swap/move.
- Keep observers from seeing half-built state.

RISK IF IGNORED

Typical failure: an early exit leaves resources or invariants half-updated. The cure is to make rollback and cleanup lexical and automatic.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Build and validate temporary state, then publish it in one small step. Prepare before mutate. Validate before publish.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
void Catalog::reload(Input in) {
  auto next = build_catalog(in);
  validate(next);
  data_.swap(next);
}
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Build and validate temporary state, then publish it in one small step. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

The final commit should be the smallest, safest step in the operation.

Strong exception guarantee

15

MEMORY DISCIPLINE CHAPTER

Concurrency: Lifetime Is a Synchronization Problem

A pointer can be valid in one thread and already dead in another unless lifetime and synchronization agree.

DISCIPLINE CARD 15.1

Join Lifetime or Transfer Ownership to Threads

PRINCIPLE **A task must not outlive the objects it borrows.**

Why it matters

Passing references/pointers to asynchronous work creates a lifetime edge. jthread helps because destruction joins. Detached work requires ownership transfer or storage with suitably long lifetime.

Engineering rules

- Prefer joining/structured concurrency.
- Do not detach while borrowing locals.
- Move owners into tasks when appropriate.
- Cancellation does not imply immediate thread termination.

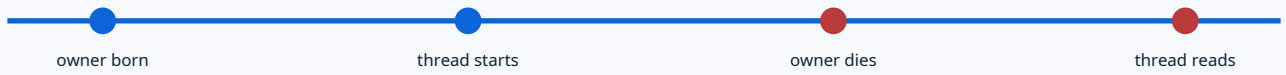
RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. A task must not outlive the objects it borrows. Prefer joining/structured concurrency. Do not detach while borrowing locals.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

For every async task, prove each borrowed object outlives task completion.

Core Guidelines CP.23-CP.25; jthread

DISCIPLINE CARD 15.2

atomic<shared_ptr> Coordinates Publication and Lifetime

PRINCIPLE **Atomic shared_ptr operations can safely publish shared immutable snapshots.**

Why it matters

The specialization coordinates access to the shared_ptr and lifetime of the managed object; it does not make mutable pointee state thread-safe.

Engineering rules

- Prefer immutable snapshots.
- Treat pointee synchronization separately.
- Measure reference-count traffic in hot paths.
- Use memory ordering consistent with publication.

RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Atomic shared_ptr operations can safely publish shared immutable snapshots. Prefer immutable snapshots. Treat pointee synchronization separately.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
std::atomic<std::shared_ptr<const Config>> current;
current.store(std::make_shared<const Config>(load()),
              std::memory_order_release);
auto cfg = current.load(std::memory_order_acquire);
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Atomic shared_ptr operations can safely publish shared immutable snapshots. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

Atomic ownership publication does not remove the need for data-race-free pointee access.

std::atomic<std::shared_ptr<T>>

DISCIPLINE CARD 15.3

Thread-Local Resources Reduce Contention

PRINCIPLE **Thread-confined pools/arenas can remove locks from hot allocation paths.**

Why it matters

The safety condition is lifetime and routing: memory allocated from a thread-local resource must not be deallocated through a resource that has died or on an incompatible thread path.

Engineering rules

- Define cross-thread ownership policy.
- Do not outlive the thread-local resource.
- Measure per-thread memory amplification.
- Use unsynchronized pools only when access is truly confined.

RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Thread-confined pools/arenas can remove locks from hot allocation paths. Define cross-thread ownership policy. Do not outlive the thread-local resource.

REVIEW CHECKLIST

- Define cross-thread ownership policy.
- Do not outlive the thread-local resource.
- Measure per-thread memory amplification.
- Use unsynchronized pools only when access is truly confined.

DESIGN CONSEQUENCE

Thread-confined pools/arenas can remove locks from hot allocation paths. Make the rule visible in the type, API, scope, or build policy so a later refactor cannot silently remove it.

REVIEW GATE

Thread-local allocation is safe only when allocation lifetime fits thread lifetime.

thread_local; unsynchronized_pool_resource

DISCIPLINE CARD 15.4

Reclamation Is Harder Than Atomic Pointer Updates

PRINCIPLE **Removing a node atomically is not the same as making it safe to delete.**

Why it matters

Readers may still hold a pointer after a writer unlinks it. Reclamation needs reference counting, hazard pointers, RCU, epochs, or locking.

Engineering rules

- Separate logical removal from physical reclamation.
- Choose one reclamation strategy.
- Test stalled readers.
- Do not invent ad-hoc lock-free reclamation in product code.

RISK IF IGNORED

Typical failure: the object is reclaimed while another thread can still observe it, or synchronization does not establish the required ordering.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Removing a node atomically is not the same as making it safe to delete. Separate logical removal from physical reclamation. Choose one reclamation strategy.

MODEL / VISUAL

HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

No lock-free node may be freed until the algorithm proves readers cannot access it.

C++26 hazard pointer/RCU motivation

16

MEMORY DISCIPLINE CHAPTER

The C++ Memory Model and Atomics

Cross-core lifetime reasoning depends on happens-before relationships, not wall-clock intuition.

DISCIPLINE CARD 16.1

Happens-Before Is the Correct Mental Model

PRINCIPLE **Cross-thread visibility comes from synchronization relations.**

Why it matters

A data race on non-atomic objects is undefined behavior. Mutexes and atomics establish the ordering needed to publish and mutate state safely. volatile is not a thread synchronization primitive.

Engineering rules

- Prefer mutexes for multi-value invariants.
- Use atomics for proven independent state/algorithms.
- Never use volatile for thread synchronization.
- Keep synchronization near protected data.

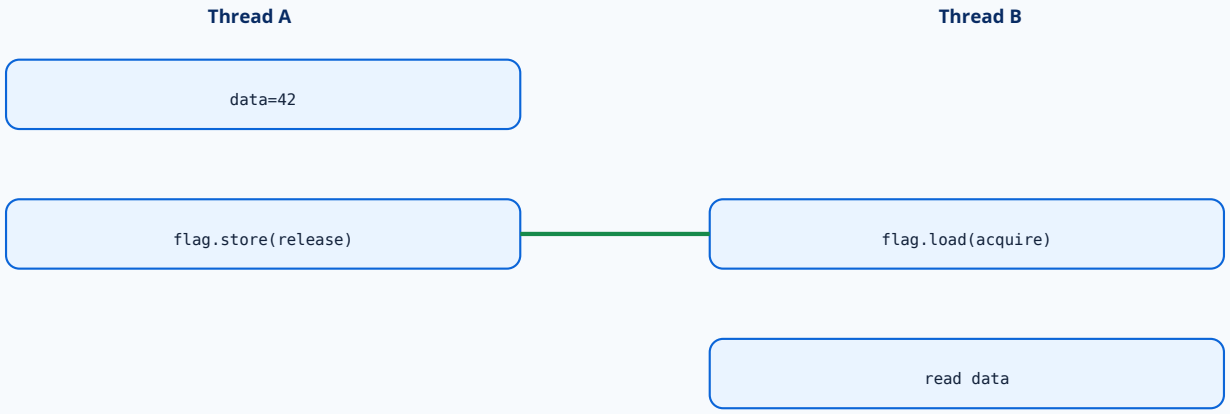
RISK IF IGNORED

Typical failure: the object is reclaimed while another thread can still observe it, or synchronization does not establish the required ordering.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Cross-thread visibility comes from synchronization relations. Prefer mutexes for multi-value invariants. Use atomics for proven independent state/algorithms.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

If a reviewer cannot draw the happens-before edge, the access is not justified.

C++ memory model; Core Guidelines CP

DISCIPLINE CARD 16.2

Use the Weakest Memory Order You Can Prove

PRINCIPLE **Start simple; weaken ordering only after proof and measurement.**

Why it matters

seq_cst is easiest to reason about. acquire/release is common for publication. relaxed provides atomicity but no additional ordering. Memory-order tuning is advanced work.

Engineering rules

- Default to correctness.
- Comment why weaker ordering is sufficient.
- Tie the comment to an invariant.
- Test on weakly ordered architectures where possible.

RISK IF IGNORED

Typical failure: the object is reclaimed while another thread can still observe it, or synchronization does not establish the required ordering.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Start simple; weaken ordering only after proof and measurement. Default to correctness. Comment why weaker ordering is sufficient.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
ready.store(true,std::memory_order_release);
if (ready.load(std::memory_order_acquire))
  consume(data);
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Start simple; weaken ordering only after proof and measurement. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

Every relaxed atomic in production needs a comment explaining why ordering is unnecessary.

C++ atomic memory ordering

DISCIPLINE CARD 16.3

atomic_ref Adds Atomic Access to Existing Objects

PRINCIPLE **Use atomic_ref only when alignment, lifetime, and access-mode requirements are satisfied.**

Why it matters

It is useful for mapped layouts or existing fields that cannot become std::atomic members. Conflicting non-atomic access is still a data race.

Engineering rules

- Check required_alignment.
- The referenced object must outlive atomic_ref.
- Do not mix unsynchronized non-atomic access.
- Use supported trivially copyable types.

RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Use atomic_ref only when alignment, lifetime, and access-mode requirements are satisfied. Check required_alignment. The referenced object must outlive atomic_ref.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
alignas(std::atomic_ref<int>::required_alignment) int counter=0;
std::atomic_ref<int> a{counter};
a.fetch_add(1,std::memory_order_relaxed);
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Use atomic_ref only when alignment, lifetime, and access-mode requirements are satisfied. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

atomic_ref changes access semantics, not ownership or lifetime.

std::atomic_ref

DISCIPLINE CARD 16.4

False Sharing Is a Layout Bug

PRINCIPLE **Independent writable variables can still contend when they share a cache line.**

Why it matters

Padding or separating hot write locations can improve scaling, but only after measurement because larger layouts increase footprint.

Engineering rules

- Measure before padding.
- Separate independently hot writes.
- Do not pack unrelated hot counters by default.
- Consider footprint trade-offs.

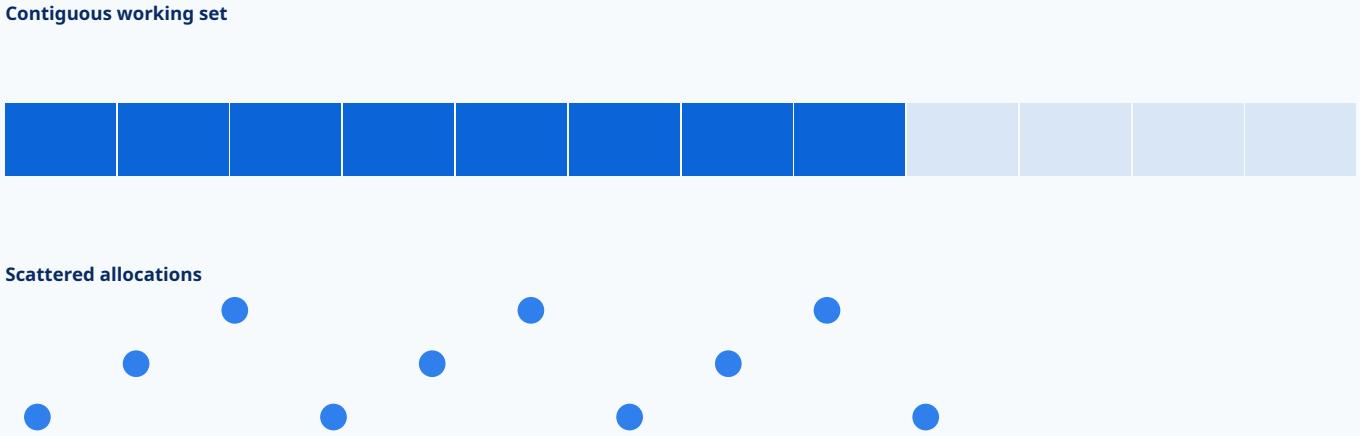
RISK IF IGNORED

Typical failure: the code leaves ownership, lifetime, or cleanup implicit. Under error paths or future refactoring, that implicit assumption becomes a memory defect.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Independent writable variables can still contend when they share a cache line. Measure before padding. Separate independently hot writes.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

Low lock contention does not rule out cache-coherence contention.

hardware_destructive_interference_size

PART 5

C++26 MEMORY ENGINEERING

Contracts, hardening, reclamation, and new ownership facilities

5

17

MEMORY DISCIPLINE CHAPTER

C++26 Contracts and Standard Library Hardening

C++26 adds stronger mechanisms for expressing and checking preconditions, postconditions, and selected library preconditions.

DISCIPLINE CARD 17.1

Contracts Express Preconditions and Postconditions

PRINCIPLE **Use C++26 pre/post conditions to make important interface assumptions explicit.**

Why it matters

Contracts provide executable documentation and can detect violated assumptions near the boundary. They do not replace ownership types, bounds-aware views, or RAII.

Engineering rules

- Use pre for caller obligations.
- Use post for result/state guarantees.
- Keep predicates free of side effects.
- Guard deployment with feature-test checks.

RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Use C++26 pre/post conditions to make important interface assumptions explicit. Use pre for caller obligations. Use post for result/state guarantees.

MODEL / VISUAL

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
#if __cpp_contracts >= 202502L
int at(std::span<const int> s,std::size_t i)
  pre(i < s.size())
  post(r: r == s[i]) { return s[i]; }
#endif
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++2c. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Use C++26 pre/post conditions to make important interface assumptions explicit. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

A contract should state a useful invariant, not repeat obvious syntax.

C++26 contracts; __cpp_contracts=202502L

DISCIPLINE CARD 17.2

contract_assert for Internal Invariants

PRINCIPLE **Use contract_assert for programmer invariants inside a function or lambda.**

Why it matters

Internal contract assertions can fail close to the broken assumption. They should not replace recoverable input validation, and predicates must not carry correctness-critical side effects because evaluation semantics are configurable.

Engineering rules

- No side effects in predicates.
- Do not use contracts as normal control flow.
- Validate untrusted input separately.
- Use contracts for invariants, not ordinary errors.

RISK IF IGNORED

Typical failure: the code leaves ownership, lifetime, or cleanup implicit. Under error paths or future refactoring, that implicit assumption becomes a memory defect.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Use contract_assert for programmer invariants inside a function or lambda. No side effects in predicates. Do not use contracts as normal control flow.

MODEL / VISUAL

HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

If correctness depends on the predicate executing, the design is wrong.

C++26 contract_assert

DISCIPLINE CARD 17.3

C++26 Standard Library Hardening

PRINCIPLE **Enable hardened implementations as defense in depth, not as permission to violate preconditions.**

Why it matters

C++26 defines hardened precondition behavior for selected library facilities. In a hardened implementation, violations can trigger contract handling; without hardening, the invalid call may remain undefined behavior.

Engineering rules

- Enable vendor hardening modes where appropriate.
- Keep preconditions valid by construction.
- Test hardened and optimized production-like builds.
- Do not remove bounds discipline because hardening exists.

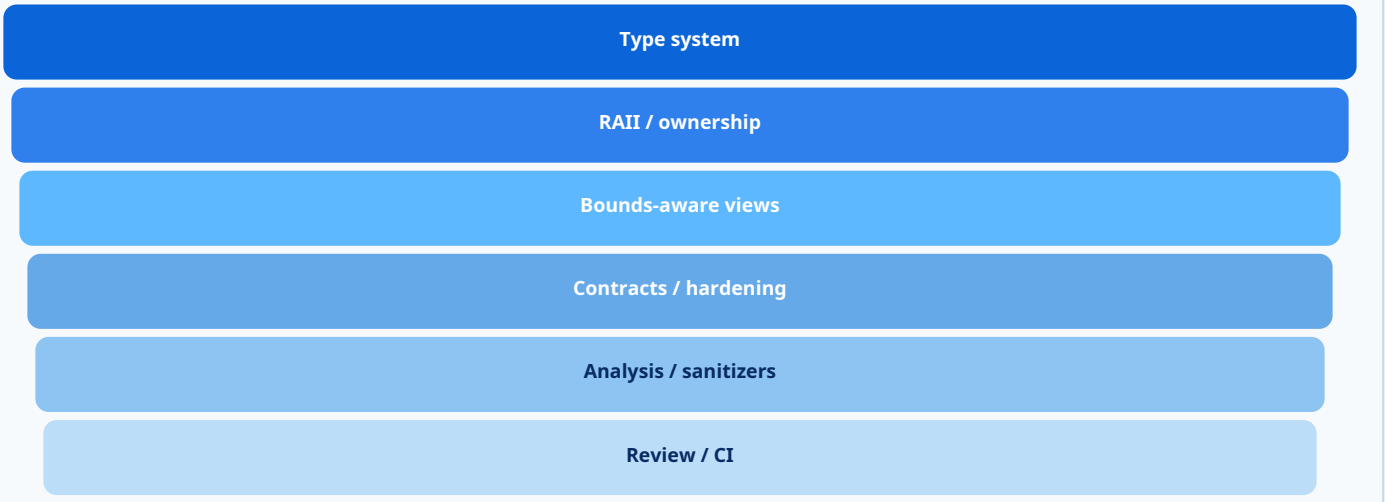
RISK IF IGNORED

Typical failure: the code leaves ownership, lifetime, or cleanup implicit. Under error paths or future refactoring, that implicit assumption becomes a memory defect.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Enable hardened implementations as defense in depth, not as permission to violate preconditions. Enable vendor hardening modes where appropriate. Keep preconditions valid by construction.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

Hardening catches violated contracts; good design avoids violating them.

C++26 standard library hardening

DISCIPLINE CARD 17.4

Feature Detection Is Part of Safe Deployment

PRINCIPLE **C++26 support is incremental across compilers and standard libraries.**

Why it matters

Feature-test macros and build-system capability checks are more reliable than compiler-version folklore. A safety-critical facility needs a defined fallback or a compile-time requirement.

Engineering rules

- Check <version> feature-test macros.
- Prefer capability checks to version checks.
- Document fallback semantics.
- Test every supported compiler + standard-library pair.

RISK IF IGNORED

Typical failure: ownership conventions differ across the boundary, causing leaks, double release, or exceptions escaping into an ABI that cannot handle them.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. C++26 support is incremental across compilers and standard libraries. Check <version> feature-test macros. Prefer capability checks to version checks.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
#include <version>
#if defined(__cpp_lib_inplace_vector)
  // C++26 path
#else
  // explicit fallback or hard error
#endif
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++2c. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

C++26 support is incremental across compilers and standard libraries. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

A safety-critical feature must not silently disappear on one supported toolchain.

C++ feature-test macro model

18

MEMORY DISCIPLINE CHAPTER

C++26 Hazard Pointers and Read-Copy Update

Standard reclamation tools reduce the need for bespoke lock-free lifetime machinery.

DISCIPLINE CARD 18.1

Hazard Pointers Protect Individual Nodes

PRINCIPLE **A reader announces the node it may dereference so reclamation waits.**

Why it matters

C++26 standardizes hazard_pointer, make_hazard_pointer, and hazard_pointer_obj_base. The reader protects, validates, uses, then clears; retired nodes are reclaimed later when no hazard protects them.

Engineering rules

- Use standard facilities instead of private protocols where support exists.
- Keep protected regions short.
- Retire removed nodes; do not delete immediately.
- Test stalled-reader scenarios.

RISK IF IGNORED

Typical failure: the object is reclaimed while another thread can still observe it, or synchronization does not establish the required ordering.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. A reader announces the node it may dereference so reclamation waits. Use standard facilities instead of private protocols where support exists. Keep protected regions short.

MODEL / VISUAL

HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

Atomic unlink and safe reclamation are separate steps.

C++26 <hazard_pointer>; P2530R3

DISCIPLINE CARD 18.2

Retire Instead of Delete

PRINCIPLE **A logically removed node may still be reachable by another reader.**

Why it matters

Hazard-pointer-aware objects can defer destruction until protection scans show no reader can still access the object. This separates container topology from physical memory reclamation.

Engineering rules

- Define the deleter strategy.
- Keep deleters non-throwing.
- Do not reuse retired storage prematurely.
- Measure retire-list growth.

RISK IF IGNORED

Typical failure: the object is reclaimed while another thread can still observe it, or synchronization does not establish the required ordering.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. A logically removed node may still be reachable by another reader. Define the deleter strategy. Keep deleters non-throwing.

MODEL / VISUAL

HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

A removed node is not reclaimable merely because the container no longer points to it.

C++26 hazard_pointer_obj_base::retire

DISCIPLINE CARD 18.3

RCU Favors Read-Mostly Workloads

PRINCIPLE **Readers use a stable version while writers publish a replacement and defer reclamation.**

Why it matters

C++26 provides rcu_domain, rcu_obj_base, synchronization/barrier operations, and retirement functions. RCU can make reads very cheap while shifting complexity to updates and grace-period reclamation.

Engineering rules

- Use for read-dominant structures.
- Keep reader critical sections bounded.
- Understand grace periods.
- Do not replace every mutex with RCU.

RISK IF IGNORED

Typical failure: storage, construction, alignment, or deallocation contracts are mixed, causing leaks, fragmentation, invalid lifetimes, or mismatched release.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Readers use a stable version while writers publish a replacement and defer reclamation. Use for read-dominant structures. Keep reader critical sections bounded.

MODEL / VISUAL

HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

Choose RCU only when read-path cost is a measured concern.

C++26 <rcu>; P2545R4

DISCIPLINE CARD 18.4

Choose Reclamation by Workload

PRINCIPLE **Locks, shared_ptr, hazard pointers, and RCU solve different lifetime topologies.**

Why it matters

Reference counting is simple but adds atomic traffic. Hazard pointers fit precise node protection. RCU fits read-heavy version replacement. A mutex may still be simpler and fast enough.

Engineering rules

- Start with locks unless requirements justify more.
- Use shared_ptr when shared lifetime is the semantic model.
- Use hazard pointers for pointer-centric lock-free structures.
- Use RCU for read-mostly replacement patterns.

RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Locks, shared_ptr, hazard pointers, and RCU solve different lifetime topologies. Start with locks unless requirements justify more. Use shared_ptr when shared lifetime is the semantic model.

MODEL / VISUAL

HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

Do not accept a lock-free design without a separate reclamation proof.

Concurrency reclamation design

19

MEMORY DISCIPLINE CHAPTER

C++26 Value-Like Dynamic Ownership Types

New C++26 facilities reduce hand-written owning-pointer boilerplate for heap-backed value semantics and stable-memory containers.

DISCIPLINE CARD 19.1

std::indirect: Heap Storage with Value Semantics

PRINCIPLE **Use indirect<T> when an object should behave like a value while its payload is dynamically allocated.**

Why it matters

Copies duplicate the owned value rather than share pointer identity. This can simplify pImpl-style designs, recursive structures, and large objects that want value semantics with indirection.

Engineering rules

- Use when value semantics are truly intended.
- Do not confuse it with shared ownership.
- Account for allocation cost on copies.
- Use allocator-aware variants when allocation policy matters.

RISK IF IGNORED

Typical failure: storage, construction, alignment, or deallocation contracts are mixed, causing leaks, fragmentation, invalid lifetimes, or mismatched release.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Use indirect<T> when an object should behave like a value while its payload is dynamically allocated. Use when value semantics are truly intended. Do not confuse it with shared ownership.

MODEL / VISUAL

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
#if __cpp_lib_indirect >= 202502L
std::indirect<LargeState> state{std::in_place,args...};
auto copy = state; // value copy
#endif
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++2c. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Use indirect<T> when an object should behave like a value while its payload is dynamically allocated. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

Choose indirect because you want value semantics plus indirection.

C++26 std::indirect

DISCIPLINE CARD 19.2

std::polymorphic: Value Semantics for Runtime Polymorphism

PRINCIPLE **Own a Base-or-derived object while copying the value rather than the pointer.**

Why it matters

std::polymorphic reduces custom clone boilerplate for designs that need dynamic type but value-like behavior. Copying may allocate and duplicate the dynamic object.

Engineering rules

- Use for value-like polymorphic abstractions.
- Do not replace borrowing references.
- Measure copying cost.
- Use a sound polymorphic base interface.

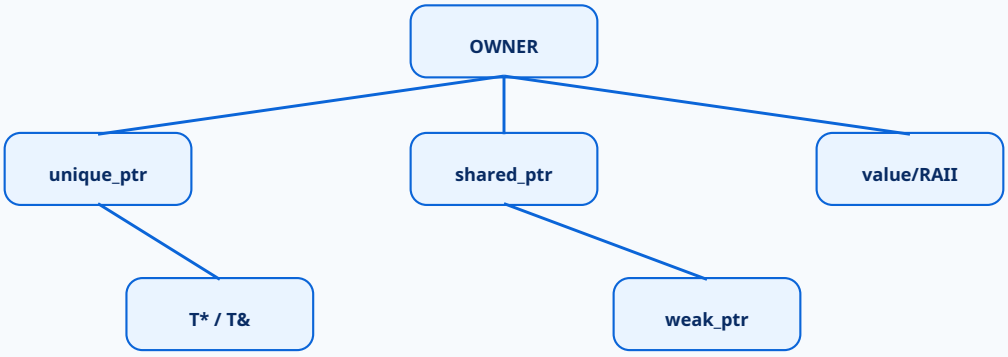
RISK IF IGNORED

Typical failure: storage, construction, alignment, or deallocation contracts are mixed, causing leaks, fragmentation, invalid lifetimes, or mismatched release.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Own a Base-or-derived object while copying the value rather than the pointer. Use for value-like polymorphic abstractions. Do not replace borrowing references.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

If two instances should share identity, value-like polymorphism may be the wrong model.

C++26 std::polymorphic

DISCIPLINE CARD 19.3

std::hive Reuses Erased-Element Storage

PRINCIPLE **Use hive for erase-heavy workloads that need stable element locations more than contiguity.**

Why it matters

C++26 hive is a bucket-based sequence container that can reuse erased-element storage. It trades contiguous locality for stability/reuse characteristics.

Engineering rules

- Prefer vector when locality dominates.
- Choose hive for measured erase-heavy stable-location needs.
- Know iterator/reference guarantees.
- Do not choose a container by novelty.

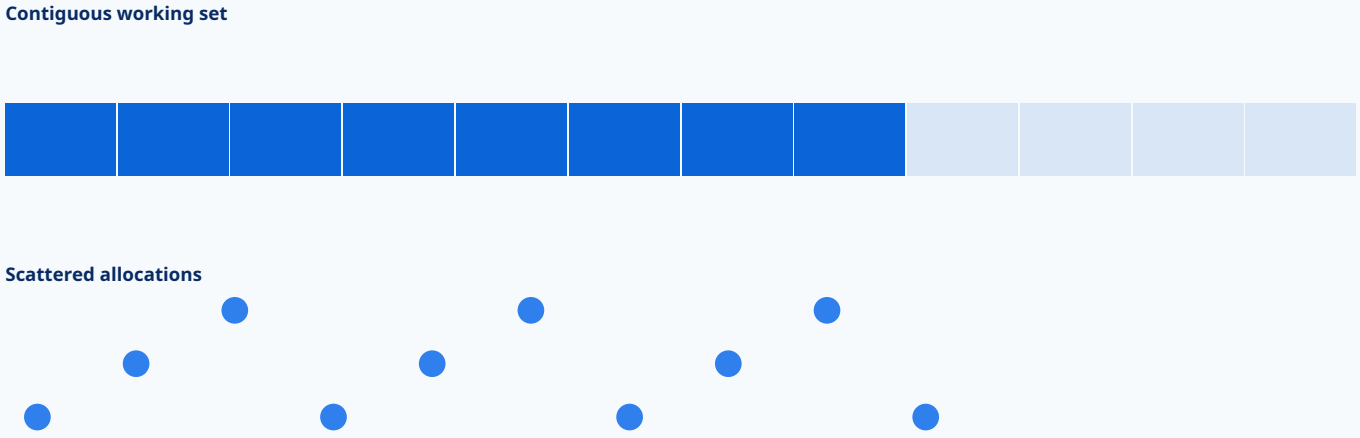
RISK IF IGNORED

Typical failure: storage, construction, alignment, or deallocation contracts are mixed, causing leaks, fragmentation, invalid lifetimes, or mismatched release.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Use hive for erase-heavy workloads that need stable element locations more than contiguity. Prefer vector when locality dominates. Choose hive for measured erase-heavy stable-location needs.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

Container choice follows access pattern, stability requirement, and measurements.

C++26 <hive>; P0447R28

DISCIPLINE CARD 19.4

New Facilities Reduce Boilerplate, Not Responsibility

PRINCIPLE **Adopt C++26 facilities when they simplify semantics or remove custom unsafe code.**

Why it matters

indirect/polymorphic clarify heap-backed value semantics, owner hashing clarifies shared lifetime identity, inplace_vector constrains allocation, and hazard/RCU standardize reclamation. The foundational ownership/lifetime rules still apply.

Engineering rules

- Adopt when semantics improve.
- Guard partial implementation support.
- Keep performance costs explicit.
- Retire custom wrappers only after behavior is matched.

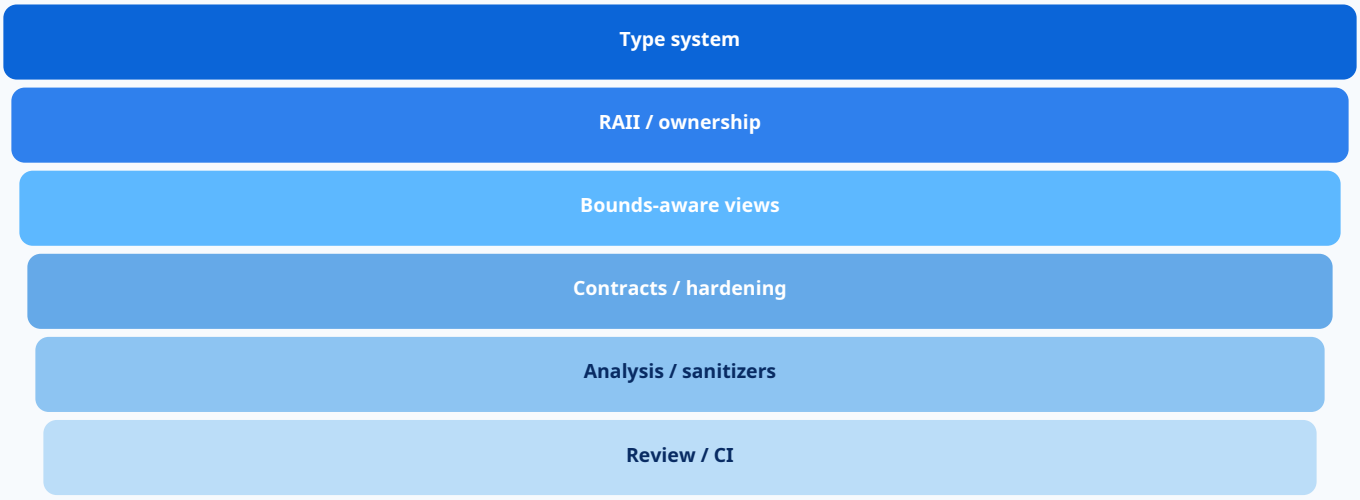
RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Adopt C++26 facilities when they simplify semantics or remove custom unsafe code. Adopt when semantics improve. Guard partial implementation support.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

A newer abstraction is valuable when it removes unsafe custom machinery or clarifies intent.

C++26 memory-management additions

20

MEMORY DISCIPLINE CHAPTER

Embedded, Real-Time, and No-Allocation Paths

Deterministic memory often means constraining allocation rather than making general-purpose allocation faster.

DISCIPLINE CARD 20.1

Design Hot Paths to Allocate Zero Times

PRINCIPLE **For hard latency paths, the strongest allocation optimization is no allocation.**

Why it matters

Preallocate during initialization, reuse buffers, use fixed-capacity containers, pools, and phase arenas. This removes allocator latency variation and resource-exhaustion surprises from the critical phase.

Engineering rules

- Count allocations in tests.
- Separate initialization from the real-time phase.
- Pre-size to validated maxima.
- Fail startup if required capacity cannot be secured.

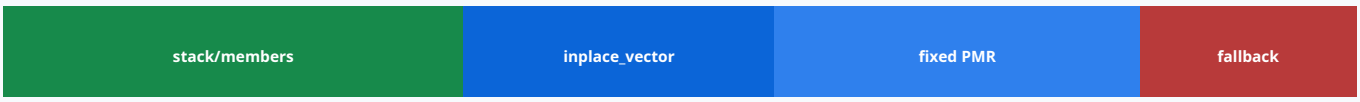
RISK IF IGNORED

Typical failure: an address travels without its valid extent, so an otherwise small indexing mistake becomes an out-of-bounds read or write.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. For hard latency paths, the strongest allocation optimization is no allocation. Count allocations in tests. Separate initialization from the real-time phase.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

A declared no-allocation phase must have an automated allocation-count test.

Real-time memory design

DISCIPLINE CARD 20.2

inplace_vector Replaces Hand-Rolled Bounded Buffers

PRINCIPLE **Use a standard fixed-capacity container instead of manual array + size bookkeeping.**

Why it matters

C++26 inplace_vector provides vector-like contiguous semantics without dynamic allocation. The capacity becomes an explicit resource budget.

Engineering rules

- Pick capacity from system bounds.
- Handle overflow deterministically.
- Prefer non-throwing try_ operations when appropriate.
- Review object/stack footprint.

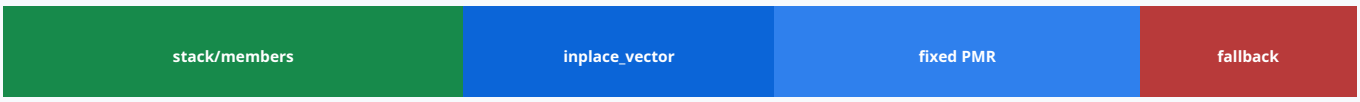
RISK IF IGNORED

Typical failure: an address travels without its valid extent, so an otherwise small indexing mistake becomes an out-of-bounds read or write.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Use a standard fixed-capacity container instead of manual array + size bookkeeping. Pick capacity from system bounds. Handle overflow deterministically.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

The capacity constant belongs in the system resource budget.

C++26 std::inplace_vector

DISCIPLINE CARD 20.3

Fixed PMR Buffer + null_memory_resource

PRINCIPLE **Make heap fallback structurally impossible when it would violate the requirement.**

Why it matters

A monotonic_buffer_resource can draw from a fixed buffer and use null_memory_resource as upstream. Once exhausted, allocation fails rather than silently using the general heap.

Engineering rules

- Size from measurement plus margin.
- Use null upstream when fallback is forbidden.
- Handle exhaustion at the defined boundary.
- Reset only when all arena-backed objects are dead.

RISK IF IGNORED

Typical failure: an address travels without its valid extent, so an otherwise small indexing mistake becomes an out-of-bounds read or write.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Make heap fallback structurally impossible when it would violate the requirement. Size from measurement plus margin. Use null upstream when fallback is forbidden.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
std::byte storage[32*1024];
std::pmr::monotonic_buffer_resource arena{
  storage,sizeof storage,std::pmr::null_memory_resource()
};
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Make heap fallback structurally impossible when it would violate the requirement. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

If fallback allocation violates latency/certification requirements, make fallback impossible.

pmr monotonic_buffer_resource/null_memory_resource

DISCIPLINE CARD 20.4

Deterministic Destruction Matters Too

PRINCIPLE **A no-allocation path can still stall while destroying a large graph or last shared owner.**

Why it matters

Destruction can release containers, files, mappings, or deep ownership trees. Move expensive teardown to a non-critical phase or bound the graph when semantics allow.

Engineering rules

- Profile destructor time.
- Avoid last-owner shared_ptr destruction in hard real-time code.
- Bound object graphs.
- Separate logical completion from deferred cleanup when safe.

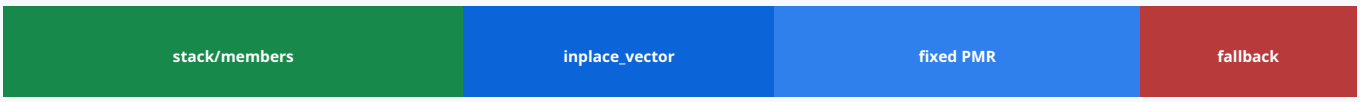
RISK IF IGNORED

Typical failure: an address travels without its valid extent, so an otherwise small indexing mistake becomes an out-of-bounds read or write.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. A no-allocation path can still stall while destroying a large graph or last shared owner. Profile destructor time. Avoid last-owner shared_ptr destruction in hard real-time code.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

No-allocation does not imply bounded destruction latency.

Real-time performance engineering

PART 6

PERFORMANCE & ENFORCEMENT

Locality, tools, Core Guidelines, and institutional migration

6

21

MEMORY DISCIPLINE CHAPTER

Performance: Locality, Growth, Fragmentation, and Measurement

Fast memory management is mostly about access patterns, locality, allocation frequency, and bounded lifetime—not clever allocation syntax.

DISCIPLINE CARD 21.1

Contiguity Is a Performance Feature

PRINCIPLE **A vector of values often beats a graph of individually allocated nodes.**

Why it matters

CPUs fetch cache lines, not abstract containers. Contiguous storage reduces pointer chasing, allocation metadata, and often improves prefetching/vectorization.

Engineering rules

- Prefer contiguous containers for iteration-heavy data.
- Store values instead of pointers when semantics permit.
- Batch work over adjacent data.
- Measure cache misses as well as wall time.

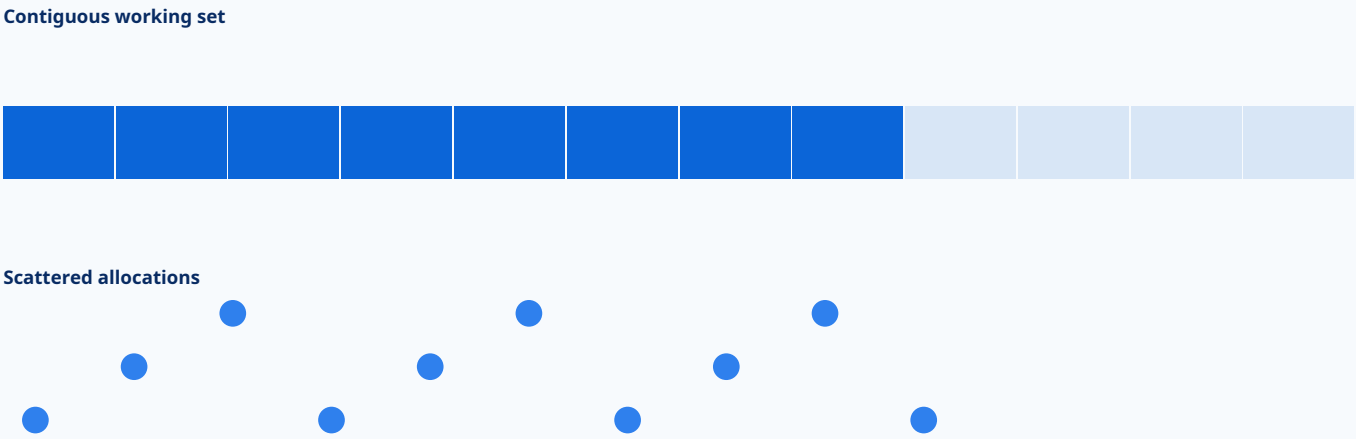
RISK IF IGNORED

Typical failure: storage, construction, alignment, or deallocation contracts are mixed, causing leaks, fragmentation, invalid lifetimes, or mismatched release.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. A vector of values often beats a graph of individually allocated nodes. Prefer contiguous containers for iteration-heavy data. Store values instead of pointers when semantics permit.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

A memory optimization proposal should explain the expected access pattern.

Core Guidelines P.9; locality principles

DISCIPLINE CARD 21.2

AoS vs SoA: Layout Follows the Hot Loop

PRINCIPLE **Choose physical layout based on which fields are touched together.**

Why it matters

Array-of-structures is convenient when whole objects are consumed; structure-of-arrays can improve locality and SIMD when only selected fields are hot. Hybrid layouts are common.

Engineering rules

- Profile hot fields.
- Keep ownership semantics separate from physical layout.
- Avoid premature SoA complexity.
- Re-test after compiler/vectorization changes.

RISK IF IGNORED

Typical failure: an address travels without its valid extent, so an otherwise small indexing mistake becomes an out-of-bounds read or write.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Choose physical layout based on which fields are touched together. Profile hot fields. Keep ownership semantics separate from physical layout.

COPY-READY C++ PATTERN SELECT • COPY • PASTE

```
struct ParticlesSoA {
  std::vector<float> x,y,z;
  std::vector<float> vx,vy,vz;
};
```

TRY IT

Paste the block into a small scratch file or your project and build with -std=c++23. For library/feature-dependent fragments, keep the feature-test guard and use the newest supported standard library.

WHAT TO NOTICE

Choose physical layout based on which fields are touched together. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

Choose layout by bytes touched per useful operation.

Data-oriented design

DISCIPLINE CARD 21.3

Growth Strategy Controls Allocation Frequency

PRINCIPLE **Repeated growth moves data, allocates, and invalidates references.**

Why it matters

reserve can reduce growth operations. In low-level custom storage, growth factor and allocate_at_least trade memory slack against relocation frequency.

Engineering rules

- Reserve from measured upper estimates.
- Avoid shrink-to-fit churn.
- Track peak capacity vs peak size.
- Reuse buffers in repetitive workloads.

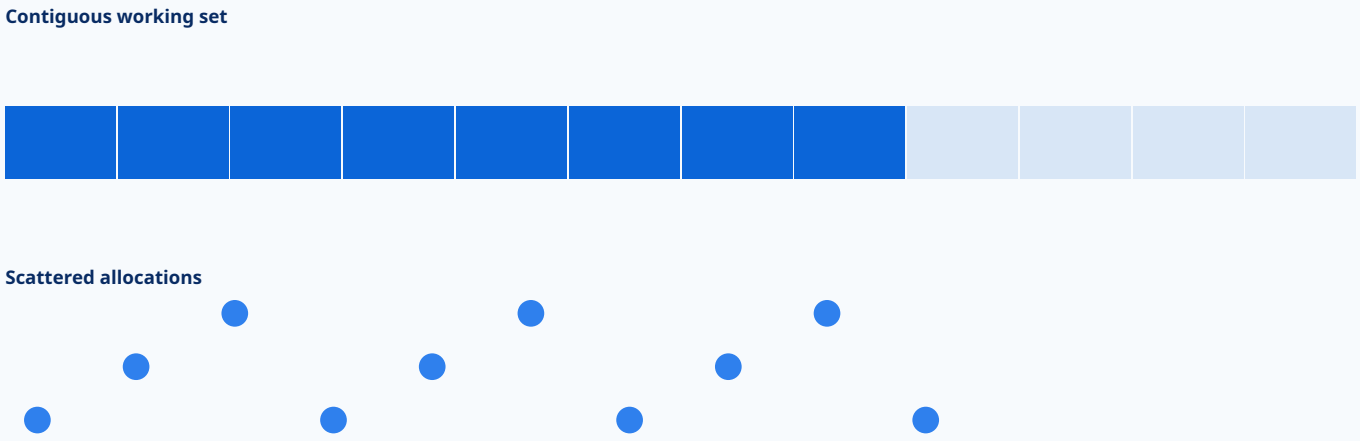
RISK IF IGNORED

Typical failure: storage, construction, alignment, or deallocation contracts are mixed, causing leaks, fragmentation, invalid lifetimes, or mismatched release.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Repeated growth moves data, allocates, and invalidates references. Reserve from measured upper estimates. Avoid shrink-to-fit churn.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

Measure allocations and bytes moved, not only final footprint.

vector capacity; C++23 allocate_at_least

DISCIPLINE CARD 21.4

Fragmentation Must Be Measured in Context

PRINCIPLE **“Heap fragmentation” is not one metric.**

Why it matters

Distinguish internal fragmentation, external fragmentation, allocator metadata, retained pages, and application slack capacity. Pools/arenas can reduce some forms while increasing retained memory.

Engineering rules

- Name the metric before optimizing.
- Use allocator/profiler statistics.
- Separate live bytes from retained bytes.
- Test long-running steady state.

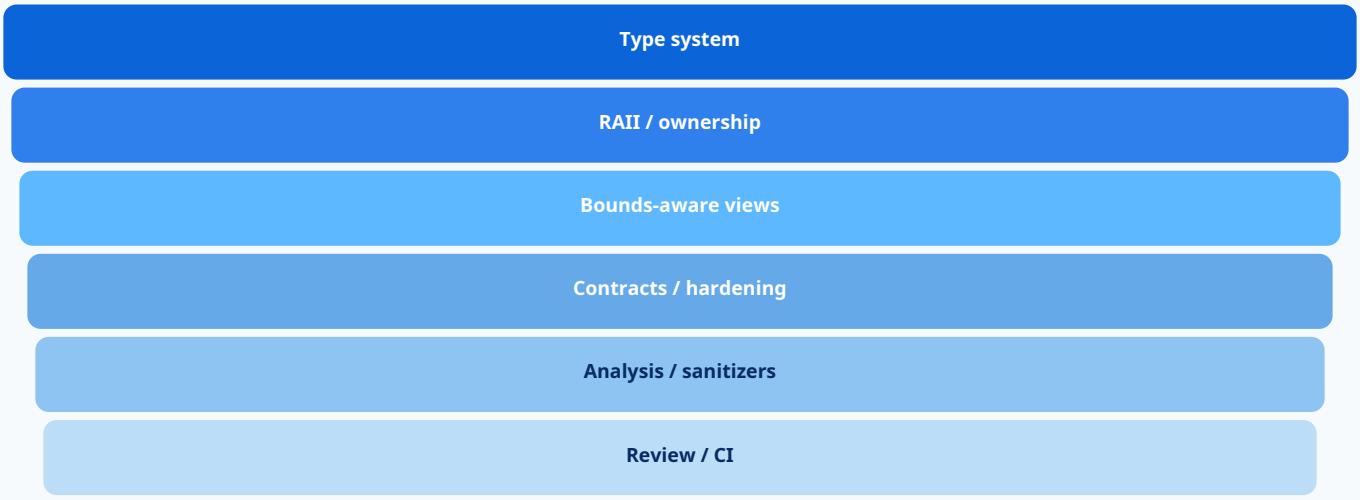
RISK IF IGNORED

Typical failure: storage, construction, alignment, or deallocation contracts are mixed, causing leaks, fragmentation, invalid lifetimes, or mismatched release.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. “Heap fragmentation” is not one metric. Name the metric before optimizing. Use allocator/profiler statistics.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

No fragmentation claim is accepted without a named metric and workload.

Memory profiling practice

22

MEMORY DISCIPLINE CHAPTER

Verification: Warnings, Sanitizers, Fuzzing, and Static Analysis

Safety discipline becomes credible when the toolchain continuously checks it.

DISCIPLINE CARD 22.1

Compiler Warnings Are the First Gate

PRINCIPLE **Keep a strong warning baseline clean on every supported compiler.**

Why it matters

Start with broad warning groups, then add project-appropriate conversion/shadow diagnostics. Suppress only locally and with a reason. Warning policy is versioned build configuration, not a developer preference.

Engineering rules

- No ignored warnings in CI.
- Suppress at the smallest scope.
- Compile headers in representative modes.
- Test multiple compilers when portability matters.

RISK IF IGNORED

Typical failure: unsafe behavior survives because the project relies on review alone. Automated diagnostics must be part of the normal build and test pipeline.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Keep a strong warning baseline clean on every supported compiler. No ignored warnings in CI. Suppress at the smallest scope.

COMPILER FLAGS SELECT • COPY • PASTE

```
# GCC/Clang baseline
-Wall -Wextra -Wpedantic
```

TRY IT

Copy these options into a dedicated debug/sanitized build profile. Keep the optimized release build separate, and make CI fail on new diagnostics.

WHAT TO NOTICE

Keep a strong warning baseline clean on every supported compiler. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

Every new warning suppression needs a written reason.

Compiler diagnostics best practice

DISCIPLINE CARD 22.2

ASan + UBSan: High-Value Dynamic Pair

PRINCIPLE **Run memory and UB detection continuously, not only after a crash.**

Why it matters

ASan catches many heap/stack/global out-of-bounds, use-after-free, invalid/double free issues. UBSan catches many invalid operations such as null/misaligned dereference, certain bounds, shifts, and signed overflow.

Engineering rules

- Build with debug symbols.
- Run unit/integration/fuzz tests under sanitizers.
- Treat findings as release blockers.
- Keep incompatible sanitizer modes in separate jobs.

RISK IF IGNORED

Typical failure: an address travels without its valid extent, so an otherwise small indexing mistake becomes an out-of-bounds read or write.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Run memory and UB detection continuously, not only after a crash. Build with debug symbols. Run unit/integration/fuzz tests under sanitizers.

MODEL / VISUAL

COMPILER FLAGS SELECT • COPY • PASTE

```
-fsanitize=address,undefined
-fno-omit-frame-pointer
-g -O1
```

TRY IT

Copy these options into a dedicated debug/sanitized build profile. Keep the optimized release build separate, and make CI fail on new diagnostics.

WHAT TO NOTICE

Run memory and UB detection continuously, not only after a crash. The important part is the ownership/lifetime meaning expressed by the API shape, not merely the syntax.

REVIEW GATE

Every release branch needs a recent sanitizer-clean CI run.

Clang ASan/UBSan; GCC instrumentation

DISCIPLINE CARD 22.3

MSan and TSan Need Dedicated Jobs

PRINCIPLE **Uninitialized reads and data races deserve their own instrumented test configurations.**

Why it matters

MemorySanitizer typically needs instrumented dependencies; ThreadSanitizer has high overhead and cannot be combined with ASan in the same binary on common toolchains.

Engineering rules

- Run TSan on concurrency-heavy tests.
- Run MSan where dependency support permits.
- Do not dismiss races as benign without proof.
- Review suppressions and keep them temporary.

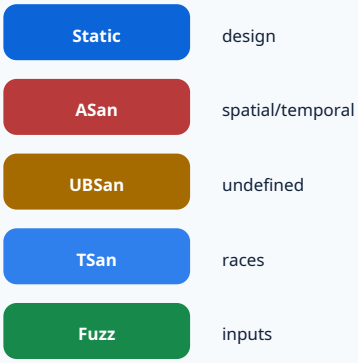
RISK IF IGNORED

Typical failure: the object is reclaimed while another thread can still observe it, or synchronization does not establish the required ordering.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Uninitialized reads and data races deserve their own instrumented test configurations. Run TSan on concurrency-heavy tests. Run MSan where dependency support permits.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

A sanitizer suppression is technical debt and needs an owner and reason.

Clang MSan/TSan; GCC sanitizer compatibility

DISCIPLINE CARD 22.4

Fuzz Ownership and Bounds Boundaries

PRINCIPLE **Fuzz where bytes become objects, indexes, lengths, or ownership transitions.**

Why it matters

Parsers, decoders, allocators, container wrappers, and C adapters should receive malformed/adversarial sequences. Combining fuzzing with ASan/UBSan turns hidden invalid accesses into immediate failures.

Engineering rules

- Fuzz length/offset arithmetic.
- Fuzz capacity exhaustion and allocation failure.
- Fuzz repeated create/destroy/reuse.
- Add production bugs as regression seeds.

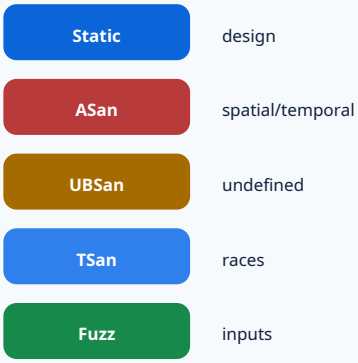
RISK IF IGNORED

Typical failure: an address travels without its valid extent, so an otherwise small indexing mistake becomes an out-of-bounds read or write.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Fuzz where bytes become objects, indexes, lengths, or ownership transitions. Fuzz length/offset arithmetic. Fuzz capacity exhaustion and allocation failure.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

Every fixed memory bug should become a regression test or fuzz seed.

Coverage-guided fuzzing + sanitizers

23

MEMORY DISCIPLINE CHAPTER

C++ Core Guidelines Profiles and Enforcement

Use the living C++ Core Guidelines as an enforceable engineering baseline while distinguishing them from the ISO standard itself.

DISCIPLINE CARD 23.1

The Guidelines Are Living Guidance, Not the ISO Standard

PRINCIPLE **Reference them precisely and use rule identifiers in engineering policy.**

Why it matters

The C++ Core Guidelines are edited by Bjarne Stroustrup and Herb Sutter. The browsable edition consulted for this book was updated June 14, 2026. They emphasize interfaces, resource management, memory management, concurrency, and safety.

Engineering rules

- Use rule IDs in review.
- Do not call the Guidelines the normative ISO standard.
- Adopt an enforceable project subset.
- Track updates as engineering inputs.

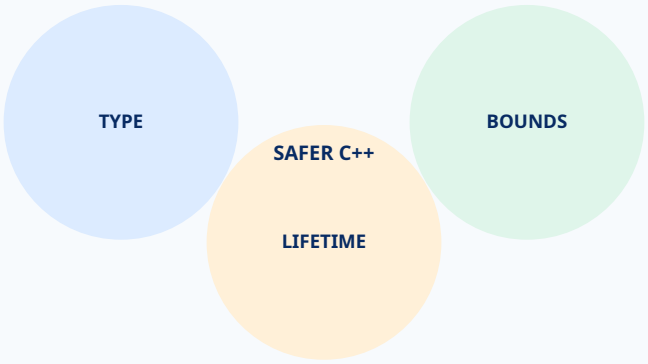
RISK IF IGNORED

Typical failure: the object is reclaimed while another thread can still observe it, or synchronization does not establish the required ordering.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Reference them precisely and use rule identifiers in engineering policy. Use rule IDs in review. Do not call the Guidelines the normative ISO standard.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

Policy should name exact rules, not merely say “follow the Core Guidelines.”

C++ Core Guidelines, June 14 2026

DISCIPLINE CARD 23.2

Type + Bounds + Lifetime Profiles Form a Safety Core

PRINCIPLE **Combine the profiles instead of treating memory safety as one smart-pointer rule.**

Why it matters

The lifetime profile targets leaks, double delete, null/dangling dereference; the bounds profile targets out-of-range access; the type profile targets unsafe reinterpretation. Together with RAII they define a strong restricted discipline.

Engineering rules

- Enable profiles supported by your analyzers.
- Use span/views to preserve bounds.
- Minimize unsafe casts.
- Classify owners and non-owners.

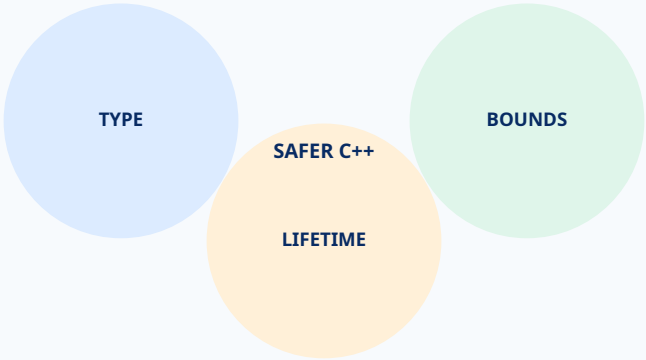
RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Combine the profiles instead of treating memory safety as one smart-pointer rule. Enable profiles supported by your analyzers. Use span/views to preserve bounds.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

A hardened-C++ claim should state which profiles are mechanically checked.

Core Guidelines safety profiles

DISCIPLINE CARD 23.3

Resource Rules R.1, R.3, R.5, R.10-R.24

PRINCIPLE **Turn the resource section into a practical ownership baseline.**

Why it matters

The key direction is consistent: RAII, raw pointers as non-owning, scoped objects, avoidance of malloc/free and explicit new/delete, immediate binding of allocations to managers, span for arrays, unique ownership by default, make_ functions, and weak_ptr for cycles.

Engineering rules

- Automate rule checks where possible.
- Teach the rule IDs.
- Allow exceptions only in resource infrastructure.
- Apply the safer interface shape to all new code.

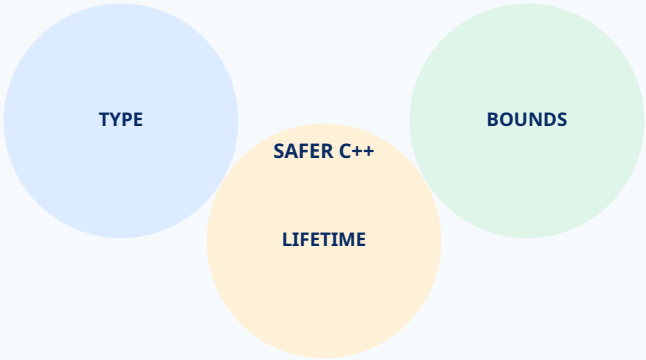
RISK IF IGNORED

Typical failure: an address travels without its valid extent, so an otherwise small indexing mistake becomes an out-of-bounds read or write.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Turn the resource section into a practical ownership baseline. Automate rule checks where possible. Teach the rule IDs.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

New product code needs an explicit waiver for naked owning raw pointers or delete.

Core Guidelines R section

DISCIPLINE CARD 23.4

Enforcement, Suppressions, and the GSL

PRINCIPLE **A rule becomes durable when violations and exceptions are visible.**

Why it matters

The Guidelines discuss tool enforcement, suppression, and the Guidelines Support Library. not_null and span-style types help encode interface contracts, especially in mixed-standard or legacy environments.

Engineering rules

- Prefer local suppression over global disable.
- Attach justification to every suppression.
- Review suppressions periodically.
- Keep unsafe wrappers small enough for focused audit.

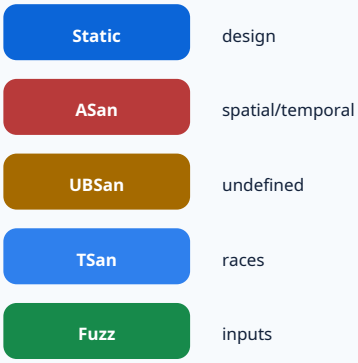
RISK IF IGNORED

Typical failure: an address travels without its valid extent, so an otherwise small indexing mistake becomes an out-of-bounds read or write.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. A rule becomes durable when violations and exceptions are visible. Prefer local suppression over global disable. Attach justification to every suppression.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

A suppression without a written justification fails review.

Core Guidelines enforcement/GSL

24

MEMORY DISCIPLINE CHAPTER

Modernizing Legacy C++ and Institutionalizing Safety

Turn individual knowledge into a staged migration, CI gates, and a short organization-wide memory-safety standard.

DISCIPLINE CARD 24.1

Inventory Ownership Before Refactoring Syntax

PRINCIPLE **Discover who owns every resource before changing pointer types.**

Why it matters

Search for new/delete, malloc/free, create/destroy pairs, owning raw members, pointer-returning factories, array parameters, pointer arithmetic, shared ownership, and hand-written cleanup. Classify each site before replacing it.

Engineering rules

- Build an ownership inventory.
- Mark high-risk lifetime crossings.
- Add sanitizer coverage before large refactors.
- Migrate one ownership boundary at a time.

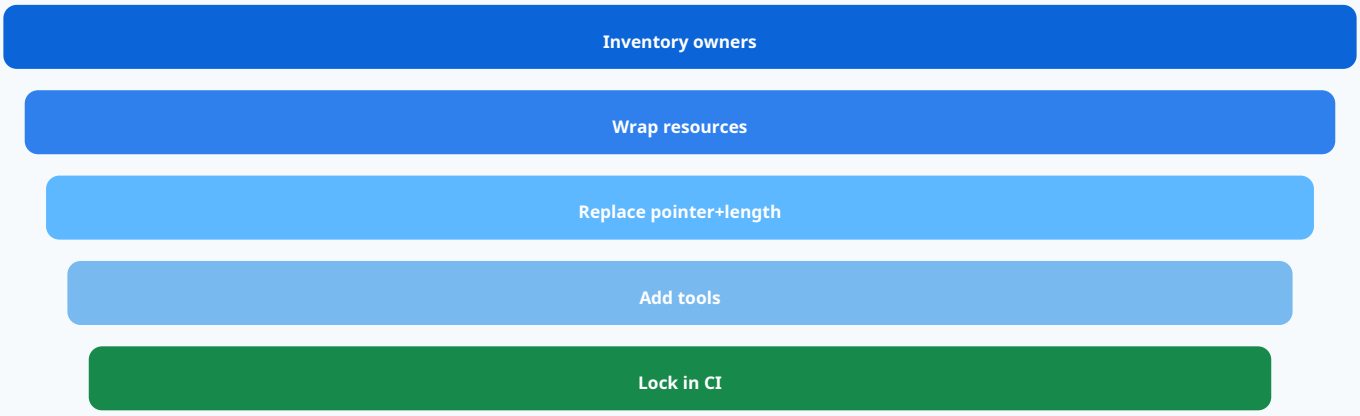
RISK IF IGNORED

Typical failure: an address travels without its valid extent, so an otherwise small indexing mistake becomes an out-of-bounds read or write.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Discover who owns every resource before changing pointer types. Build an ownership inventory. Mark high-risk lifetime crossings.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

Do not mechanically replace raw pointers with shared_ptr; determine semantics first.

Core Guidelines gradual adoption

DISCIPLINE CARD 24.2

Migrate Owners First, Then Interfaces

PRINCIPLE **Converting owners to RAII gives the highest immediate reduction in leak/double-free risk.**

Why it matters

Once ownership is stable, convert pointer+length to span, clarify nullability, replace caller-delete factories, and shorten long-lived borrows. Staging limits behavior changes and simplifies diagnosis.

Engineering rules

- Owners -> unique_ptr/resource handles.
- Dynamic arrays -> standard containers.
- Pointer+length -> span.
- Required/optional borrow -> T&/T*.

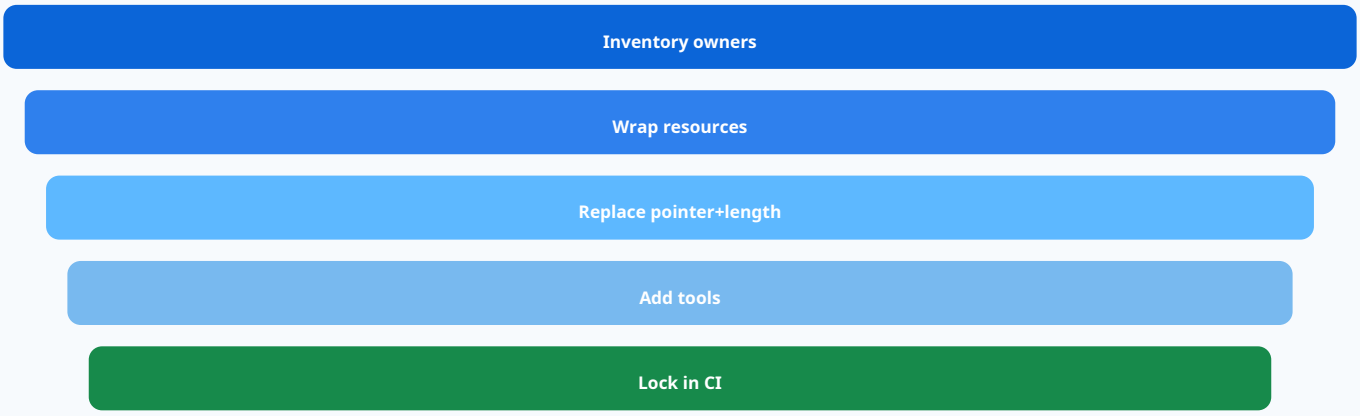
RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Converting owners to RAII gives the highest immediate reduction in leak/double-free risk. Owners -> unique_ptr/resource handles. Dynamic arrays -> standard containers.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

Each stage should preserve behavior before the next abstraction change.

Modern C++ migration

DISCIPLINE CARD 24.3

Lock the Improvement in CI

PRINCIPLE **A migration is temporary unless the build prevents old patterns from returning.**

Why it matters

Add lint/static analysis, sanitizer jobs, allocation-count tests, fuzz targets, warning gates, and review checklists. New code should meet the target profile while legacy islands shrink.

Engineering rules

- No new naked delete outside approved infrastructure.
- No new unchecked array interfaces.
- Sanitizer-clean CI.
- Track unsafe-code and waiver counts.

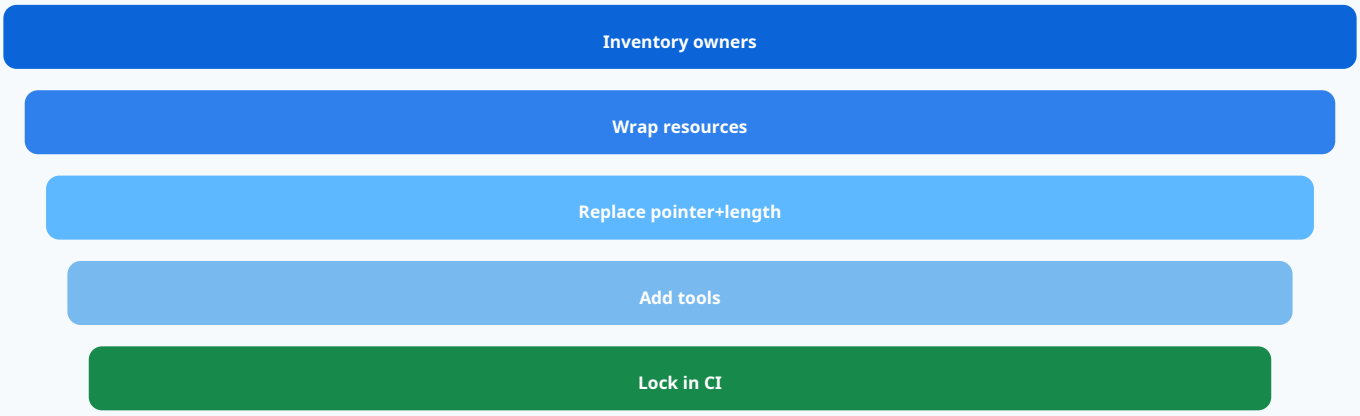
RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. A migration is temporary unless the build prevents old patterns from returning. No new naked delete outside approved infrastructure. No new unchecked array interfaces.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

Safety debt must trend downward and be measurable.

Engineering process

DISCIPLINE CARD 24.4

The Organization Memory-Safety Standard

PRINCIPLE **Write a one-page policy developers can apply without interpretation.**

Why it matters

A strong standard requires Rule of Zero, RAII, unique ownership by default, non-owning raw pointers, bounds-aware sequence interfaces, isolated low-level code, contract/precondition discipline, sanitizer/static-analysis CI, reviewed unsafe boundaries, and feature-test guards for C++26.

Engineering rules

- Keep policy short enough to remember.
- Back every rule with automation where possible.
- Define a waiver path.
- Revisit after language/toolchain upgrades.

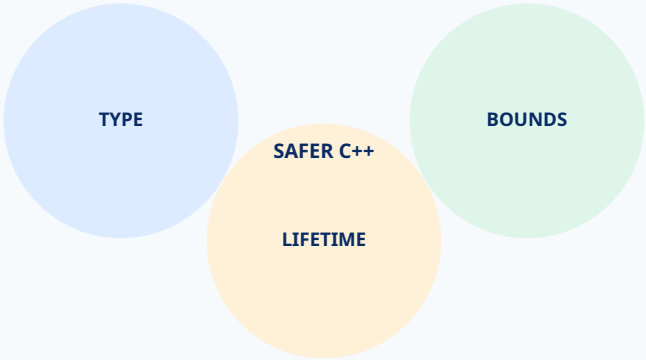
RISK IF IGNORED

Typical failure: a borrow outlives its owner or survives an invalidating operation, producing use-after-lifetime behavior that can remain invisible until optimization or load changes.

PRACTICAL APPLICATION

Apply this rule at API boundaries and during code review. Write a one-page policy developers can apply without interpretation. Keep policy short enough to remember. Back every rule with automation where possible.

MODEL / VISUAL



HOW TO READ THIS MODEL

Follow the arrows as lifetime/ownership transitions. The diagram is a review aid: identify the owner, the borrower, the invalidation event, and the point where the invariant is checked.

REVIEW GATE

The safest C++ is a codebase whose defaults, tools, and reviews make unsafe patterns difficult to introduce.

Synthesis: Core Guidelines + C++23/26 + verification

Appendix A - Core Guidelines Memory-Safety Matrix

P.4 Ideally, a program should be statically type safe.

Applied here: prefer prevention/checking that is automatic, early, reproducible, and tool-supported.

P.5 Prefer compile-time checking to run-time checking.

Applied here: prefer prevention/checking that is automatic, early, reproducible, and tool-supported.

P.6 What cannot be checked at compile time should be checkable at run time.

Applied here: prefer prevention/checking that is automatic, early, reproducible, and tool-supported.

P.7 Catch run-time errors early.

Applied here: prefer prevention/checking that is automatic, early, reproducible, and tool-supported.

P.8 Do not leak any resources.

Applied here: prefer prevention/checking that is automatic, early, reproducible, and tool-supported.

P.12 Use supporting tools as appropriate.

Applied here: prefer prevention/checking that is automatic, early, reproducible, and tool-supported.

Appendix A - Core Guidelines Memory-Safety Matrix

R.1 Manage resources automatically using resource handles and RAII.

Applied here: encode resource ownership in RAII/value/smart-pointer types and keep raw pointers non-owning.

R.2 Use raw pointers in interfaces to denote individual objects only.

Applied here: encode resource ownership in RAII/value/smart-pointer types and keep raw pointers non-owning.

R.3 A raw pointer is non-owning.

Applied here: encode resource ownership in RAII/value/smart-pointer types and keep raw pointers non-owning.

R.4 A raw reference is non-owning.

Applied here: encode resource ownership in RAII/value/smart-pointer types and keep raw pointers non-owning.

R.5 Prefer scoped objects; do not heap-allocate unnecessarily.

Applied here: encode resource ownership in RAII/value/smart-pointer types and keep raw pointers non-owning.

Appendix A - Core Guidelines Memory-Safety Matrix

R.10 Avoid malloc() and free().

Applied here: encode resource ownership in RAII/value/smart-pointer types and keep raw pointers non-owning.

R.11 Avoid calling new and delete explicitly.

Applied here: encode resource ownership in RAII/value/smart-pointer types and keep raw pointers non-owning.

R.12 Immediately give the result of explicit resource allocation to a manager object.

Applied here: encode resource ownership in RAII/value/smart-pointer types and keep raw pointers non-owning.

R.13 Perform at most one explicit resource allocation in a single expression statement.

Applied here: encode resource ownership in RAII/value/smart-pointer types and keep raw pointers non-owning.

R.14 Avoid [] parameters; prefer span.

Applied here: encode resource ownership in RAII/value/smart-pointer types and keep raw pointers non-owning.

Appendix A - Core Guidelines Memory-Safety Matrix

R.20 Use unique_ptr or shared_ptr to represent ownership.

Applied here: encode resource ownership in RAII/value/smart-pointer types and keep raw pointers non-owning.

R.21 Prefer unique_ptr over shared_ptr unless ownership must be shared.

Applied here: encode resource ownership in RAII/value/smart-pointer types and keep raw pointers non-owning.

R.22 Use make_shared() to make shared_ptrs.

Applied here: encode resource ownership in RAII/value/smart-pointer types and keep raw pointers non-owning.

R.23 Use make_unique() to make unique_ptrs.

Applied here: encode resource ownership in RAII/value/smart-pointer types and keep raw pointers non-owning.

R.24 Use weak_ptr to break cycles of shared_ptrs.

Applied here: encode resource ownership in RAII/value/smart-pointer types and keep raw pointers non-owning.

Appendix A - Core Guidelines Memory-Safety Matrix

R.30	Take smart pointers as parameters only to explicitly express lifetime semantics.	*Applied here: encode resource ownership in RAII/value/smart-pointer types and keep raw pointers non-owning.*
R.32	Take unique_ptr by value to express transfer of ownership.	*Applied here: encode resource ownership in RAII/value/smart-pointer types and keep raw pointers non-owning.*
R.33	Take unique_ptr& to express reseating.	*Applied here: encode resource ownership in RAII/value/smart-pointer types and keep raw pointers non-owning.*
ES.20	Always initialize an object.	*Applied here: keep object state initialized and pointer manipulation simple enough to review mechanically.*
ES.42	Keep use of pointers simple and straightforward.	*Applied here: keep object state initialized and pointer manipulation simple enough to review mechanically.*

Appendix A - Core Guidelines Memory-Safety Matrix

C.20 If you can avoid defining default operations, do.

Applied here: use Rule of Zero/defaulted operations so resource invariants live in members, not handwritten special members.

C.21 If you define or delete any copy/move/destructor, consider all.

Applied here: use Rule of Zero/defaulted operations so resource invariants live in members, not handwritten special members.

CP.23 Think of a joining thread as a scoped container.

Applied here: make thread lifetime and shared-state ownership explicit; never let detached execution silently outlive borrowed state.

CP.24 Think of a thread as a global container.

Applied here: make thread lifetime and shared-state ownership explicit; never let detached execution silently outlive borrowed state.

CP.25 Prefer a joining thread over detached std::thread.

Applied here: make thread lifetime and shared-state ownership explicit; never let detached execution silently outlive borrowed state.

Appendix B - C++23 Memory-Relevant Features

`std::out_ptr / std::inout_ptr`

Smart-pointer adaptors for foreign T** / void** APIs.

Adopt at C/COM-style output parameters so ownership moves directly into a smart pointer instead of temporary raw owning pointers.

`std::start_lifetime_as`

Explicitly start lifetime of implicit-lifetime objects in existing storage.

Low-level only: use when bytes already exist and the standard explicitly permits starting an implicit-lifetime object there.

`allocate_at_least`

Allocator may return capacity no smaller than requested.

Allocator-aware code must retain the returned count and deallocate with the same allocation contract.

`std::scope_exit / scope_fail / scope_success`

Standard lexical cleanup/rollback guards.

Use for rollback/cleanup that is lexical but does not naturally deserve a full custom RAII type.

`std::mdspan`

Non-owning multidimensional view with layout/access policy.

A view, not an owner: the underlying multidimensional storage must outlive the mdspan.

`std::expected`

Value-or-error representation that can simplify ownership on failure paths.

Useful when failure is part of ordinary control flow and ownership should remain explicit without exception-only paths.

`std::unreachable`

Optimization/semantic assertion that control cannot reach a point; violating it is dangerous.

Use only when reachability has already been proven; reaching it is not a recoverable assertion failure.

Appendix B - C++26 Memory-Relevant Features

`Contracts: pre / post / contract_assert`

Executable conditions; __cpp_contracts is listed as 202502L on cppreference.

Treat contracts as executable interface conditions, not as a replacement for ownership types, bounds-aware views, or tests.

`Standard library hardening`

Selected preconditions can be enforced by hardened implementations.

Enable the implementation hardening mode appropriate to your deployment and test both checked and production profiles.

`std::inplace_vector`

Dynamic size, fixed compile-time capacity, contiguous storage.

Use when maximum capacity is a real invariant and you want contiguous storage without dynamic allocation.

`std::indirect`

Heap-backed object with value-like semantics.

Use value-like semantics to hide dynamic storage behind an object whose ownership behavior is explicit and composable.

`std::polymorphic`

Value-like ownership for dynamic polymorphic objects.

Use value-like semantics to hide dynamic storage behind an object whose ownership behavior is explicit and composable.

`std::owner_hash / std::owner_equal`

Hash/equality by shared ownership identity.

Use when associative containers must reason about shared ownership identity rather than raw pointer value.

`<hazard_pointer>`

Standard hazard-pointer reclamation support.

Concurrency infrastructure only: pair the reclamation mechanism with a documented publication/removal protocol.

`<rcu>`

Standard Read-Copy Update support.

Concurrency infrastructure only: pair the reclamation mechanism with a documented publication/removal protocol.

`std::is_within_lifetime`

Specialized lifetime query utility.

Specialized utility for low-level lifetime reasoning; keep it close to the storage-management boundary.

`std::is_sufficiently_aligned`

Alignment query before making stronger assumptions.

Check before strengthening an alignment assumption; an incorrect assumption can turn optimization into undefined behavior.

`std::hive`

Bucket-based container that reuses erased-element storage.

Useful when stable/reused element storage better matches the workload than repeated node allocation and destruction.

Appendix C - Toolchain Recipes

ASan + UBSan

COMPILER / CI — SELECT • COPY • PASTE

```
-fsanitize=address,undefined -fno-omit-frame-pointer -g -O1
```

WHAT IT GIVES YOU

Catches heap/stack out-of-bounds, use-after-free, double-free classes and many forms of undefined behavior when combined with UBSan.

HOW TO DEPLOY IT

Use on unit/integration tests and fuzz targets. Keep symbols and frame pointers for actionable traces.

MemorySanitizer

COMPILER / CI — SELECT • COPY • PASTE

```
-fsanitize=memory -fno-omit-frame-pointer -g -O1
```

WHAT IT GIVES YOU

Detects reads of uninitialized memory. It is primarily a Clang workflow and is most effective when dependencies are instrumented too.

HOW TO DEPLOY IT

Run a dedicated job where toolchain support exists; do not treat an uninstrumented dependency boundary as fully covered.

ThreadSanitizer

COMPILER / CI SELECT • COPY • PASTE

```
-fsanitize=thread -g -O1
```

WHAT IT GIVES YOU

Detects data races and synchronization mistakes that can become lifetime and memory-safety defects.

HOW TO DEPLOY IT

Use on representative concurrent tests. Expect a large slowdown and keep it as a separate CI configuration.

Warnings baseline

COMPILER / CI SELECT • COPY • PASTE

```
-Wall -Wextra -Wpedantic
```

WHAT IT GIVES YOU

Compiler warnings are the cheapest continuous signal for suspicious conversions, lifetime mistakes, unused results, and API misuse.

HOW TO DEPLOY IT

Keep warnings clean in normal development builds; raise project-specific warnings deliberately rather than blindly enabling everything.

CMake policy

COMPILER / CI SELECT • COPY • PASTE

```
Use separate sanitized presets; keep sanitizer configuration in version control.
```

WHAT IT GIVES YOU

A reproducible safety profile belongs in the build system, not in a developer notebook.

HOW TO DEPLOY IT

Create named presets for warnings, sanitizers, analysis, and release so every engineer and CI agent runs the same configuration.

Fuzzing policy

COMPILER / CI SELECT • COPY • PASTE

```
Build fuzz targets with ASan/UBSan and keep regression corpora in CI.
```

WHAT IT GIVES YOU

Fuzzing drives unusual inputs through parsers, decoders, and state machines where bounds and lifetime assumptions often fail.

HOW TO DEPLOY IT

Combine fuzzing with ASan/UBSan and retain crashing inputs as regression tests.

Appendix D - Release and Review Checklists

Ownership

Every resource has one explicit owner model; raw pointers/references are non-owning by project policy.

EVIDENCE TO REQUIRE

Evidence: API review shows owners in values/RAII/smart pointers; no raw owning returns; teardown path is explicit.

Lifetime

Every borrow is shorter than the owner lifetime, and invalidating mutations are known.

EVIDENCE TO REQUIRE

Evidence: invalidating operations are documented; long-lived borrows are justified; lifetime/sanitizer tests cover risky paths.

Bounds

Sequence interfaces retain bounds; external indexes and lengths are validated.

EVIDENCE TO REQUIRE

Evidence: pointer+length pairs are replaced by span/ranges where possible; external sizes are validated before indexing.

Allocation

No naked new/delete outside approved resource-management infrastructure.

EVIDENCE TO REQUIRE

Evidence: explicit new/delete is confined to reviewed infrastructure; allocation failure and deallocation pairing are tested.

Exceptions

Throwing mutations preserve documented invariants; RAII cleans temporary resources.

EVIDENCE TO REQUIRE

Evidence: throwing operations have a documented guarantee; rollback is RAII/scope-guarded; partially built state cannot escape.

Concurrency

Every cross-thread access has a happens-before argument and, when lock-free, a reclamation strategy.

EVIDENCE TO REQUIRE

Evidence: synchronization establishes the required happens-before relation; lock-free removal uses hazard/RCU/other safe reclamation.

Interop

Foreign handles are wrapped immediately and released by the matching API.

EVIDENCE TO REQUIRE

Evidence: each foreign handle has one wrapper/deleter; ownership transfer is documented; exceptions do not cross incompatible ABIs.

Appendix D - Release and Review Checklists

Low-level

Casts, pointer arithmetic, raw storage, and lifetime manipulation are isolated and justified.

EVIDENCE TO REQUIRE

Evidence: casts/raw storage/pointer arithmetic are isolated, commented with preconditions, and exercised under sanitizers.

Tools

Warnings clean; sanitizer/static-analysis/fuzz jobs pass.

EVIDENCE TO REQUIRE

Evidence: clean warning build plus ASan/UBSan, targeted TSan/MSan where supported, static analysis, and regression tests.

C++26

Feature-test guards and fallback semantics are explicit for partially supported facilities.

EVIDENCE TO REQUIRE

Evidence: feature-test macros or toolchain matrix prove support; fallback behavior is explicit; hardening/contracts are CI-tested.

Appendix E - Patterns and Counterpatterns

GOOD

Value-returning factory

Returns T; ownership is obvious and often allocation-free.

REVIEW / REFACTOR CUE

Review cue: ask whether dynamic allocation is actually required before introducing an owning pointer.

GOOD

unique_ptr transfer

Takes unique_ptr<T> by value to consume ownership.

REVIEW / REFACTOR CUE

Review cue: the move at the call boundary should make consumption of ownership obvious.

GOOD

Borrowing span

Takes span<const byte> instead of pointer + length.

REVIEW / REFACTOR CUE

Review cue: verify that the owner outlives the span and that no pointer/size split remains.

GOOD

RAII C handle

Custom wrapper closes a foreign resource automatically.

REVIEW / REFACTOR CUE

Review cue: constructor/acquisition and destructor/release must use the exact matching foreign API.

GOOD **PMR phase arena**

monotonic_buffer_resource lifetime matches request/frame lifetime.

REVIEW / REFACTOR CUE

Review cue: resource lifetime must dominate every container/object that allocates from it.

GOOD **Transactional update**

Build new state, validate it, then swap/commit.

REVIEW / REFACTOR CUE

Review cue: failure before commit must leave the previous valid state intact.

BAD **shared_ptr everywhere**

Reference counting without genuine shared-lifetime semantics.

REVIEW / REFACTOR CUE

Refactor cue: identify the independent owners. If there is only one, use unique ownership and borrow.

BAD **Pointer cached into vector**

A long-lived pointer survives a possible reallocation.

REVIEW / REFACTOR CUE

Refactor cue: store an index/iterator only when its invalidation rules are controlled, or reacquire after mutation.

BAD

release() into nowhere

unique_ptr ownership is discarded without an immediate new owner.

REVIEW / REFACTOR CUE

Refactor cue: transfer immediately to another owner; a naked released pointer is an ownership gap.

BAD

Detached thread borrows local

Task may access an object after scope exit.

REVIEW / REFACTOR CUE

Refactor cue: join, use jthread, or move owned state into the task so no local borrow can dangle.

BAD

reinterpret_cast serialization

Native representation is treated as portable wire layout.

REVIEW / REFACTOR CUE

Refactor cue: define a wire format and use bit_cast/memcpy/explicit serialization under documented representation rules.

BAD

Lock-free delete-now

Node is freed immediately after unlink without safe reclamation.

REVIEW / REFACTOR CUE

Refactor cue: unlinking and freeing are different phases; introduce a proven reclamation mechanism before delete.

Appendix F - Glossary

RAII

Resource Acquisition Is Initialization: resource lifetime is bound to object lifetime.

Owner

Object/value responsible for releasing a resource.

Borrow

Non-owning access whose validity depends on another object.

Dangling

Pointer/reference/view that refers to an object outside its lifetime.

Spatial safety

Access remains within the bounds of the intended object/region.

Temporal safety

Access occurs only while the object is alive.

Control block

Shared ownership metadata used by shared_ptr and weak_ptr.

Arena

Memory domain where many allocations are released together.

Pool

Allocator that serves requests from reusable groups of blocks.

Fragmentation

Waste/unusable capacity caused by allocation/layout patterns; the metric must be defined.

Happens-before

C++ memory-model relation establishing ordering/visibility across evaluations.

Hazard pointer

Reader protection record that delays reclamation of a node.

RCU

Read-Copy Update: read-mostly synchronization/reclamation via version replacement and grace periods.

PMR

Polymorphic memory resources: runtime-selectable allocation strategies.

Hardened library

C++26 model where selected standard-library preconditions can be checked by an implementation.

Contract

C++26 precondition, postcondition, or contract_assert condition.

Unsafe boundary

Small audited region where low-level operations bypass normal safe abstractions.

Storage duration

How long storage exists: automatic, static, thread, or dynamic.

Object lifetime

Interval during which an object exists and may be accessed according to the language rules.

Invalidation

Operation that makes a pointer, reference, iterator, or view no longer valid.

Observer

Non-owning handle used to access an object without controlling its lifetime.

Ownership transfer

Deliberate movement of responsibility for releasing a resource to another owner.

Ownership cycle

Graph of shared owners that keeps objects alive after external ownership disappears.

Raw storage

Bytes with suitable size/alignment in which no object may yet be alive.

Placement construction

Constructing an object in previously obtained storage; construction is distinct from allocation.

Alignment

Address constraint required by a type or optimized operation.

Allocation domain

Region/resource whose allocations follow one lifetime or release policy.

Scope guard

RAII helper that executes cleanup, rollback, or success action at scope exit.

Reclamation

Making removed concurrent objects safe to destroy after readers can no longer access them.

Data race

Conflicting unsynchronized accesses to the same memory location where at least one access writes.

False sharing

Performance interference when independent hot data shares cache lines across cores.

Strong exception guarantee

A failed operation leaves observable state unchanged.

Basic exception guarantee

A failed operation preserves invariants and prevents resource leaks, though state may change.

std::span

Non-owning contiguous view that carries pointer and element count together.

std::string_view

Non-owning view of character data; the referenced characters must outlive the view.

std::mdspan

Non-owning multidimensional view with explicit extents/layout/accessor policy.

memory_resource

PMR interface that provides allocation/deallocation behavior behind polymorphic allocators.

Upstream resource

Fallback memory_resource used when a PMR resource cannot satisfy an allocation locally.

Use-after-free

Access through a pointer/reference after the owned storage has been released.

Double free

Releasing the same allocation twice or through mismatched ownership paths.

Out-of-bounds

Reading or writing outside the valid extent of an object or sequence.

Leak

Resource remains allocated because no live owner can release it.

References and Standards Notes

C++ Core Guidelines

https://isocpp.github.io/CppCoreGuidelines/CppCoreGuideli

Living guidelines edited by Bjarne Stroustrup and Herb Sutter; browsable edition updated June 14, 2026.

cppreference - Memory management

https://en.cppreference.com/cpp/memory

Smart pointers, allocators, PMR, explicit lifetime, and C++26 memory facilities.

cppreference - C++23

https://en.cppreference.com/cpp/23

C++23 language/library feature index.

cppreference - C++26

https://en.cppreference.com/cpp/26

C++26 feature index and implementation notes.

cppreference - Standard library hardening

https://en.cppreference.com/cpp/standard_library

C++26 hardened implementation model and hardened preconditions.

Clang AddressSanitizer

https://clang.llvm.org/docs/AddressSanitizer.html

Dynamic detection of out-of-bounds, use-after-free, invalid free, and related errors.

Clang UndefinedBehaviorSanitizer

https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.ht

Runtime detection of many undefined behaviors.

Clang MemorySanitizer

https://clang.llvm.org/docs/MemorySanitizer.html

Detection of uninitialized memory use.

Clang ThreadSanitizer

https://clang.llvm.org/docs/ThreadSanitizer.html

Data-race detector.

GCC Instrumentation Options

https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Option

GCC sanitizer and instrumentation options.

P2900R14

WG21 Contracts for C++26

Contracts proposal adopted for C++26.

P2530R3

WG21 Hazard Pointers

Hazard pointer support for C++26.

P2545R4

WG21 Read-Copy Update

RCU support for C++26.

P3471R4 and related papers

WG21 Standard Library Hardening

C++26 standard library hardening work.

Topic Index

Topic Index

Memory safety in C++ is an engineering discipline.

Own resources explicitly. Borrow briefly. Preserve bounds. Make lifetime visible. Isolate unsafe operations. Verify continuously. Use C++23 and C++26 facilities where they clarify and harden the design.

Prepared by Ayman Alheraki

Third Edition - August 2026