ISO C++ Core Guidelines Condensed

DRAFT

The Engineer's On-Desk Reference

All major rules with rationale, examples, and pitfalls

Prepared by Ayman Alheraki

2026

Based on the ISO C++ Core Guidelines (CppCoreGuidelines)

ISO C++ Core Guidelines Condensed

The Engineers OnDesk Reference

All major rules with rationale, examples, and pitfalls

Prepared by Ayman Alheraki

2026 Based on the ISO C++ Core Guidelines (CppCoreGuidelines)

Contents

Author's Introduction

C++ does not forgive assumptions it exposes intent, rewards discipline, and amplifies both skill and negligence. For over four decades it has powered systems where correctness, performance, and longevity are nonnegotiable: operating systems, compilers, databases, financial engines, embedded platforms, and scientific software.
The ISO C++ Core Guidelines exist because realworld failures rarely come from obscure language rules; they come from inconsistent ownership, unsafe interfaces, ambiguous lifetimes, and architectural shortcuts that quietly accumulate risk.
This book works from one principle:
Professional C++ engineering is not about knowing syntax. It is about consistently applying proven rules that prevent entire classes of errors before they occur.
The Guidelines are not a style guide. They are distilled knowledge extracted from decades of realworld failures, postmortems, largescale codebases, and languagedesign experience. They answer questions every serious developer eventually faces:

- Who owns this object, and for how long?
- What assumptions does this interface impose?
- Which errors are prevented by construction, which are deferred to runtime?
- Where does performance matter, and where does safety dominate?
- How do we write code that survives maintenance, refactoring, and scale?

The Guidelines provide a coherent framework for using modern C++ in a way that is explicit, verifiable, and reviewable. They unify language rules, ownership semantics, concurrency models, and error handling under one engineering philosophy:

> Make correctness the default. Make misuse difficult. Make intent visible. Make performance predictable.

This book exists because the Guidelines themselves are dense and crossreferenced. Here, each rule is treated as part of a larger system: language rules connect to the memory model, API design to ownership, concurrency to datarace prevention, and performance to real architectural tradeoffs.

The volume is substantial because professional C++ is substantial. There are no shortcuts that replace understanding only disciplined engineering. Whether you maintain legacy code, design new APIs, review critical infrastructure, teach C++, or build systems that must remain correct for decades, the Core Guidelines offer a shared technical language.

This book is not meant to be read once. It is meant to be consulted, argued with, and applied.

Its goal is simple:

To help you write code that is not merely clever or fast, but correct, maintainable, and worthy of longterm trust.

`Ayman Alheraki`

Book's Introduction

What Are the ISO C++ Core Guidelines?

The **C++ Core Guidelines** are a curated, living set of rules and best practices for modern ISO C++. They help developers write safer, simpler, and more maintainable code while preserving performance and zerooverhead principles. Maintained by community editors (notably Bjarne Stroustrup and Herb Sutter), they are **not** the ISO C++ Standard, but practical guidance on *how* to use the standard effectively.

Why They Exist C++s power invites accidental complexity, undefined behavior, and brittle ownership. The Guidelines:

- Establish consistent, modern defaults.
- Reduce common defects especially **lifetime**, **bounds**, and **type** mistakes.
- Encourage scalable design: explicit ownership, clear interfaces, predictable resource management.
- Make code amenable to automated checking.

Organization and Profiles Rules are grouped into themes (philosophy, interfaces, resource management, concurrency, etc.). Many are grouped into **safety profiles**:

- **Type safety** (avoid unsafe casts, narrowing)
- **Bounds safety** (prevent outofrange access)
- **Lifetime safety** (prevent dangling references, useafterfree)

Profiles allow incremental adoption and are frequently targeted by static analysis tools.

Guidelines Support Library (GSL) To express intent more directly, the GSL provides lightweight helpers:

- `gsl::not_null<T>` nonnull pointer contract
- `gsl::span<T>` range view with explicit size

These make interfaces selfdocumenting and misuse harder.

Enforcement in Practice Tool support is a primary goal. Major tools include:

- **clangtidy** with `cppcoreguidelines-*` checks.
- **Microsoft C++ Code Analysis** (C++ Core Guidelines Checkers) aligned with profiles.

Adoption is most effective when guidelines become part of codereview vocabulary, enforced incrementally.

What This Book Is This book presents the Guidelines as a coherent engineering system: intent, rationale, correct usage, violations, fixes, and enforcement notes all in a consistent rulecentered template.

Objectives of This Reference

This reference exists to provide a single, consistent, engineeringfocused way to read, apply, and enforce the Core Guidelines.

Primary Objectives

- **Complete ruleoriented reference** follows canonical structure and preserves each rules identity.
- **Translate rules into practice** for each guideline group: motivation, safe modern alternative, typical failure patterns, migration strategy.
- **Focus on highimpact safety profiles** lifetime, bounds, and type safety (the most expensive bug families).
- **Make guidelines enforceable** map rules to clangtidy and MSVC checkers, showing how to apply and suppress checks selectively.

Scope and NonGoals The book covers the Guidelines as an engineering system, not as a replacement for the C++ standard or a guarantee of bugfree code. It treats them as wellmotivated best practices to be adopted gradually.

Book Organization Rulecentered chapters follow a standard template: Intent, Risk Addressed, Preferred Modern Form, Bad vs. Good Examples, Enforcement Notes, and Migration Advice. The emphasis is on making intent explicit (nonnull, range views, RAII).

How to Use This Book as a Reference

Two Reading Modes

- **Study mode** read early chapters for fundamentals, then explore major guideline families.
- **Lookup mode** jump directly from a bug class, review comment, or static-analysis warning to the matching rule section.

Standard Rule Template Every rule section includes:

- **Intent** onesentence goal.
- **Why It Matters** defect class addressed.
- **Rule Statement** and scope.
- **Bad vs. Good examples** minimal, realistic transformations.
- **Enforcement Notes** what tools can and cannot check.
- **Migration Strategy** safe stepbystep for legacy code.

Common Entry Points

- **Code review:** unclear ownership resource/lifetime chapters; unchecked indexing bounds safety; narrowing casts type safety.

- **Bug fixing:** map bug to defect class (lifetime, bounds, type) apply corresponding guideline.
- **Staticanalysis warnings:** look up warning family, identify intent, apply recommended transformation, rerun analyzer.

Tool Integration

- **clangtidy:** enable `cppcoreguidelines-*` checks incrementally; use `NOLINT` for justified deviations.
- **Visual Studio:** use C++ Core Guidelines checkers; suppress specific warnings with `/wd` or `[[gsl::suppress]]`.

Adoption Plan

1. Phase 1: Apply safe defaults to new code (RAII, explicit ownership, avoid raw pointer arithmetic).
2. Phase 2: Adopt highimpact safety topics (lifetime, bounds, type).
3. Phase 3: Turn on enforcement gradually, starting with the most defectreducing checks.

Guidelines vs. Mandatory Rules

Four Kinds of Rules

- **A: Languagemodel requirements** (nonnegotiable; avoid UB, respect lifetime, ODR).
- **B: Contractual/regulatory requirements** (projectspecific policies, safety standards).
- **C: Core Guidelines recommendations** (expert defaults to reduce defects).
- **D: Local style conventions** (naming, formatting lower safety impact).

Key Distinction

- **Mandatory** means must breaking it risks undefined behavior or nonconformance.
- **Guideline** means should a default to follow, but deviations are allowed with documented justification.

When Guidelines Become Mandatory Teams may promote selected guidelines to mandatory policy (e.g., a profile enforced in CI). The Guidelines were designed with this in mind.

Examples Legal C++ that compiles may still violate a guideline: e.g., passing a possiblynull pointer without a nonnull contract. A guidelinedriven interface makes the precondition explicit via `gsl::not_null`.
Mandatory rule violation: outofbounds access is undefined behavior not a style issue. The guideline helps design interfaces that prevent it.

When and Why a Rule May Be Broken

Core Principle First, classify the rule:

- **Mandatory** do not break.
- **Guideline** may be broken with disciplined engineering justification.

Valid Reasons for Deviation

- **Compatibility constraints** (ABI, legacy C APIs) confine unsafe boundary code, convert to safe internals immediately.
- **Proven performance constraint** backed by benchmarks; isolate exception, keep safe design elsewhere.
- **Incremental migration** apply safer defaults to new code; gradually modernize highrisk areas.
- **Tool limitations / false positives** suppress narrowly.

Invalid Reasons We always do it this way, it compiles, or breaking rules that prevent undefined behavior.

Professional Deviation Procedure

1. Identify the rule and its intent.
2. Prove the constraint (ABI, performance data, migration risk).

3. Choose the safest alternative that preserves the intent.
4. Localize, isolate, and document the exception.
5. Keep enforcement enabled everywhere else.

Documenting Deviations A comment should include: rule/check suppressed, reason, mitigation, exit plan (if temporary). Use controlled suppression:

- `// NOLINT(cppcoreguidelines-...)` or `// NOLINTNEXTLINE` for clangtidy.
- `[[gsl::suppress(...)]]` or compilerspecific warning suppression for MSVC.

When Breaking a Rule Is Good Engineering The exception preserves the guidelines intent via boundary conversion, encapsulation, or explicit contracts. It is reviewed, owned, measurable, and testable.

Summary Checklist A guideline may be broken only when:

1. It is a guideline, not a mandatory requirement.
2. The constraint is real and documented.
3. The exception is localized and encapsulated.
4. Risk is mitigated (checks, tests, contracts).
5. Tooling suppression is narrow and justified.

Part I

Philosophy (P)

CHAPTER 1

P.1 – P.12

1.1 Purpose and Reading

The Philosophy rules set the conceptual foundation: they guide design choices toward code that is safe, clear, and checkable. They are principles, not microrules; many are not fully enforceable by tools, but they lead to checkable code.

1.2 The Rules

P.1: Express ideas directly in code.
Use constructs that mirror the design intent. Prefer `std::find` over a hand-written loop, explicit types for domain concepts, and `const` to signal readonly access.

P.2: Write in ISO Standard C++.
Stick to standard C++. If extensions are unavoidable, encapsulate them behind portable interfaces and keep the rest of the code standard.

P.3: Express intent.
Code should make intent visible. Use range`for` rather than indexbased loops, `span` instead of pointer+size, and strong types instead of builtin types where meaning matters.

P.4: Ideally, a program should be statically type safe.
Avoid `reinterpret_cast`, Cstyle casts, unions, and unnecessary narrowing. Prefer `variant`, `span`, and `narrow_cast` when narrowing is required.

P.5: Prefer compiletime checking to runtime checking.
Use `static_assert`, `constexpr`, and typerich interfaces. Move errors to build time.

P.6: What cannot be checked at compile time should be checkable at run time.
Avoid hiding size information; use sizecarrying types like `span` or `vector` so that bounds checks are possible.

P.7: Catch runtime errors early.
Validate at boundaries, not deep inside computation. Use types that allow early checking without changing algorithmic complexity.

P.8: Dont leak any resources.
Use RAII for every resource (memory, files, locks). Avoid naked `new`/`delete` and `malloc`/`free`.

P.9: Dont waste time or space.
Efficiency matters, but time and space spent for safety, clarity, and testability are not waste. Prefer move semantics, avoid gratuitous copies, and use prefix `++` where appropriate.

P.10: Prefer immutable data to mutable data.
Make objects `const` by default, use `constexpr`, and design APIs that minimize shared mutable state. Immutability simplifies reasoning and concurrency.

P.11: Encapsulate messy constructs.
Wrap lowlevel complexity (raw memory, pointer manipulation, casting) inside welldesigned interfaces. Use standard containers and algorithms instead of handcrafted loops.

P.12: Use supporting tools as appropriate.
Run static analyzers (clangtidy, MSVC Core Checkers) regularly. Let machines enforce repetitive checks; keep toolchains portable.

Note

Summary Checklist

- Code meaning directly (P.1, P.3).
- Use standard, portable C++ (P.2).
- Push correctness into types and compile time (P.4, P.5).
- Design for feasible runtime checks (P.6, P.7).
- RAII for all resources (P.8).
- Respect performance without sacrificing safety (P.9).
- Prefer immutability (P.10).
- Encapsulate lowlevel mess (P.11).
- Automate enforcement with tools (P.12).

CHAPTER 2

Design Philosophy of C++

C++ is engineered around a few core commitments:

- **Direct map to hardware** operations and object layouts map to machine concepts.
- **Zerooverhead abstraction** you dont pay for what you dont use; abstractions are as efficient as handwritten code.
- **Deterministic resource management (RAII)** destructors guarantee cleanup even with exceptions.
- **Strong static typing** intent and invariants are expressed in types, enabling compiletime checks.
- **Multiparadigm flexibility** procedural, OO, generic, and functional styles coexist.
- **Compiletime guarantees** shift errors left with `static_assert`, `constexpr`, and template constraints.
- **Safety through engineering** safety comes from patterns, libraries, and tools, not from a separate safe dialect.
- **Encapsulation of lowlevel complexity** confine dangerous operations inside safe interfaces.
- **Continuous evolution with stability** modernize while protecting existing code.

Zerooverhead in a nutshell

1. You dont pay for what you dont use.
2. What you do use is as efficient as a handcrafted lowlevel implementation.

This drives the preference for templates, inlining, and move semantics.

RAII Example

```cpp
#include <fstream>
#include <string>
std::string read_first_line(const std::string& path) {
    std::ifstream in(path);   // resource acquired
    std::string line;
    std::getline(in, line);
    return line;              // automatically released
}
```

CHAPTER 3

Zerooverhead Abstraction

Definition: Abstractions impose no runtime cost beyond what a handwritten version would require. The compiler can eliminate abstraction overhead.

How C++ achieves it

- **Compiletime composition:** templates and `constexpr` generate specialized code.
- **RAII:** deterministic cleanup without hidden bookkeeping.
- **Move semantics:** avoid unnecessary copies.

Where costs appear

- **Dynamic polymorphism** virtual functions add indirection (intentional, not hidden).
- **Type erasure** (`std::function`) may allocate and dispatch indirectly.
- **Safety checks** bounds checks or precondition validation have a cost; the Core Guidelines consider this acceptable if the cost is under control.

Rules of thumb

- Prefer abstractions the compiler can see through (templates, `constexpr`).
- Measure before microoptimising; dont guess.
- Encapsulate paying abstractions at system boundaries; keep hot loops free of unnecessary indirection.

CHAPTER 4

Expressing Intent Clearly

Intent is expressed when a reader (and tool) can see what the code does, what it assumes, and what it guarantees.

Core techniques (with examples)

1. Use descriptive names.
`int timeout_ms = 250;` beats `int t = 250;`.

2. Strong types for domains.
`struct Milliseconds { int64_t value; };` prevents mixing with bytes.

3. `const` for nonmutation.
`int count_digits(const std::string& s);` signals readonly.

4. RAII for ownership.
Objects own resources; destructors release them. Makes leaks impossible by construction.

5. `span` instead of pointer+size.
`int sum(gsl::span<const int> xs);` size travels with the view.

6. `not_null` for nonnull contracts.
`void write_value(gsl::not_null<int*> p);` null is rejected at the call site.

7. Scoped enumerations.
`enum class Mode { read_only, read_write };` typesafe and intentionrevealing.

8. Named constants instead of magic numbers.
`constexpr size_t kMaxPacketBytes = 4096;` meaning is clear.

9. Algorithms and rangefor over manual loops.
`std::any_of` tells you is there any? instantly.

10. Keep scopes small.
Declare loop variables in the `for` initialiser, and variables near first use.

Tooling

Both `clang-tidy` (`cppcoreguidelines-*`) and Microsoft C++ Core Guidelines checkers enforce many of these patterns. Expressing intent in types also makes automated checking more effective.

Review checklist

1. Who owns each resource and for how long?
2. Are nullability and bounds explicit?
3. Are units and domain meanings in names/types?
4. Could a maintainer misuse this interface unnoticed?
5. Can tools check the critical assumptions?

CHAPTER 5

Making Invalid States Unrepresentable

Core idea: Design types so that invalid states cannot be expressed. This shifts bugs from runtime to compile time (P.1, P.3, P.4, P.5).

Principle 1: Put meaning in types, not comments. Use strong types for units, roles, and constraints.

Principle 2: Narrow interfaces to the valid domain. If an input can be invalid, represent that explicitly (`std::optional`, error channels).

Principle 3: Carry required context together. Replace pointer+size with `span`; carry size with the view.

Patterns that encode validity

Scoped enumerations instead of integers. `enum class Mode { read_only, read_write };` impossible to pass a raw number.

Strong types for units. `struct Milliseconds { int64_t value; };` prevents mixing with Bytes.

Nonnull by construction (`gsl::not_null`). `void write_value(gsl::not_null<int*> p);` null is rejected at the call.

Sizecarrying views (`gsl::span`). `int sum(gsl::span<const int> xs);` bounds information is part of the interface.

Optional values (`std::optional`) instead of sentinels. `std::optional<std::string> find_user_name(int id);` absence is explicit.

Closed alternatives (`std::variant`) instead of tagged unions. `using Token = std::variant<int, double, std::string>;` only one valid alternative.

Expected or error (`std::expected`) for recoverable failures. `std::expected<int, ParseError> parse_int(std::string_view s);` prevents silent misuse of error codes.

Encoding invariants: private constructor + factory

Example

```
class Port {
    std::uint16_t value_;
    explicit Port(std::uint16_t v) : value_(v) {}
public:
    static std::expected<Port, const char*> make(int v) {
        if (v < 0 || v > 65535) return std::unexpected("out of range");
        return Port(static_cast<std::uint16_t>(v));
    }
    std::uint16_t value() const noexcept { return value_; }
};
```

`Port` outside 065535 cannot exist.

Note

Checklist

- Use `enum class` for categories.
- Use strong types for units/roles.
- Prevent null with `not_null` or references.
- Carry size with views (`span`), not pointer arithmetic.
- Use `optional` for absence, not sentinels.
- Use `expected` for fallible operations.
- Enforce invariants at construction.

CHAPTER 6

Clarity versus Cleverness

Clarity means intent, assumptions, and consequences are obvious. **Cleverness** relies on subtle language rules and is expensive to maintain. The Core Guidelines demand clarity (P.1, P.3) and explicitly warn against complicated expressions (ES.40, ES.41, ES.43, ES.44).

Guideline anchors

- ES.1: Prefer the standard library to handcrafted code.
- ES.40: Avoid complicated expressions.
- ES.41: Parenthesize when precedence is not obvious.
- ES.43/44: Avoid undefined/unspecified order of evaluation.
- ES.30/31: Dont use macros for constants or functions.

Common clever traps and clear alternatives

Complicated expressions

```
// Dont
if ((x=f()) && (y=g(x)) && (z=h(y))) { use(z); }
// Do
x = f(); if (!x) return;
y = g(x); if (!y) return;
z = h(y); use(z);
```

Precedence tricks

```
// Dont: if (a & b == 0)
// Do:   if ((a & b) == 0)
```

Undefined/unspecified evaluation order

```
// Dont: v[i] = ++i;  // UB before C++17, still risky
// Do:   ++i; v[i] = i;
// Dont: f(++i, ++i); // argument order unspecified
// Do:   int a=++i; int b=++i; f(a,b);
```

Clever loops hide intent

```
// Dont: handcrafted sum loop
// Do:   auto sum = std::accumulate(v.begin(), v.end(), 0.0);
```

Modifying loop index inside the body

```
// Dont: for (int i=0; i<n; ++i) { if (skip(i)) ++i; ... }
// Do:   make stepping explicit, or use while.
```

Macro cleverness

```
// Dont: #define SQUARE(x) ((x)*(x))  // SQUARE(i++) increments twice
// Do:   constexpr int square(int x) noexcept { return x*x; }
```

When clever is acceptable

Only when its local, measured, documented, and behind a clear interface. If a reviewer must simulate evaluation order to understand a line, it fails clarity.

CHAPTER 7

Engineering Simplicity

Simplicity means obvious intent, easy correctness reasoning, low maintenance cost, and visible performance costs. It aligns with P.1, P.3, P.8, P.11 and the preference for library abstractions.

Principles

1. Make the valid path the easy path. Design so that correct usage is the simplest usage (strong types, nonnull, sizecarrying views).

2. Keep invariants local. Constructors establish invariants; public member functions preserve them.

3. Prefer narrow interfaces with clear contracts. References, `span`, `optional`, `expected` communicate contracts directly.

4. Separate policy from mechanism. Encapsulate lowlevel details; expose highlevel intent.

5. Use tools to maintain simplicity. Static analysis, clangtidy, and Core Guidelines checkers prevent drift.

Complexity traps and simple alternatives

Pointer + length `span`

```
int sum(gsl::span<const int> xs); // size travels with view
```

Manual cleanup RAII

```
std::ifstream in(name); // automatic close, no leak
```

Sentinel values `std::optional`

```
std::optional<int> find_id(std::string_view name);
```

Overly clever expressions named steps (see Clarity chapter)

Patterns that scale

- Small, singlepurpose functions.
- Convert at the boundary: encapsulate legacy/complex interfaces and translate to safe internals immediately.
- Strong types for domain values.
- Standard algorithms when they express intent.
- Localise performancesensitive code; keep general code simple.

Note

Review checklist

1. Is intent clear from the function signature?
2. Is ownership explicit (RAII, no hidden cleanup)?
3. Are invalid states unrepresentable?
4. Is complexity encapsulated?
5. Are standard abstractions used where they fit?
6. Are tools (static analysis, formatters) employed to prevent drift?

Part II

Interfaces (I)

CHAPTER 8

I.1 – I.26

The **Interfaces** rules define how to build safe, clear, and stable contracts between parts of a C++ program. A good interface makes correct use easy, incorrect use hard, and is amenable to tool enforcement.

Rule Overview

I.1: Make interfaces explicit. Avoid hidden global state; pass everything through parameters and return values.

I.2: Avoid `nonconst` global variables. Prefer `const`/`constexpr` for immutables; pass mutable state explicitly.

I.3: Avoid singletons. They are disguised global objects; prefer explicit dependency injection.

I.4: Make interfaces precisely and strongly typed. No `void*`, no weakly typed bags of primitives; use strong, domainspecific types.

I.5: State preconditions (if any). Document what must be true before the call; this is part of the contract.

I.6: Prefer `Expects()` for expressing preconditions. Use GSLs `Expects(condition)` to make preconditions toolcheckable.

I.7: State postconditions. Document what the function guarantees on return.

I.8: Prefer `Ensures()` for expressing postconditions. Use `Ensures(condition)` to make postconditions explicit and checkable.

I.9: If an interface is a template, document its parameters using concepts. C++20 concepts turn documentation into compilerenforced constraints.

I.10: Use exceptions to signal a failure to perform a required task. Exceptions are nonignorable; for expected alternative outcomes, use `std::expected` or `std::optional`.

I.11: Never transfer ownership by a raw pointer (`T*`) or reference (`T&`). Use `std::unique_ptr` for exclusive ownership, `std::shared_ptr` for shared ownership; raw pointers are for nonowning observation.

I.12: Declare a pointer that must not be null as `not_null`. Use `gsl::not_null` to enforce nonnull contracts at the type level.

I.13: Do not pass an array as a single pointer. Always pass size information; use `std::span` or `gsl::span` to carry bounds.

Note

I.14–I.21 are reserved / not present in the current rule set.

I.22: Avoid complex initialization of global objects. Global initialization order across translation units is fragile. Prefer local objects or firstuse functionlocal statics.

I.23: Keep the number of function arguments low. Too many parameters indicate a complex interface; group related parameters into a configuration type.

I.24: Avoid adjacent parameters that can be swapped with different meaning. Use strong types (e.g., `Width`, `Height`) or a parameter object to prevent accidental swaps.

I.25: Prefer empty abstract classes as interfaces to class hierarchies. Stateless interfaces are more stable and reduce coupling.

I.26: If you want a crosscompiler ABI, use a Cstyle subset. For binary compatibility across compilers, use `extern "C"` functions and simple data layouts.

CHAPTER 9

Designing Clear Interfaces

A clear interface expresses its contract through types and names, making misuse difficult and correct use natural.

Design goals:

- Contracts are explicit (no hidden dependencies).
- Correct use is the easiest path.
- Invalid states are unrepresentable or rejected early.
- The interface is checkable by compilers and static analysis.
- It remains stable under maintenance.

Encoding Intent in Types

Replace weak types with strong ones:

- `enum class` for option sets
- `std::chrono::duration` for time
- `std::string_view` for readonly character views
- `std::span` for contiguous ranges
- `std::optional` / `std::expected` for absence/failure
- `gsl::not_null` for required nonnull pointers

Parameter shapes

- Prefer the most specific type (I.4).

- Avoid adjacent sametype parameters with different meaning; use strong types or a parameter struct (I.24).
- Keep argument count low; group cohesive parameters into a configuration type (I.23).
- For ranges, use `span` it carries size (I.13). (clangtidy flags pointer arithmetic and recommends `span`.)

Ownership and nullability

- Never transfer ownership with raw pointers/references; use smart pointers (I.11).
- If null is invalid, make it a typelevel requirement with `gsl::not_null` (I.12).

Contracts

Use `Expects()` and `Ensures()` (from GSL) to state pre and postconditions (I.5I.8). Precondition violations are meant to terminate, not to be handled as ordinary errors.

Error handling

- Use exceptions for failures to perform a required task (I.10).
- Mark mustcheck return values with `[[nodiscard]]`.

Readonly observation

Use `std::string_view` when the callee only reads a string; document that the viewed data must outlive the view.

Generic interfaces

Document template requirements with C++20 concepts (I.9). This turns informal documentation into compilerchecked constraints.

ABI boundaries

When crosscompiler binary stability is required, use `extern "C"` and simple data structs (I.26).

Note

Interface design checklist

1. Is the contract clear from the signature? (I.1, I.4)
2. Are ownership, nullability, and bounds explicit? (I.11I.13)
3. Are parameters hard to mix up and reasonably few? (I.23, I.24)
4. Are pre/postconditions stated and enforceable? (I.5I.8)
5. Are failures explicit and hard to ignore? (I.10, `[[nodiscard]]`)
6. For templates, are requirements expressed as concepts? (I.9)
7. If ABI portability is required, is the boundary Cstyle? (I.26)

CHAPTER 10

Separation of Interface and Implementation

Separating **what** is promised (interface) from **how** it is achieved (implementation) improves build times, maintainability, and ABI stability.

Core techniques

Header / source files place declarations in `.h`, definitions in `.cpp`. Headers must include what they use; avoid relying on transitive includes.

Abstract interfaces use a pure abstract class (no data members) to define an interface. Implementations derive from it. Provides full runtime substitution at the cost of virtual dispatch. Aligns with I.25.

PImpl (Pointer to Implementation) move private data and implementation details into a separate `Impl` struct, accessed through an opaque `unique_ptr`. The header stays stable, recompilation is minimised, and object layout can be hidden. Cost: one indirection and a heap allocation per object.

Modules (C++20+) use `export module` in an interface unit to explicitly declare what clients may use. Implementation details stay in module implementation units, reducing textual inclusion and accidental dependencies.

Templates and separation

Template definitions often must be visible at the point of instantiation, so full header/source separation is not always possible. Mitigations:

- Keep templates thin; delegate complex logic to nontemplate implementation functions.
- Use explicit instantiation when the set of instantiated types is controlled.
- Leverage modules to organise exported declarations without exposing implementation details.

Note

Separation checklist

1. Are private details kept out of public headers?
2. Are ownership and bounds contracts clear in the interface (not just in the implementation)?
3. Do headers include what they use and avoid imposing transitive dependencies on clients?
4. For ABI stability, are layoutsensitive details hidden (PImpl, abstract interface, or Cstyle boundary)?
5. For C++20+, are modules used to make exported interfaces explicit?

CHAPTER 11

Minimizing Dependencies

Dependencies drive compile times, rebuild cascades, and coupling. Control them through disciplined interface design.

Core goals:

- Expose only what clients must know.
- Hide implementation details.
- Make each header selfsufficient.
- Prevent cyclic dependencies.
- Leverage tooling (includegraph analyzers, static analysis).

Technique 1: Forward declarations

If a header only uses a type as a pointer/reference, forwarddeclare it instead of including its full definition. Include the definition only in the `.cpp`.

Technique 2: Include hygiene

- Include what you use; never rely on transitive includes.
- Include your own header first in the `.cpp` to catch missing includes.
- Use standard include guards: `#ifndef/#define/#endif` (or `#pragma once`, but note it is not ISO C++).

Technique 3: Move definitions to `.cpp`

Keep headers free of object definitions and noninline function definitions. Heavy includes belong in the implementation.

Technique 4: PImpl (Pointer to Implementation)

Hide private data and implementationonly includes behind an opaque `unique_ptr<Impl>`. The header stays stable, rebuilds are reduced. Cost: one indirection + heap allocation per object.

Technique 5: Nonowning views

Use `std::string_view` and `std::span` for readonly parameters. They avoid coupling to concrete containers and reduce header dependencies.

Technique 6: Break cyclic dependencies

Extract a common interface (pure abstract class), use dependency inversion, or split a common header. Aligns with I.25.

Technique 7: C++20 Modules

Module interface units export declarations; implementation units keep details hidden. Module builds eliminate textual inclusion and reduce accidental dependencies.

Note

Dependencyminimization checklist

1. Does the public header expose only the necessary types/functions?
2. Can includes be replaced by forward declarations where legal?
3. Are noninline definitions kept out of headers?
4. Does each `.cpp` include its own header first?
5. Are cycles eliminated?
6. Where rebuild cost is high, does PImpl hide private details?
7. If using modules, is the exported surface minimal?

CHAPTER 12

Resourcesafe Interfaces

A resourcesafe interface makes correct resource use easy and misuse hard. The foundation is RAII and explicit ownership.

R.1: Manage resources automatically using RAII. Bind resource acquisition to object construction, release to destruction. Use resourcehandle classes.

I.11: Never transfer ownership by a raw pointer (`T*`) or reference (`T&`). Use `std::unique_ptr` for exclusive ownership, `std::shared_ptr` for shared ownership. Raw pointers/references are for nonowning observation.

I.12: Declare a pointer that must not be null as `not_null`. Use `gsl::not_null<T>` to make the nonnull contract part of the type.

I.13: Do not pass an array as a single pointer. Carry bounds with the pointer using `std::span<T>` (or `gsl::span<T>`).

Interface ownership patterns

- **Borrow (nonowning observe):** `T&` (nonnull) or `T*` (nullable).
- **Borrow with nonnull:** `gsl::not_null<T*>`.

- **Take exclusive ownership:** `std::unique_ptr<T>` by value; return `std::unique_ptr<` to transfer out.
- **Share ownership:** `std::shared_ptr<T>` (use sparingly).
- **Observe shared without extending lifetime:** `std::weak_ptr<T>`.

RAII for nonmemory resources

Locks (`std::lock_guard`, `std::scoped_lock`), files (`std::ifstream`), sockets, etc. The same rule applies: acquire in constructor, release in destructor.

Note

Tooling reinforces the contract

- MSVC C26400: storing `new` result in a raw pointer (violates I.11, R.3).
- MSVC C26429: use `gsl::not_null` instead of `assert` for nonnullness.
- MSVC C26481: use `span<T>` instead of raw pointer arithmetic (I.13).

Note

Resourcesafe interface checklist

1. Is ownership explicit? No ownership transfer via raw pointers.
2. Are raw pointers used only as nonowning observers?
3. Is nullability explicit (if null is invalid, use `not_null`)?
4. Are bounds explicit for contiguous sequences (`span`)?
5. Is RAII used for every resource?
6. Are static analysis checks enabled to prevent drift?

CHAPTER 13

Interfaces That Express Ownership and Lifetime

Ownership and lifetime bugs are among the most expensive in C++. The Core Guidelines demand that contracts about who owns what, and for how long, be visible at the interface.

Core vocabulary

- **Ownership:** responsibility to release a resource. Use RAII handles.
- **Lifetime:** period during which an object is valid. Nonowning pointers/references must not outlive the object.
- **Observation:** raw `T*` and `T&` are **nonowning** by default (R.3, R.4).

I.11: No ownership transfer via raw pointer/reference. Make ownership explicit with `unique_ptr`, `shared_ptr`, or return-by-value.

I.12: `not_null` for nonnull contracts. `gsl::not_null<T>` makes nonnullness a typelevel guarantee, not a comment.

Interface shapes by ownership category

Borrow, nonnull `gsl::not_null<T*>` for required pointers. **Borrow, nullable** `T*` (check for null inside). **Take exclusive ownership** `std::unique_ptr<T>` by

value. **Share ownership** `std::shared_ptr<T>` (only when truly shared). **Weak reference** `std::weak_ptr<T>` to observe without extending lifetime. **Nonowning views** `std::span`, `std::string_view`; the caller must guarantee the viewed data outlives the view.

Legacy annotations

When interfaces cannot be changed, use `gsl::owner<T>` to mark owning raw pointers and `gsl::not_null<T>` for nonnull. These aid tools and readers but do not enforce lifetime safety.

Note

Ownership/lifetime checklist

1. Is every ownership transfer expressed via RAII handle (smart pointer or return-by-value)?
2. Are raw pointers/references used only for borrowing?
3. Is nonnullness encoded in the type where required?
4. Are bounds explicit for contiguous sequences?
5. Is the lifetime contract stated for views and observers?
6. Are ownership and nullness checks enabled in static analysis?

Part III

Functions (F)

CHAPTER 14

F.1 – F.60: Function Guidelines (Compact)

The **Functions** rules cover definition, parameter passing, return values, and local function objects. Numbers F.12F.14 and F.28F.41, F.57F.59 are unassigned in the Jul82025 revision.

Definition and Basic Structure

F.1: Package meaningful operations as carefully named functions. Extract coherent actions; naming clarifies intent and limits variable scope.

F.2: A function should perform a single logical operation. One responsibility per function easier to test, reuse, and maintain.

F.3: Keep functions short and simple. Small functions with straightforward control flow are less errorprone.

F.4: If a function may be evaluated at compile time, declare it `constexpr`. Enables compiletime computation; still callable at run time.

F.5: If very small and timecritical, declare `inline`. Use `inline` primarily for ODR management, rarely as a performance hint.

F.6: If your function must not throw, declare it `noexcept`. Communicates guarantee, aids optimization, and affects exceptionsafety.

F.7: For general use, take `T*` or `T&` arguments rather than smart pointers. Smart pointers in parameters unintentionally imply ownership; use raw pointers/references for nonowning access.

F.8: Prefer pure functions. No side effects, result depends only on inputs easier to reason about and parallelise.

F.9: Unused parameters should be unnamed. Suppress warnings; use `[[maybe_unused]]` when conditionally used.

F.10: If an operation can be reused, give it a name. Named functions (or lambdas) document intent and reduce duplication.

F.11: Use an unnamed lambda for a simple function object used in one place only. Keep oneoff behaviour close to its use.

Parameter Passing

F.15: Prefer simple and conventional ways of passing information. Avoid clever encodings; stick to value, reference, pointer.

F.16: For in parameters, pass cheaply copied types by value, others by `const T&`. Small/trivial types value; large types `const&`.

F.17: For inout parameters, pass by reference to nonconst. Mutation is explicit in the signature.

F.18: For willmovefrom parameters, pass by `T&&` and `std::move` the parameter. Communicates consumption; alternatively pass by value and move.

F.19: For forward parameters, pass by `TP&&` and only `std::forward`. Preserves value category; forward exactly once.

F.20: For out output values, prefer return values to output parameters. Clearer, enables chaining, works with NRVO/move.

F.21: To return multiple out values, prefer returning a struct. Named aggregates over raw `pair`/`tuple` improve readability.

Ownership and Pointer/Lifetime Semantics

F.22: Use `T*` or `owner<T*>` to designate a single object. `T*` for nonowning; `owner<T*>` only as an annotation for legacy owning raw pointers.

F.23: Use a `not_null<T>` to indicate null is not valid. Makes precondition explicit; enables tool enforcement.

F.24: Use a `span<T>` to designate a halfopen sequence. Carries pointer + size, preventing bounds errors.

F.25: Use a `zstring` (or `not_null<zstring>`) for Cstyle strings. Distinguishes zeroterminated buffers from arbitrary `char*`.

F.26: Use `unique_ptr<T>` to transfer ownership where a pointer is needed. Explicit exclusive ownership transfer.

F.27: Use `shared_ptr<T>` to share ownership. Only when shared lifetime is genuinely required.

Return Values

F.42: Return a `T*` to indicate a position (only). A pointer naturally represents a location and can be `nullptr` for not found.

F.43: Never return a pointer/reference to a local object. Prevents dangling references and undefined behaviour.

F.44: Return a `T&` when copy is undesirable and no object isnt needed. Always refers to an existing, valid object.

F.45: Dont return a `T&&`. Rvalue references returned often dangle or mislead.

F.46: `int` is the return type for `main()`. Standardrequired; returns process exit status.

F.47: Return `T&` from assignment operators. Enables chaining and matches convention.

F.48: Dont return `std::move(local)`. Inhibits copy elision; rely on natural move/elision.

F.49: Dont return `const T`. Prevents move semantics; constness belongs to the receiving object.

Other Function Rules

F.50: Use a lambda when a function wont do (capture local variables or write a local function). Lambdas keep local logic close and enable capturing.

F.51: Where there is a choice, prefer default arguments over overloading. Reduces overload sets for small behavioural variations.

F.52: Prefer capturing by reference in lambdas that will be used locally. Avoids copies when the lambda does not outlive the scope.

F.53: Avoid capturing by reference in lambdas that will be used nonlocally. Prevents dangling references and data races; capture by value or shared state.

F.54: When writing a lambda that captures `this`, dont use `[=]` default capture. `[=]` implicitly captures `this` and can cause lifetime bugs; capture `this` explicitly or copy needed members.

F.55: Dont use `va_arg` arguments. Not typesafe; prefer variadic templates or typesafe formatting.

F.56: Avoid unnecessary condition nesting. Prefer early exits and clear structure.

F.60: Prefer `T*` over `T&` when no argument is a valid option. Pointers naturally express nullable; references express a guaranteed valid object.

CHAPTER 15

Function Design Principles

A welldesigned function has a clear contract, a single responsibility, minimal side effects, and explicit ownership.

Single Responsibility One function, one logical operation. Split if it does multiple unrelated things.

Clear and Minimal Interfaces Signature should convey intent: prefer expressive parameter types, group related parameters, use names that describe purpose.

Prefer Return Values to Output Parameters Return results by value (leverage NRVO/move). Output parameters are harder to compose and read.

Minimize Side Effects Prefer pure functions; isolate mutations. Side effects should be documented.

Express Ownership and Lifetime

- `T&` / `const T&` required, nonnull, nonowning.
- `T*` optional, nonowning.
- `std::unique_ptr<T>` transfers exclusive ownership.
- `std::shared_ptr<T>` shared ownership (use sparingly).

Parameter Passing Strategies

- **Small/cheap types** pass by value.
- **Large readonly** pass by `const T&`.
- **Mutate inout** pass by `T&`.
- **Consume (willmove)** pass by `T&&` or by value and move.
- **Perfect forwarding** use `T&&` + `std::forward`.

Safety and Error Handling

- Mark nonthrowing functions `noexcept`.
- Use consistent error reporting: exceptions for failure to perform a required task, `std::expected` / `std::optional` for expected alternatives.
- Prefer `[[nodiscard]]` for mustcheck results.

Lambdas and Local Functions

- Use lambdas for short, immediate, oneoff tasks.
- Capture explicitly: by reference for local use, by value (or shared) when the lambda outlives the scope.
- Avoid `[=]` when `this` is involved; capture `this` explicitly.

Note

Function Design Checklist

1. Does the function have a single, clear responsibility?
2. Is the signature selfdocumenting (ownership, nullability, bounds)?
3. Are parameters passed optimally (value, `const&`, `&`, `&&`)?
4. Are side effects minimal and explicit?
5. Is error handling consistent and difficult to ignore?
6. For lambdas: are captures safe for the intended lifetime?

CHAPTER 16

Parameter Passing

Parameter types define the **contract**: they tell the caller what the function needs, modifies, produces, or consumes.

Core Design Goals

- **Readonly input** value (small/cheap) or `const T&` (large).
- **Mutation** `T&` (inout).
- **Output** prefer return values; avoid output parameters.
- **Ownership transfer** `std::unique_ptr` (exclusive) or `std::shared_ptr` (shared).
- **Optionality** `T*` (nullable) or `std::optional<T>` (value absence).

Core Parameter Passing Rules (F.15–F.21)

F.15: Prefer simple and conventional ways of passing information.
Stick to straightforward patterns; use advanced techniques only with measured justification.

Intent	Preferred Form	Notes
Inputonly, cheap to copy	`T` by value	int, double, small structs
Inputonly, not cheap	`const T&`	avoids copy; accepts rvalues
Inout (modify)	`T&`	signals mutation
Outputonly	return value	selfdocumenting, composes
Multiple outputs	return `struct` or `tuple`	named struct preferred
Optional argument	`T*` (nullable)	`nullptr` means no object

F.16: In parameters: cheap by value; otherwise `const T&`. A few machine words is cheap. Avoid `const&` on small types to prevent indirection.

F.17: Inout parameters `T&`. Mutation is explicit in the signature. Keep inout parameters minimal.

F.18: Willmovefrom `X&&` + `std::move` the parameter. Signature communicates consumption. Alternatively, pass `unique_ptr` by value.

F.19: Forward `TP&&` + `std::forward` (exactly once). Only use forwarding references when the function itself does not interpret the argument.

F.20: Output values prefer return values. Return values are selfdocumenting and compose; modern C++ makes returning large objects efficient.

F.21: Multiple output values return a `struct` (or `tuple`). Named structs are clearer for stable APIs.

Semantic Rules: Ownership, Nullness, Sequences (F.22–F.27)

F.22: `T*` for a single object; `owner<T*>` for legacy owning raw pointers. Modern code: `T*` is nonowning; owning pointers use smart pointers.

F.23: `not_null<T>` when null is invalid. Typelevel nonnull contract, better than `assert`.

F.24: `span<T>` for a halfopen sequence. Replaces pointer+length; carries size, supports rangefor.

F.25: `zstring` or `not_null<zstring>` for Cstrings. Explicit zeroterminated requirement.

F.26: `unique_ptr<T>` to transfer ownership where a pointer is needed. Explicit, cheapest safe owning pointer.

F.27: `shared_ptr<T>` to share ownership (only when truly shared). Prefer `unique_ptr` by default.

F.60: Optional Arguments

F.60: Prefer `T*` over `T&` when no argument is valid. `nullptr` means absent. Consider `std::optional<T>` for optional value (not identity).

Common Modern Patterns

- **Readonly string** `std::string_view` (nonowning; ensure lifetime).

- **Contiguous range** `std::span` (carries size, safer than pointer+length).
- **Smart pointer parameters** avoid when you only need to access the object; use `const T&` or `T*`.

Pitfalls to Avoid

- Output parameters that look like inputs prefer return values.
- Forwarding references for performance without measurement use normal techniques.
- Dangling nonowning handles the referred object must outlive the use.

Note

Parameter Passing Checklist

1. Is each parameter clearly **in**, **inout**, **out**, **consume**, **forward**, or **optional**?
2. Are large inputs passed as `const&` (unless intentionally copied)?
3. Are outputs returned by value (or structured result) when practical?
4. Do ownership types appear only when ownership is part of the API?
5. Are nullable parameters expressed as `T*` (or `optional`), not misused references?
6. For sequences, is `span` used instead of raw pointer+length?

CHAPTER 17

Return Values

Returning by value is the default and safest way to communicate results. Modern C++ eliminates unnecessary copies via copy elision and move semantics.

Design Goals

- **Express meaning:** value, position, reference, ownership.
- **Express lifetime:** guarantee no dangling references.
- **Enable safe composition:** allow chaining.
- **Avoid hidden coupling:** never return owning raw pointers; use smart pointers for ownership transfer.

Return Categories

Return by value (`T`) produces a new result (default choice).

Return `T*` a position (F.42) and `nullptr` for not found; nonowning.

Return `T&` existing object, always valid (F.44). Prefer `const&` for read-only, nonnull.

Return `std::optional<T>` value that may be absent. Use for value semantics; `T*` for optional object identity.

Core Return Value Rules (F.42–F.49)

F.42: Return `T*` to indicate a position (only). Not for ownership transfer; `nullptr` means not found.

F.43: Never return a pointer/reference to a local object. Locals die; returning their address causes dangling.

F.44: Return `T&` when copy is undesirable and no object isnt needed. Use for accessors that always succeed (or throw).

F.45: Dont return `T&&`. Rvalue references returned almost always misrepresent lifetime.

F.46: `int` is the return type for `main()`. Standard requirement.

F.47: Return `T&` from assignment operators. Enables chaining; conventional.

F.48: Dont `return std::move(local)`. Redundant; may inhibit NRVO. Just `return local;`.

F.49: Dont return `const T`. Prevents move semantics; adds no safety. Return nonconst by value.

Practical Patterns

- **Maybe results** `std::optional<T>` for value absence; `T*` for position.
- **Structured results** named `struct` with descriptive members.
- **Error signaling** consistent per layer: exceptions for application code; status codes / `expected` for lowlevel.
- **Views/pointers to temporary data** never; ensure lifetime.
- **Owning raw pointers** never; use `std::unique_ptr` or `std::shared_ptr`.

Note

Return Value Checklist

1. Is the return type clear (value vs position vs existing object)?
2. Lifetime safe (no dangling references/pointers)?
3. No result expressed explicitly (`nullptr`, `std::optional`)?
4. Ownership explicit (`unique_ptr`/`shared_ptr` when ownership transferred)?
5. Avoid `T&&`, `return std::move(local)`, `const T` return?

CHAPTER 18

`noexcept` Usage

`noexcept` is a promise: an exception escaping from a `noexcept` function calls `std::terminate()`. Use it when throwing would be a bug (F.6).

Core Guideline

F.6: If your function must not throw, declare it `noexcept`. Use only when the contract demands nonthrowing; do not use as a casual performance hint.

Where `noexcept` Matters Most

- **Destructors** must not throw, especially during stack unwinding. Provide explicit errorreporting member functions for fallible cleanup.
- **Move operations** if truly nonthrowing, mark `noexcept` to allow optimizations in generic code (e.g., vector reallocation).
- **`swap`** nonthrowing swap is a cornerstone of exceptionsafe code.
- **Lowlevel primitives** simple operations that do not allocate or call throwing code.

Conditional noexcept

Templates should use conditional `noexcept` based on the operations of template parameters:

```
template<class T>
void swap(T& a, T& b) noexcept(
    std::is_nothrow_move_constructible_v<T> &&
    std::is_nothrow_move_assignable_v<T>
);
```

When Not to Use noexcept

- Function can allocate, format strings, perform I/O, or call userprovided callbacks that may throw.
- Template with unknown type operations (unless conditional).
- Using `noexcept` as a microoptimization without a correctness reason. If the promise cannot be kept, the result is `std::terminate()`.

Patterns for noexcept-Friendly Code

- **Split canfail from cannotfail:** provide a throwing function and a `noexcept` wrapper that catches and converts to error code/optional.
- **Keep `noexcept` functions simple:** no allocations, no heavy library calls with exceptionbased failure.

Note

`noexcept` Checklist

1. Is throwing considered a bug for this function?
2. Do all callees on all paths satisfy the nonthrowing requirement?
3. Are templatedependent operations protected with conditional `noexcept`?
4. Is the terminationonviolation policy acceptable?
5. For cleanup code, is error reporting handled explicitly?

CHAPTER 19

Small vs. Large Functions

Function size is not about line countits about clarity, single responsibility, and testability. The Core Guidelines anchor this in three rules.

F.2: A function should perform a single logical operation.
One responsibility per function. Split parsing from validation, computation from I/O. Singlepurpose functions are easier to name, test, and reuse.

F.3: Keep functions short and simple.
Limit nesting, use early returns, keep variable scopes minimal, and extract repeated logic into helpers.

F.5: If a function is very small and timecritical, declare it `inline`.
Only for tiny, hotpath utilities after measurement; do not inline large functions for speed.

Why small functions usually win

- **Readability** names become documentation; call sites express intent.
- **Testability** fewer branches, clearer contracts, easier mocking.
- **Maintainability** localized changes, safer refactoring.

When larger functions may be justified

- **Proven performance** profile first; splitting may hinder optimization only if measured.
- **Cohesive algorithm** one logical operation that would lose clarity if split (e.g., a parser).
- **Encapsulation** splitting could expose internal intermediate states, weakening invariants.

Extracting helpers without losing clarity

- Extract decision predicates: `is_valid()`, `has_permission()`.
- Split by conceptual phases: validate transform compute output.
- Do not split arbitrarily just to reduce line count.

Common largefunction failure modes

- Too many responsibilities (validation + I/O + business logic).
- Deep nesting and long variable lifetimes.
- Hidden side effects on globals or shared state.

Note

Checklist: choosing small vs. large

1. Does the function do one logical thing? (F.2)
2. Can nesting be reduced? (F.3)
3. Are key decisions extracted into wellnamed predicates?
4. Are variables declared as late as possible?
5. Is `inline` used only for tiny, hot functions? (F.5)

CHAPTER 20

Const Correctness

`const` turns a promise (this does not mutate) into a compiletime contract. The Core Guidelines recommend immutability by default (Con.1Con.3) and use `const` to distinguish input from inout parameters (F.16F.17).

Con.2: By default, make member functions `const`.
Mark a member function `const` unless it changes observable state. This makes interfaces selfdocumenting and enables use on `const` objects.

Con.3: By default, pass pointers and references to `const`.
Inputonly parameters should not be modifiable. Cheap types by value; large types `const T&`. For pointers, `const T*` when the pointedto object is not modified.

F.16F.17:
- **In parameters:** cheap types by value, others by `const T&`.
- **Inout parameters:** non`const` reference (`T&`) mutation is explicit.

Const overloads for accessors

Provide both `const` (returns `const T&`) and non`const` (returns `T&`) overloads when exposing internal references. This preserves constness of the object.

Logical constness and `mutable`

Use `mutable` only for caches or bookkeeping that do not change the objects observable value. The externally visible behaviour remains `const`.

Return value constness

- **F.49:** Never return `const T` by value it inhibits moves and adds no safety.
- Return `const T&` from `const` member functions; `T&` from non`const` when mutation is allowed.

Common pitfalls

- Forgetting `const` on readonly member functions blocks `const` objects.
- Using `const_cast` to remove constness usually a design error; true const objects mutating is undefined behaviour.
- Passing non`const` references for inputonly parameters implies mutation incorrectly.

Note

Const Correctness Checklist

1. Are inputonly parameters `const` (by value or `const&`)? (Con.3, F.16)
2. Are inout parameters non`const` references? (F.17)
3. Are member functions that do not mutate state marked `const`? (Con.2)
4. Do accessors offer `const` and non`const` overloads where needed?
5. Are return types correct (`const T&` for readonly, avoid `const T`)? (F.49)

CHAPTER 21

Overloads

Overloads should all represent the same conceptual operation. The Core Guidelines (F.51) prefer default arguments over overloads for simple missingparameter cases.

F.51: Where there is a choice, prefer default arguments over overloading.
When overloads differ only by trailing optional parameters, use a single function with defaults. This avoids duplicate implementations and drift.

One name, one concept
All overloads sharing a name must do the same logical thing. Type differences are acceptable; semantic differences are not.

Safe overload patterns

- **Const/nonconst accessors** e.g., `T& operator[]` and `const T& operator[] const`.
- **Refqualifier overloads** different semantics for lvalue vs. rvalue (e.g., return by value for rvalues to enable moves).

Avoid almost the same overloads

Overload resolution can pick unexpectedly due to integral promotions, signed/unsigned, `bool` vs. integer. Prefer distinct names when meanings differ; use `nullptr`

for null pointer overloads, not `0`.

Default arguments and virtual functions

Never put default arguments on virtual functions in public interfaces they bind to the static type, not the dynamic type, causing silent wrong behaviour. Use overloads or nonvirtual wrappers instead.

Overloading vs. templates

- Do not use a forwarding reference `T&&` as a greedy catchall; it can steal calls from normal overloads.
- If templates participate in overload sets, constrain them with concepts so they only match intended types.

Note

Overload Design Checklist

1. Do all overloads express the same concept?
2. If overloads only differ by trailing optional parameters, should they be default arguments? (F.51)
3. Are ambiguous conversions prevented (use `explicit`, avoid `bool` vs. int traps)?
4. Are default arguments avoided on virtual functions?
5. Are templates constrained when part of an overload set?

CHAPTER 22

Lambdas

Lambdas create local, unnamed function objects. The Core Guidelines (F.50F.54) govern their safe use, especially capture by reference/value and lifetime.

F.50: Use a lambda when a function wont do (capture local variables, or write a local function).
Lambdas keep customization close to use and capture local context without global state.

F.52: Prefer capturing by reference in lambdas that will be used locally.
For immediate use (e.g., algorithm call), reference capture is efficient and avoids copies.

F.53: Avoid capturing by reference in lambdas that will be used nonlocally.
If a lambda outlives the scope (stored, returned, or asynchronous), capture by value. Reference captures will dangle.

F.54: When capturing `this`, do not use `[=]` default capture.
Default capture can silently capture `this` and hide lifetime risks. Capture explicitly: `[this]` or `[self = *this]` for safe copy.

Capture semantics

- **By value** safe for escaping lambdas, snapshots state at capture time.
- **By reference** efficient, but only valid if all referenced objects outlive every invocation.
- **`this`** stores a pointer; if lambda escapes, ensure object lifetime, or capture a copy (`[self = *this]`).

Other lambda patterns

- **Generic lambdas** use `auto` parameters when typegenericity is needed; otherwise prefer explicit types.
- **Mutable lambdas** allow modifying capturedbyvalue copies; use sparingly.
- **Avoid `std::function` unless necessary** prefer `auto` or templates for compiletime known callables to avoid type erasure overhead.

Note

Lambda Checklist

1. Is a lambda the right tool (local helper, capture needed)? (F.50)
2. If localonly, are references captured safely? (F.52)
3. If escaping, are captures by value (or safe copies) to prevent dangling? (F.53)
4. Are captures explicit, especially with `this`? (F.54)
5. Would a named function or named lambda variable improve clarity?
6. Is `std::function` used only when runtime type erasure is required?

Part IV

Classes (C)

CHAPTER 23

Class Guidelines (Condensed)

The Classes rules (C.1–C.183) cover how to design types: encapsulate invariants, manage resources safely, and use inheritance only when needed. Below are the most impactful rules, grouped by category.

General Class Rules (C.1–C.9)

C.2: Use `class` if an invariant exists; `struct` if data can vary independently.
C.6: Make members `private` by default.
C.9: Minimize exposure; expose operations, not data.

- **C.1** Group related data into a `struct`/`class`.
- **C.4** Make a function a member only if it needs direct representation access.
- **C.5** Place helper functions in the same namespace.

Concrete Types (C.10–C.49)

C.10: Prefer concrete types for simple value-like concepts.
C.12: Keep data members private unless true aggregate.
C.20: Operations must preserve the class invariant.

- **C.23: Rule of zero** if members handle resources, let the compiler generate special members.
- **C.24: Rule of five** if you write one of destructor/copy/move, define or delete all.
- **C.33: Provide `swap`** when efficient and meaningful.
- **C.38: Prefer composition over inheritance** for code reuse.
- **C.46: RAII** manage every resource in a class that releases it in the destructor.
- **C.48: Avoid `new/delete`** in user code; use resource-managing types.

Special Members (C.50–C.99)

C.61: Follow rule of zero; otherwise rule of five when owning resources.
C.63: Make move operations `noexcept` when they cant throw.
C.66: Prefer destructors to release resources; avoid manual cleanup calls.

- **C.59** Never call virtual functions from constructors/destructors.
- **C.85** Polymorphic base classes must have a virtual destructor.
- **C.87** Delete base-class copy operations to prevent slicing.
- **C.89: Use `override`** on all overriding functions.
- **C.90: `final`** only where further derivation is prohibited.

Class Hierarchies (C.120–C.149)

C.120: Use hierarchies only for runtime polymorphism.
C.121: Prefer pure abstract classes as interfaces.
C.124: Virtual destructor in polymorphic base.
C.129: Pass polymorphic objects by pointer/reference, never by value (avoid slicing).
C.135/136: Multiple inheritance only for distinct interfaces; prefer composition.

- **C.130** Disable copy for polymorphic bases; provide virtual `clone()` if needed.
- **C.143** Never call virtual functions from constructors/destructors.
- **C.144** Avoid downcasting; design interfaces to eliminate it.

Overloaded Operators (C.160–C.179)

C.160: Define operators only when they match conventional meaning.
C.161: Use non-member operators for symmetric operations.
C.166: Prefer named functions when operator meaning might be ambiguous.

- **C.163: Provide `operator==`** only when equality is well-defined.
- **C.168: `explicit` conversion operators** unless implicit conversion is essential.

Unions (C.180–C.183)

C.181: Avoid naked unions; track the active member (tagged union).
C.183: Do not use unions for type punning.

Prefer `std::variant` for safe tagged unions.

CHAPTER 24

Class Design

Class design is about expressing meaning, enforcing invariants, and making ownership explicit. Choose the right kind of type before writing any code.

Three type families

- **Concrete / value type** copyable, movable, regular (e.g., `Percentage`, `User`).
- **Resource handle (RAII)** owns a resource; usually moveonly (e.g., `Buffer`, `File`).
- **Polymorphic hierarchy** interface base + derived; uses virtual dispatch (e.g., `Shape`).

Encapsulation and invariants

- Private data members; expose operations that preserve the invariant (C.2, C.6, C.9).
- Constructors establish invariants; public mutators preserve them.
- Never expose writable baseclass data; use protected functions for extension points.

Interface vs. implementation

- Use Pimpl for ABI stability or heavy implementation hiding.
- Prefer nonmember helper functions in the same namespace (C.4, C.5).

Special members: rule of zero / five

- If all members manage themselves, rely on compilergenerated special members (rule of zero).
- If you own a resource, design copy/move/destructor explicitly (rule of five).
- Move operations should be `noexcept` when possible; destroy must not throw.

Const correctness

- Mark readonly member functions `const`.
- Provide `const` and non`const` overloads when returning references to internals.

Polymorphic base checklist

- Virtual destructor (or protected nonvirtual for nondeletable bases).
- No public data.
- `override` on all overriding functions.
- Delete copy operations; provide virtual `clone()` if copying polymorphic objects is needed.
- Pass and return by pointer/reference, never by value (avoid slicing).

Unions and alternatives

Avoid naked unions; use `std::variant` for a typesafe tagged union. Never typepun with unions.

Note

Class Design Checklist

1. Is the class invariant clear and enforced by constructors?
2. Are data members private, with operations exposing meaning?
3. Are special members correct (rule of zero or five)?
4. Is ownership explicit (no raw owning pointers)?
5. Are const correctness and accessor overloads correct?
6. For polymorphic types: is destruction safe, slicing avoided, overriding explicit?

CHAPTER 25

Invariants

An invariant is a condition that always holds for an object when it is observable. Every constructor must establish it; every public function must preserve it.

Defining invariants:

- Write the invariant explicitly: `0 <= value <= 100`, `size <= capacity`, `handle valid or invalid`.
- Prefer making invalid states unrepresentable (private data + validated construction).

Constructors and factories:

- Constructors must establish the invariant or throw / return an error.
- Use factories for complex validation only return valid objects.

Preserving invariants in member functions:

- On normal exit, invariant holds.
- If an exception is thrown, the object must remain valid (basic guarantee).
- Keep mutators small and checked; refactor to commit changes only after success.

Debugging and checking

- Use assertions (`assert`) in constructors and at end of mutators (debug builds).
- Consider an RAII guard that checks the invariant on construction and destruction, ensuring it holds even during stack unwinding.

Invariants and class hierarchies:

- Baseclass invariants must be minimal and documented.
- Never expose representation through `protected` data.
- Do not call virtual functions from constructors/destructors it breaks invariants.

Ownership as invariant:

- A resourceowning class guarantees release exactly once.
- After a move, the movedfrom object must remain in a valid (but often unspecified) state.

Note

Invariant Checklist

1. Is the invariant clearly stated?
2. Do constructors/factories either establish the invariant or fail?
3. Are data members private so users cannot violate the invariant?
4. Do all public mutators preserve the invariant (on success and on failure)?
5. Are constcorrectness and ownership rules enforced?
6. In hierarchies, is the base invariant minimal and independent of derived details?

CHAPTER 26

Constructors

A constructor must establish the class invariantthe object becomes valid and usable, or construction fails (no partially valid state). Prefer initialization over assignment, avoid implicit conversions, and never call virtual functions from constructors.

Core rules

- **Establish invariants**: validate and set all members in the member initializer list; throw if the invariant cannot be met.
- **Avoid twophase initialization**: no `init()` after default construction.
- **Use `explicit`** for singleargument constructors unless implicit conversion is intended.
- **Member initializer list order** must match declaration order.
- **Delegating constructors** reduce duplication; **inheriting constructors** (`using Base::Base`) when the derived class adds no new initialization.

Rule of zero / five interaction If members are RAII types (smart pointers, containers), the compilergenerated special members are correct (rule of zero). If the class manages a raw resource, design copy, move, and destructor explicitly.

Exception safety If a constructor throws, alreadyconstructed subobjects are automatically destroyed. RAII members make this safe.

Example

```
class Percentage {
public:
    explicit Percentage(int v) : v_(v) {
      if (v_ < 0 || v_ > 100) throw std::out_of_range("Percentage");
    }
    int value() const noexcept { return v_; }
private:
    int v_; // invariant: 0..100
};
```

Factories for complex initialization If construction requires nontrivial validation or polymorphic behaviour, use a static factory that returns a fully constructed object (or `std::optional` / throws).

Note

Constructor checklist

1. Does every constructor establish the invariant or fail?
2. Are all members initialized in the initializer list (not assigned later)?
3. Is the order in the initializer list the same as declaration order?
4. Are singleargument constructors `explicit` unless conversion is intended?
5. Is a default constructor provided only when it yields a meaningful valid state?
6. Are virtual calls avoided in constructors/destructors?
7. Are special members correctly defined or left to the compiler (rule of zero/-five)?

CHAPTER 27

Destructors

A destructor releases resources and must never throw. It is the cornerstone of RAII. Prefer the rule of zero: if all members manage their own resources, do not write a destructor at all.

Core rules

- **Never throw** from a destructor. Throwing during stack unwinding calls `std::terminate()`.
- **RAII members** make custom destructors unnecessary (rule of zero). Use `std::unique_ptr`, containers, etc.
- **If you must manage a raw resource**, release it in the destructor; keep the destructor simple and nonthrowing.
- **Explicit cleanup functions** (`close()`, `commit()`) for operations that may fail; destructor does besteffort nonthrowing cleanup.

Polymorphic base classes A base class with virtual functions must have a **public virtual destructor** (or protected nonvirtual if deletion through base pointer is prohibited). Deleting through a nonvirtual base destructor is undefined behaviour.

Destructor and special member generation Declaring a destructor can suppress implicit move operations. If you write a destructor, reevaluate copy/-move constructors and assignment (rule of five mindset).

Example

```
class FileWriter {
public:
    explicit FileWriter(const char* path) {
        f_ = std::fopen(path, "wb");
        if (!f_) throw std::runtime_error("open failed");
    }
    void close() { /* may throw */ if (f_) {
        if(std::fclose(f_)!=0) throw ...;
        f_=nullptr;
        }
    }
    ~FileWriter() noexcept {
    if (f_) std::fclose(f_);
    } // never throws
private:
    std::FILE* f_{nullptr};
};
```

Note

Destructor checklist

1. Does the class follow the rule of zero (no destructor needed)?
2. If a destructor exists, is it guaranteed not to throw?
3. Does it release resources exactly once and tolerate movedfrom state?
4. For polymorphic bases: is the destructor virtual (or protected nonvirtual)?
5. Are failing cleanup operations provided as explicit functions, not solely in the destructor?
6. Have copy/move operations been reviewed after adding a destructor?

CHAPTER 28

Rule of 0 / 3 / 5

These rules determine which special member functions (destructor, copy/move constructors, copy/move assignment) you must define.

Rule of 0: If your class does not directly manage a resource (all members are RAII types), define **none** of the special members. The compilergenerated ones are correct.

Rule of 3 (preC++11, still relevant): If you manually manage a resource and define a destructor, copy constructor, or copy assignment, you likely need all three to avoid doublefree or leaks.

Rule of 5 (C++11+): If you define or delete any of the five special members (destructor, copy constructor, copy assignment, move constructor, move assignment), review all others. For resource owners, either:

- make the type **moveonly** (delete copy, provide move, destructor as needed), or
- provide **deep copy** and also implement efficient move (often `noexcept`).

Modern practice:

- Prefer **rule of 0**. Use `std::unique_ptr`, `std::vector`, etc. to manage resources.
- If you must write a destructor, consider using `=default` for others where correct.
- **Move operations should be `noexcept`** if possible this enables optimizations in standard containers.
- **Destructors must not throw**.

Example

Moveonly resource handle (rule of five, moveonly variant):

```
class Buffer {
public:
    explicit Buffer(std::size_t n)
        : n_(n), p_(std::make_unique<unsigned char[]>(n)) {}
    ~Buffer() = default;
    Buffer(const Buffer&) = delete;
    Buffer& operator=(const Buffer&) = delete;
    Buffer(Buffer&&) noexcept = default;
    Buffer& operator=(Buffer&&) noexcept = default;
private:
    std::size_t n_{0};
    std::unique_ptr<unsigned char[]> p_;
};
```

Note

Rule of 0/3/5 checklist

1. Can the class be built from RAII members only? Rule of 0.
2. If manual resource management is unavoidable, does the class need copy? No moveonly (delete copy). Yes implement deep copy and move (rule of 5).
3. Are invariants preserved after copy and after move (valid movedfrom state)?
4. Are move operations `noexcept` when they cannot throw?
5. Does the destructor never throw?

CHAPTER 29

Copy and Move Semantics

Copy and move define how objects are duplicated or transferred. Design your type as a **value**, **unique owner**, or **shared owner** and make that choice visible in the interface.

Core design decision

- **Value type** copy creates independent equivalent value (e.g., `std::string`). Rule of zero preferred.
- **Unique owner** move-only, transfer of ownership (e.g., `std::unique_ptr`). Copy deleted.
- **Shared owner** explicit sharing via `std::shared_ptr`; use sparingly.

Rule of zero first If all members are RAII types (containers, smart pointers), do not write any special member functions. The compiler-generated ones are correct.

Copy semantics

- Copy constructor creates an independent object with the same value; preserves invariants.
- Copy assignment must handle self-assignment and exception safety (at least basic guarantee: object remains valid).
- Deep copy for owning raw resources; prefer standard RAII types to avoid manual management.

Move semantics

- Move transfers resources; the source must remain valid but can be empty.
- Move operations should be `noexcept` when they cannot throw this enables optimizations in standard containers.
- Unique ownership move-only: delete copy, provide move (often defaulted).

Deleted and defaulted special members Use `=delete` to prohibit copy/-move; `=default` to request compiler generation. Declaring any special member may suppress implicit generation of others review the full set.

Exception safety Copy-and-swap provides strong guarantee for assignment: copy to a temporary, then swap (no-throw). Destructors must never throw.

Example

Move-only buffer:

```
class Buffer {
public:
    explicit Buffer(std::size_t n)
    : n_(n), p_(std::make_unique<unsigned char[]>(n)) {}
    Buffer(const Buffer&) = delete;
    Buffer& operator=(const Buffer&) = delete;
    Buffer(Buffer&&) noexcept = default;
    Buffer& operator=(Buffer&&) noexcept = default;
    std::size_t size() const noexcept { return n_; }
private:
    std::size_t n_{0};
    std::unique_ptr<unsigned char[]> p_;
};
```

Hierarchies: no slicing Polymorphic objects must not be copied by value (slicing). Use pointers/references. If copying is needed, provide a virtual `clone()` returning `std::unique_ptr<Base>`; consider deleting base copy operations.

Note

Copy/Move Checklist

1. Is the types ownership model clear (value, unique, shared)?
2. Can you follow the rule of zero?
3. If you declare any special member, did you decide all five?
4. For copyable types: does copy produce an independent value and preserve invariants?
5. For movable types: does move transfer ownership and leave the source valid?
6. Are move operations `noexcept` when correct?
7. Are destructors nonthrowing?
8. In hierarchies, is slicing prevented?

CHAPTER 30

Inheritance

Use inheritance primarily for runtime polymorphism (substitutability). Prefer composition for code reuse.

Public inheritance = isa If `D` publicly derives from `B`, a `D` object must be usable wherever a `B` is expected. Use only when you need dynamic dispatch.

Safe polymorphic base requirements

- **Virtual destructor** if deletion through `Base*` is possible, it must be virtual (otherwise UB).
- **No public/protected data** keep representation private; use protected functions for extension.
- **Override explicit** every overriding function marked `override`.
- **Final** only where further derivation/overriding is prohibited.

Avoid slicing Polymorphic objects must be passed by pointer/reference, never by value. If copying is required, use virtual `clone()` returning `std::unique_ptr<Base>`. Delete base copy operations to prevent accidental slicing.

Virtual calls during construction/destruction Never call virtual functions from constructors or destructors dispatch does not work as expected and breaks invariants. Use the NonVirtual Interface (NVI) pattern: public nonvirtual function protected virtual hook.

Name hiding A derived function name hides all base overloads with the same name. Use `using Base::f;` to keep overloads visible.

Multiple inheritance Use primarily for combining pure interfaces (abstract classes with no data). Virtual inheritance only for real diamond problems; keep it rare.

Downcasting Avoid downcasting; redesign the interface if frequent. Use `dynamic_cast` only when runtime type check is unavoidable.

Note

Inheritance Checklist

1. Do you truly need runtime polymorphism? (prefer composition/templates/-variant otherwise)
2. Does the base have a virtual destructor (if deletion through base)?
3. No public/protected data in base; representation is private.
4. Overrides marked `override`; final used intentionally.
5. Polymorphic objects passed by pointer/reference (no slicing).
6. Ownership explicit (`std::unique_ptr<Base>` usually).
7. Virtual calls avoided in constructors/destructors.
8. Downcasts rare and justified.

CHAPTER 31

Polymorphism

Runtime polymorphism allows different implementations behind a common interface. Use only when dynamic behavior selection is required.

When to use Need to choose implementation at run time, stable interface for many variants, or heterogeneous collections. Otherwise prefer templates, `std::variant`, or composition.

Core principles

- **Substitutability**: derived must honor base contracts.
- **Small stable base**: pure abstract, minimal, no data exposure.
- **Virtual destructor** mandatory for polymorphic deletion.
- **Override** on all overriding functions; `final` to seal.

Antipatterns

- Slicing never copy/pass by value.
- Calling virtuals in constructors/destructors undefined/disrupted dispatch.
- Exposing base data makes derived fragile.

Safe patterns

- **Abstract base class**: pure virtual interface + virtual destructor.
- **NVI (NonVirtual Interface)**: public nonvirtual method enforces invariants, calls protected virtual hook.
- **Ownership**: store `std::unique_ptr<Base>` in containers.
- **Clone**: virtual `clone()` returning `std::unique_ptr<Base>` instead of copy constructor.

Performance note Virtual dispatch has indirection cost. In hot paths, consider static alternatives. `final` may help devirtualization.

Note

Polymorphism Checklist

1. Is runtime polymorphism truly needed? (vs. templates/variant/composition)
2. Base interface minimal, coherent, stable?
3. Virtual destructor present?
4. No public/protected data in base.
5. All overrides marked `override`?
6. Ownership via `unique_ptr<Base>`?
7. Slicing prevented (reference/pointer, `clone()`)?
8. Virtual functions not called from constructors/destructors?
9. Downcasts rare and justified?

CHAPTER 32

`final` / `override`

These specifiers make class hierarchies correct and maintainable. `override` catches accidental mismatches; `final` expresses design constraints.

`override`: mandatory for overriding Declare every intended override with `override`. Without it, a signature mismatch silently creates a new function. The compiler diagnoses any mismatch when `override` is used.

Example

```
struct Base {
    virtual std::string name() const { return "Base"; }
};
struct Derived : Base {
    std::string name() const override { return "Derived"; }
    // correct
};
```

`final`: prevent further overriding/derivation

- `final` on a **virtual function** no derived class may override it.
- `final` on a **class** no class may inherit from it.

NVI + final Seal public nonvirtual interface methods with `final` so derived classes cannot bypass invariants.

Name hiding and overloads A derived function hides all base overloads with the same name. Use `using Base::f;` to preserve overload sets.

Note

Final/Override Checklist

1. Every overriding function marked `override`?
2. Leaf classes/sealed functions marked `final` where design requires?
3. Public NVI methods `final` to protect invariants?
4. Base overload sets preserved with `using` where needed?
5. Virtual calls avoided in constructors/destructors?

Part V

Enumerations (Enum)

CHAPTER 33

Enum Guidelines (Condensed)

The Core Guidelines for enumerations (Enum.1–Enum.8, with 910 unassigned) establish strong typing, safety, and clarity.

Enum.1: Prefer enumerations over macros.
Enums respect scope and type; macros dont. Use `enum class` (or `constexpr` variables) for named constants.

Enum.2: Use enumerations for related named constants.
Group cohesive values into an enum; enables switch completeness warnings.

Enum.3: Prefer `enum class` over plain `enum`.
Scoped enums prevent implicit int conversions and name leakage; default to `enum class`.

Enum.4: Define operations on enumerations for safe and simple use.
If an enum needs arithmetic, iteration, or bitmask operations, provide explicit functions/operators.
Example: `next_day(Day)` or `operator|` for flags.

Enum.5: Dont use `ALL_CAPS` for enumerators.
Avoid macro collisions; use normal identifiers (e.g., `Color::red`).

Enum.6: Avoid unnamed enumerations.
Unnamed enums for unrelated constants are unsafe. Prefer `constexpr` variables.

Enum.7: Specify underlying type only when necessary.
Needed for forward declarations, ABI/storage constraints. Otherwise omit.

Enum.8: Specify enumerator values only when necessary.
Default consecutive values are simplest. Explicit values for bit flags, protocols, or external numbering.

CHAPTER 34

`enum class` – Scoped and Strongly Typed

Scoped enumerators prevent name collisions and force qualification (`Color::red`).
No implicit conversion to int – accidental mixing with integers/other enums is a compile error.
Default choice for all new enums.

Example

```
enum class Web_color : std::uint32_t { red = 0xFF0000, blue = 0x0000FF };
void paint(std::uint32_t);
// paint(Web_color::blue); // error: no implicit conversion
paint(static_cast<std::uint32_t>(Web_color::blue)); // explicit boundary conversion
```

Conversions

Use `std::to_underlying` (C++23) or a helper based on `std::underlying_type` for explicit integer conversion. Validate when decoding integers to enums (e.g., with `std::optional`).

Flags

Define bitwise operators explicitly for flag enums (Enum.4).

`using enum` (C++20)

Bring enumerator names into local scope where it improves readability; use sparingly.

CHAPTER 35

Avoiding Unsafe Enums

Unsafe patterns

- Unscoped enums: implicit int conversion, name leakage.
- Using enums for unrelated constants (unnamed enums).
- Casting integers directly to enum without validation.
- Relying on implicit bitwise operations.

Replace with safer alternatives

- Use `enum class` by default.
- Group related values into named enums; unrelated constants `constexpr` variables.
- Validate external input: use a `try_decode` function returning `std::optional<Enum>`.
- Define explicit operations for flags (Enum.4).
- Convert to integers only at I/O boundaries, preferably with `std::to_underlying`.

Example

```
enum class Status : std::uint8_t { ok = 0, failed = 1, timeout = 2 };
std::optional<Status> try_status(std::uint8_t raw) {
    switch (raw) {
    case 0: return Status::ok;
    case 1: return Status::failed;
    case 2: return Status::timeout;
    default: return std::nullopt;
    }
}
```

CHAPTER 36

Explicit Conversions for Enums

Enum to integer: use `std::to_underlying` (C++23) or `static_cast` + `underlying_type_t`.
Integer to enum: validate, dont blindly cast. Prefer a mapping/`switch` that returns `std::optional` or an error.
Domain types: mark constructors and conversion operators `explicit` to prevent implicit conversions.

Rules

- Convert only at boundaries (serialization, C APIs).
- Define enum operations (increment, flags) instead of scattering casts.
- Avoid narrowing; check range before casting.

CHAPTER 37

Type Compatibility with Enums

Unscoped enum: too compatible – implicit conversion to int, can call unintended overloads.
Scoped enum: incompatible with integers and other enums by default – prevents accidental mixing.

Compatibility rules

- Different enum types are distinct, even with same underlying type.
- Underlying type does not imply semantic compatibility.
- Comparisons only allowed with same enum type.
- Overload resolution: take enum parameter, not int, to express domain.

Design for safe compatibility

- Use `enum class` for domain separation.
- Convert explicitly at boundaries; never rely on implicit conversions.
- Define encode/decode functions to centralize integer mapping and validation.

Note

Enumerations Checklist

1. Use `enum class` by default.
2. Group related values; use `constexpr` for unrelated constants.
3. Avoid `ALL_CAPS` enumerators.
4. Specify underlying type and explicit values only when necessary.
5. Validate integer-to-enum conversions; use `std::to_underlying` for enum-to-integer.
6. Define explicit operations for flags and iteration.
7. Convert only at I/O boundaries.

Part VI

Resource Management (R)

CHAPTER 38

Resource Management Guidelines (R.1–R.15)

Resource management is built on **RAII**: bind resource lifetime to object lifetime. The Core Guidelines rules (R.1–R.6, R.10–R.15) eliminate leaks, doublerelease, and unsafe interfaces. R.7R.9 and R.16R.17 are unassigned.

R.1: Manage resources automatically using RAII.
Wrap every acquire/release pair in a class whose destructor releases the resource. Prefer standard RAII types: `std::vector`, `std::unique_ptr`, `std::lock_guard`, file streams.

R.2: Use raw pointers to denote individual objects only.
Never use `T*` for arrays; prefer `std::span` for views, `std::vector`/`std::array` for ownership.

R.3: A raw pointer (`T*`) is nonowning.
Treat `T*` as an observer; ownership is expressed via `unique_ptr`, `shared_ptr`, or containers.

R.4: A raw reference (`T&`) is nonowning.
Use `T&` for required objects, `T*` for optional. Never transfer ownership through references.

R.5: Prefer scoped objects; avoid unnecessary heap allocation.
Local variables are simpler, faster, and exceptionsafe. Only allocate when dynamic lifetime or polymorphism requires it.

R.6: Avoid `nonconst` global variables.
They are shared mutable state dangerous, hard to test, and cause initialization order issues. Prefer dependency injection.

R.10: Avoid `malloc()` and `free()`.
They do not run constructors/destructors. Use RAII wrappers with custom deleters if forced to interface with C.

R.11: Avoid calling `new` and `delete` explicitly.
Use `std::make_unique` / `std::make_shared` and containers; they guarantee exception safety.

R.12: Immediately give allocation results to a manager object.
A raw pointer from `new` must be handed to a smart pointer (or container) before any throwing code. Factory functions avoid the problem entirely.

R.13: Perform at most one explicit allocation per expression.
Multiple allocations in one expression can leak due to evaluation order subtleties. Split them into separate statements.

R.14: Avoid `[]` parameters; prefer `span`.
Array parameters decay to pointers and lose size. `std::span` carries bounds and supports rangefor.

R.15: Always overload matched allocation/deallocation pairs.
If you provide `operator new`, also provide the matching `operator delete` (and likewise for arrays). Mismatch is undefined behaviour.

Example

RAII in action:

```
class Port {
    PortHandle h_;
public:
    Port() : h_{open_port()} {}
    ~Port() { close_port(h_); }
    Port(const Port&) = delete;
    Port& operator=(const Port&) = delete;
};

void send_raii(std::unique_ptr<int[]> data, int n) {
    Port port;
    std::lock_guard<std::mutex> lock(mtx);
    transmit(port.get(), data.get(), n);
} // automatic cleanup: lock, port, memory
```

Note

Resource Management Checklist

1. Every resource owned by an RAII object (R.1).
2. Raw pointers/references are nonowning (R.3, R.4).
3. No raw `new` / `delete` in application code (R.11).
4. Allocation immediately assigned to an owner (R.12).
5. Arrays passed via `span` or container, never as `T*` alone (R.2, R.14).
6. Custom allocators paired with matching deallocators (R.15).

CHAPTER 39

RAII (Resource Acquisition Is Initialization)

Definition: Bind resource acquisition to object construction, release to destruction. C++ guarantees destructors run, even during exceptions.

Why RAII?

- Leakfree by construction.
- Exception safety: cleanup on all paths.
- Local reasoning: resource lifetime = object lifetime.
- Composability: multiple resources, reverse destruction order.

The RAII contract

- Constructor acquires the resource and establishes a valid state; if it fails (throws), no resource leaks.
- Destructor releases the resource and **must not throw**. Handle cleanup errors by logging, storing status, or providing an explicit `close()` that can fail; the destructor does besteffort nonthrowing cleanup.

Standard RAII owners

- **Memory:** `std::vector`, `std::string`, `std::unique_ptr`, `std::shared_ptr`.

- **Synchronisation:** `std::lock_guard`, `std::unique_lock`.
- **Files:** `std::ifstream`, `std::ofstream`.
- **Custom handles:** template wrapper with moveonly semantics, custom deleter.

Example

Generic moveonly handle:

```
template<class Handle, class Invalid, class Close>
class unique_handle {
    Handle h_ = Invalid::value();
public:
    explicit unique_handle(Handle h) noexcept : h_(h) {}
    ~unique_handle() noexcept { reset(); }
    unique_handle(unique_handle&& o) noexcept : h_(o.release()) {}
    unique_handle& operator=(unique_handle&& o) noexcept { reset(o.release()); return *this; }
    // no copy
    Handle get() const noexcept { return h_; }
    void reset(Handle nh = Invalid::value()) noexcept { if(h_!=Invalid::value()) Close{}(h_);
    ↪  h_=nh; }
    Handle release() noexcept { Handle t=h_; h_=Invalid::value(); return t; }
};
```

Move semantics and RAII

Uniqueownership RAII types are moveonly. Standard containers use move to transfer ownership efficiently. Follow the **rule of zero** when possible; if you manage a raw resource, implement the **rule of five** (destructor, copy, move).

Views and observers: completing RAII

Functions that dont own a resource should use safe views: `std::span<T>` for contiguous sequences, `T&` / `T*` for objects. Never return views to local objects.

Note

RAII Design Checklist

1. Does the type clearly own a resource? (constructor acquires, destructor releases)
2. Is the destructor nonthrowing?
3. Copy deleted if copying would duplicate ownership.
4. Move operations provided (or defaulted) for ownership transfer.
5. Preferred: use standard RAII members (rule of zero).
6. If a raw resource is wrapped, provide `get()` and minimal `release()`.

CHAPTER 40

Resource Ownership

Ownership = responsibility to release a resource exactly once. Express it with RAII owners; nonowners are observers that never release.

Owners vs Observers

- **Owners** (perform cleanup): `std::unique_ptr`, `std::shared_ptr`, `std::vector`, custom RAII handles.
- **Observers** (no cleanup): `T&`, `T*`, `std::span`, `std::string_view`.

Core Guidelines Rules: R.3 (raw pointer is nonowning), R.4 (raw reference is nonowning). This convention makes ownership explicit and toolcheckable.

Three Ownership Models

1. **Single owner** (preferred default): use `std::unique_ptr`, move transfers ownership.
2. **Shared owner** (only when multiple owners required): `std::shared_ptr`, avoid cycles with `weak_ptr`.
3. **Borrowed (no ownership)**: pass by reference/pointer/span, return value or view only if lifetime is guaranteed.

Expressing Ownership in APIs

- **Borrow object, nonnull:** `const T&` (readonly) or `T&` (mutable).
- **Borrow object, optional:** `T*` (nullable).
- **Take unique ownership:** `std::unique_ptr<T>` by value; caller `std::moves`.
- **Share ownership:** `std::shared_ptr<T>` (rare; use `const&` to avoid unnecessary refcount bumps).
- **Return ownership:** return by value (preferred) or `std::unique_ptr<T>` for polymorphic/necessary heap.
- **Borrow sequence:** `std::span<const T>` or `std::span<T>`.

Class Design and Rule of Zero

Store RAII members; let the compiler generate special members. For rawhandle wrappers, make the type moveonly, delete copy, and provide nonthrowing destructor.

Example

```
class UniqueHandle {
    int h_;
    static constexpr int invalid = -1;
public:
    UniqueHandle() noexcept : h_(invalid) {}
    explicit UniqueHandle(int h) noexcept : h_(h) {}
    ~UniqueHandle() noexcept { reset(); }
    // no copy
    UniqueHandle(UniqueHandle&& o) noexcept : h_(o.release()) {}
    UniqueHandle& operator=(UniqueHandle&& o) noexcept { reset(o.release()); return *this; }
    int get() const noexcept { return h_; }
    void reset(int nh = invalid) noexcept { if(h_!=invalid) close(h_); h_=nh; }
    int release() noexcept { int t=h_; h_=invalid; return t; }
};
```

Pitfalls

- **Never transfer ownership via raw pointer/reference.**
- **No owning raw pointers or references**; use owners.
- **Avoid `shared_ptr` overuse**; cycles leak break with `weak_ptr`.
- **Views must not outlive their data**: `span`, `string_view`, `T*`.
- **Destructors must not throw**.

Note

Ownership Checklist

1. Is every resource owned by an RAII object?
2. Are raw pointers/references used only for observing?
3. Is ownership transfer done via `unique_ptr` or move (not raw pointer)?
4. Is `shared_ptr` used only when truly shared? Are cycles broken?
5. Do views/references have guaranteed lifetimes?
6. Are destructors nonthrowing?

CHAPTER 41

`new` / `delete` in Modern C++

Core Guidelines: Avoid explicit `new` and `delete` (R.11). Use RAII owners and factory functions. If you must allocate, bind immediately to a manager object (R.12) and keep expressions simple (R.13). Never mix `malloc`/`free` with C++ objects (R.10).

What `new` / `delete` Do

- `new T` allocates storage + calls constructor; throws `std::bad_alloc` on failure.
- `delete p` calls destructor + deallocates storage.
- `new T[n]` / `delete[] p` must be paired correctly; mismatch is undefined behavior.
- Deleting through a base pointer requires a virtual destructor.

Why Explicit `new`/`delete` Are Discouraged

Leaks, double deletes, mismatched `delete`/`delete[]`, dangling pointers, exception-safety holes, and ambiguous ownership.

Modern Replacements (Always Prefer)

- `std::vector<T>` for dynamic arrays.
- `std::make_unique<T>(args...)` for singleobject exclusive ownership.

- `std::make_shared<T>(args...)` only when shared ownership is truly required.
- `std::unique_ptr<T[]>` if you absolutely need a raw array (but `vector` is better).

Example

```cpp
// BAD: leak if do_something throws
int* p = new int(7);
do_something();
delete p;

// GOOD: ownership immediate and automatic
auto p2 = std::make_unique<int>(7);
do_something(); // safe
```

Exception Safety (R.12, R.13)

Immediately assign the result of `new` to an owner (R.12). Do not interleave multiple allocations in a single expression (R.13); split them.

When Explicit Allocation May Be Justified

Custom allocators, memory pools, placementnew into preallocated storage, ABI/-CAPI boundaries (wrap with RAII). Even then, never expose raw owning pointers to general code.

Placement `new`

Constructs an object at a given address without allocating memory. Must call destructor explicitly. Use only in lowlevel infrastructure.

Example

```cpp
alignas(int) unsigned char storage[sizeof(int)];
int* p = ::new (storage) int(42);
p->~int(); // manual destruction
```

Overloading Operators

If you provide custom `operator new`, always provide the matching `operator delete` (and likewise for arrays, sized, and aligned forms).

Note

`new/delete` Checklist

1. Avoid explicit `new`/`delete` in application code.
2. Use `std::make_unique` / `std::make_shared` and containers.
3. If you must allocate, immediately store the result in an RAII owner.
4. Never mix `new[]` with `delete`, or `new` with `delete[]`.
5. Base class for polymorphic deletion must have virtual destructor.
6. Custom `operator new` must be paired with matching `operator delete`.

CHAPTER 42

Avoiding Leaks

Leak = losing the ability to release a resource. Applies to memory, file descriptors, sockets, locks, threads, etc. RAII is the universal cure.

Core Principles

- **R.1:** Every resource must be owned by an RAII object whose destructor releases it.
- **R.3/R.4:** Raw pointers/references are nonowning observers.
- **R.10R.12:** Avoid explicit `new`/`delete`; bind acquisition to owner immediately.

Common Leak Causes

Missing cleanup on a code path (early return, exception), unclear ownership (nobody deletes or two delete), mismatched alloc/dealloc, cycles in `shared_ptr`, dangling views.

The RAII Cure

- **Memory:** `std::vector`, `std::string`, `std::unique_ptr`, `std::shared_ptr`.
- **Files/Sockets/Handles:** custom moveonly RAII wrappers with nonthrowing destructor.
- **Locks:** `std::lock_guard`, `std::unique_lock`.
- **Threads:** `std::jthread` (C++20) autojoins; otherwise a joining guard.

Example

```cpp
// Memory leak avoided:
auto p = std::make_unique<int[]>(1000);

// File handle RAII wrapper:
class UniqueFD { /* constructor opens, destructor closes */ };
UniqueFD fd = open_readonly("data.bin");
// no leak  automatically closed

// Lock leak avoided:
std::lock_guard<std::mutex> lock(mtx);

// Thread leak avoided:
std::jthread worker([]{ /* work */ });
```

Ownership in Interfaces

Raw pointers/references are nonowning. Transfer ownership via `std::unique_ptr` or returnbyvalue. Use `std::span` for sequences instead of pointer+size.

Exception Safety

Destructors must not throw. If cleanup can fail, provide an explicit `close()` / `commit()` that reports errors; the destructor does besteffort cleanup. Avoid multiple allocations in a single complex expression.

Shared Ownership Cycles

Use `std::weak_ptr` to break cycles; otherwise `shared_ptr` objects will never be destroyed.

Note

Leak Prevention Checklist

1. Every resource immediately given to an RAII owner.
2. Raw pointers/references used only as observers.
3. No explicit `new`/`delete` in application logic; use factories.
4. Sequences: `vector`/`array` for ownership, `span` for views.
5. Locks scoped; threads autojoined (`jthread`) or guarded.
6. Destructors nonthrowing; explicit error handling for fallible cleanup.
7. `shared_ptr` cycles broken with `weak_ptr`.

CHAPTER 43

Non-Memory Resources

Uniform treatment: The same RAII rules (R.1, R.3, R.4, R.12) apply to all resources files, sockets, locks, threads, DB connections, OS handles.

Designing RAII Wrappers

For any resource not covered by the standard library:

- **Noncopyable** (delete copy) to prevent doublerelease.
- **Movable** (transfer ownership).
- **Destructor never throws** handle cleanup errors inside or via explicit `close()`.
- Provide `get()` for raw access; `release()` only when needed.

Example

POSIX file descriptor:

```
class UniqueFD {
    int fd_ = -1;
public:
    explicit UniqueFD(int fd) noexcept : fd_(fd) {}
    ~UniqueFD() noexcept { if (fd_ != -1) ::close(fd_); }
    UniqueFD(UniqueFD&& o) noexcept : fd_(o.release()) {}
    UniqueFD& operator=(UniqueFD&& o) noexcept {
        if (this != &o) reset(o.release());
        return *this;
    }
    int get() const noexcept { return fd_; }
    int release() noexcept { int t=fd_; fd_=-1; return t; }
    void reset(int fd=-1) noexcept { if(fd_!=-1)::close(fd_); fd_=fd; }
};
```

Common Resources

- **Files/Sockets:** wrap handles as moveonly RAII.
- **Locks:** use `std::lock_guard`, `std::unique_lock`.
- **Threads:** prefer `std::jthread` (autojoin); otherwise a joining guard.
- **Transactions:** scope guard that rolls back unless committed.

API Design

Make ownership explicit: transfer with uniqueownership types, borrow via `const&` / `span`. Never expose raw handles as owning.

Note

NonMemory Resource Checklist

1. All nonmemory resources wrapped in RAII objects.
2. Wrappers are noncopyable, movable, and have nonthrowing destructors.
3. Scoped locks and automatic thread management used consistently.
4. Fallible cleanup provided via explicit functions, not destructor.
5. Raw handles exposed only when nonowning.

Part VII

Pointers and References (P)

CHAPTER 44

Philosophy Applied to Pointers and References

The Core Guidelines philosophy (P.1P.12) shapes safe, clear use of pointers and references.

P.1 Express ideas directly. Use `T&` when null is invalid, `T*` when nullable, `std::span` for sequences, owners for ownership.

P.2 Write ISO Standard C++. Prefer standard types (`unique_ptr`, `span`, `string_view`) over nonportable wrappers. Encapsulate platform handles behind standard RAII.

P.3 Express intent. Every pointer/reference must clearly show: owning or nonowning? nullable or not? one object or a sequence? mutation intended?

P.4 Strive for static type safety. Replace pointer+length with `span`; avoid `void*` and casts that destroy type information.

P.5 Compiletime checking over runtime. Use references for nonnull, `const` for immutability, `std::array/span` with fixed extents where possible.

P.6 Checkable at run time. Use `span` for boundscheckable interfaces; validate nullable pointers at boundaries.

P.7 Catch errors early. Check `nullptr` and index at API entry; never return dangling pointers/views.

P.8 Dont leak. Raw pointers never own; ownership is only via RAII (`unique_ptr`, containers, handles).

P.9 Dont waste time/space. Pass by reference to avoid copies; avoid unnecessary `shared_ptr` (referencecounting cost). Prefer value semantics by default.

P.10 Prefer immutable data. Default to `const T&` / `const T*`. Make mutation explicit.

P.11 Encapsulate messy constructs. Pointer arithmetic, manual lifetime, casts belong inside small abstractions (RAII handle, view, iterator).

P.12 Use tools. Enable compiler warnings, static analysis (Core Guideline checks), and sanitizers for pointer/lifetime bugs.

Note

Checklist

1. Use `T&` for required, `T*` only for optional (nullable).
2. Never use raw pointers as owners.
3. Prefer `span` for sequences, `string_view` for text (lifetime permitting).
4. Default to `const` pointers/references.
5. Encapsulate pointer arithmetic and casts.
6. Validate null/range at boundaries.
7. Prefer single ownership; `shared_ptr` only when necessary.
8. Use tooling continuously.

CHAPTER 45

Raw Pointers (`T*`)

Raw pointers are nonowning observers; they do not manage lifetime. Use them for optional access, lowlevel interop, or as implementation details.

Interface meaning

- **Single object (optional):** `T*` may be `nullptr`.
- **Sequence?** Prefer `std::span`, never pointer+size.
- **Nonnull requirement?** Use `T&` instead.

Ownership: never via raw pointer Transfer ownership only with `std::unique_ptr` or RAII containers. If legacy code returns an owning raw pointer, wrap it immediately in an RAII owner (e.g., `unique_ptr` with custom deleter).

Lifetime safety Never return a pointer/reference to a local. Store views/-pointers only to objects that outlive them. For containers, beware of reallocation invalidating pointers.

Bounds safety Raw pointer arithmetic hides size; use `std::span` for contiguous sequences. At interop boundaries (`T*, size_t`) convert to `span` immediately.

Const correctness

- `const T*` points to readonly object.
- `T* const` pointer itself is immutable.

Default to `const T*` when mutation is not needed.

Polymorphic deletion If deleting through a base pointer, the base destructor must be virtual. In modern code, avoid explicit `delete`; return `std::unique_ptr<Base>`.

Note

Raw Pointer Checklist

1. Treat `T*` as nonowning observer.
2. Check `nullptr` at API boundaries.
3. Prefer `T&` when null is not meaningful.
4. Never transfer ownership via raw pointer.
5. Use `span` for sequences, not pointer+size.
6. Avoid pointer arithmetic in application logic.
7. Ensure pointedto objects outlive pointers.

CHAPTER 46

References

References are aliases; they cannot be null and cannot be reseated. They simplify code and enforce nonnull contracts.

Types and binding

- `T&` lvalue reference, binds to existing object.
- `const T&` binds to both lvalues and rvalues, readonly access.
- `T&&` rvalue reference, enables move semantics and perfect forwarding.

Usage guidance

- **Required parameter, nonnull:** `const T&` (readonly) or `T&` (mutable).
- **Avoid copies:** pass by `const T&` for large objects.
- **Returning existing objects:** return `T&` or `const T&` when the object outlives the call.
- **Never return a reference to a local/temporary.**

References vs. pointers

- Reference: cannot be null, no explicit dereference.
- Pointer: can be null, requires `*` to access, suitable for optionality.
- Both are nonowning; use RAII for ownership.

Move semantics and forwarding Rvalue references (`T&&`) are used in move constructors/assignment and in templates with `std::forward` for perfect forwarding.

Pitfalls

- **Dangling references**: returning reference to local variable (UB).
- **Binding to temporaries**: only `const T&` and `T&&` may bind to rvalues; `T&` cannot.
- **Container invalidation**: references to elements become invalid after reallocation.

Note

Reference Checklist

1. Use `T&` / `const T&` for required, nonoptional access.
2. Prefer `const T&` for large readonly input parameters.
3. Use `T&&` only for move semantics or perfect forwarding.
4. Never return a reference to a local/temporary object.
5. Ensure the referred object lives at least as long as the reference.

CHAPTER 47

Pointers as Observers (Non-owning Use)

Default convention: raw pointers (`T*`) and references (`T&`) are **non-owning observers**. Ownership is expressed only via RAII owners (`unique_ptr`, containers, handles). (R.3, R.4)

Observer vs Owner

- **Observer** refers to an existing object, never releases it: `T*`, `T&`, `std::span`, `std::string_view`.
- **Owner** responsible for releasing the resource exactly once: `std::unique_ptr`, `std::shared_ptr`, `std::vector`, custom RAII handles.

Interface Patterns

- **Required object:** `const T&` (readonly) or `T&` (mutable).
- **Optional object:** `T*` (nullable, check at boundary).
- **Sequence:** `std::span` (size carried with pointer).
- **Text view:** `std::string_view` (lifetime guaranteed by caller).

Ownership Rules

- Never return owning raw pointers; use `unique_ptr` or return by value.
- If legacy returns owning raw pointer, immediately wrap in RAII (e.g., `unique_ptr` with custom deleter).

Lifetime Constraints Observers must not outlive the observed object. Common causes of dangling:

- Returning address of a local.
- Storing a pointer into a container that later reallocates/erases.
- Deleting an object while observers still hold its address.
- Returning `span`/`string_view` referring to temporary storage.

Example

```
// Good: observer via const ref
void render(const Image& img); // non-null, non-owning

// Good: optional observer via nullable pointer
void maybe_log(const char* msg) { if (msg) log(*msg); }

// Good: sequence observer via span
int sum(std::span<const int> s);

// Bad: returning pointer to local
int* bad() { int x=7; return &x; } // dangling
```

Const Correctness Prefer `const T*` / `const T&` unless mutation is intended.

Note

Observer Checklist

1. Treat `T*` and `T&` as nonowning.
2. Use references for required, pointers for optional.
3. Sequences as `span`, never pointer+size.
4. Never transfer ownership via raw pointer.
5. Ensure observed object outlives the observer.
6. Encapsulate pointer arithmetic.

CHAPTER 48

Avoiding Dangling Pointers

Dangling pointer: a pointer/reference to an object that has been destroyed or memory that has been freed. Dereferencing is undefined behaviour.

Common Causes

- Returning pointer/reference to local variable.
- Storing pointer to container element and then reallocating or erasing.
- Deleting/freeing memory while observers still point to it.
- Binding a view (`string_view`, `span`) to a temporary.

Prevention Strategies

- **RAII ownership:** resources tied to object lifetime, automatic cleanup (R.1).
- **Return by value** instead of returning references/pointers to locals.
- **Reobtain pointers** after operations that may invalidate them.
- **Views bound to stable storage only** ensure underlying data outlives the view.
- **Use `std::weak_ptr`** to observe a shared object safely; check `expired()` or lock before use.

Example

```cpp
// Bad: dangling reference
const std::string& bad() {
    std::string s = "tmp";
    return s; // dangling
}

// Good: return by value
std::string good() { return "tmp"; }

// Good: use weak_ptr for safe observation
std::shared_ptr<int> sp = std::make_shared<int>(10);
std::weak_ptr<int> wp = sp;
if (auto locked = wp.lock()) { /* use *locked */ }
sp.reset(); // object destroyed; wp.expired() == true
```

Concurrency and lifetime If multiple threads access shared objects, synchronize destruction to ensure no thread still holds a pointer/reference after the object is destroyed.

Note

Dangling Prevention Checklist

1. Never return pointer/reference to a local.
2. Ensure stored observers dont outlive the object.
3. Prefer owning members when lifetime is complex.
4. Reobtain pointers after invalidating operations.
5. Views must refer to stable storage.
6. Synchronize destruction with access in concurrent code.
7. Use `weak_ptr` for safe observation of shared objects.

CHAPTER 49

Nullability

Nullability = ability to represent no object. Pointers can be null; references cannot. Express nullability explicitly in the type.

Parameter Design

- **Nonnull, required:** `T&` / `const T&`.
- **Nullable, optional:** `T*` (check for `nullptr`).
- **Optional reference semantics:** `std::optional<std::reference_wrapper<T>>`.
- **Optional with ownership:** `std::unique_ptr<T>` or `std::shared_ptr<T>`.

Return Values

- Maybe absent value `std::optional<T>`.
- Maybe absent object identity `T*` (rarely; prefer optional or smart pointer).
- Avoid returning raw nullable pointers when a safer alternative exists.

Class Members

- Nonnull observer reference (ensures nonnull at construction).
- Nullable observer `T*`; initialize to `nullptr`; document lifetime.
- If null is never allowed, prefer a reference or an owning type.

Example

```
void apply(const Config& cfg); // cannot be null
void maybe_apply(const Config* cfg) { if(cfg) apply(*cfg); }

std::optional<Result> find(int key);

// Optional reference wrapper
void show(std::optional<std::reference_wrapper<const Data>> d) {
    if (d) display(d->get());
}
```

Initialization & Checks

- Always initialize pointers: `int* p = nullptr;`.
- Use `if (ptr)` or `if (ptr != nullptr)`.
- Prefer `nullptr` over `0` or `NULL`.
- Document when null is prohibited; enforce with early throw if violated.

Note

Nullability Checklist

1. Use references for required nonnull parameters.
2. Use pointers or `std::optional` for optional.
3. Initialize raw pointers to `nullptr`.
4. Use `nullptr` not 0/NULL.
5. Smart pointers for nullable owning semantics.
6. Avoid returning nullable raw pointers when better types exist.
7. Document nullability contracts.

Part VIII

Expressions and Statements (ES)

CHAPTER 50

ES Guidelines – Quick Reference

The ES rules (ES.1–ES.107) cover declarations, expressions, conversions, control flow, and arithmetic safety. ES.108–ES.114 are unassigned.

Declarations, Initialization, and Scope

ES.20: Always initialize an object.
ES.23: Prefer `{}`-initializer syntax.
ES.25: Declare objects `const` or `constexpr` unless modification is intended.
ES.28: Use lambdas for complex initialization of `const` objects.
ES.11: Use `auto` to avoid redundant type names.
ES.5: Keep scopes small. **ES.6: Declare loop variables in the `for`-statement initializer.**

Names, Macros, and Constants

ES.14: Avoid magic numbers; use named constants.
ES.15: Prefer `constexpr` for compile-time values.
ES.30/31: Dont use macros for constants, functions, or program logic.
ES.32: Use `ALL_CAPS` only for macro names.
ES.33: If macros are unavoidable, give them unique names.

Expressions and Conversions

ES.40: Avoid complicated expressions.
ES.41: Parenthesize when operator precedence is not obvious.
ES.35/36: Avoid implicit narrowing conversions; use `{}` to prevent narrowing.
ES.47: Use `nullptr` instead of `0` or `NULL`.
ES.48: Avoid casts when possible.
ES.49: If a cast is necessary, use a named cast (`static_cast`, etc.).
ES.50: Never cast away `const`.
ES.56: Write `std::move()` only when an explicit move is intended.

Pointer, Lifetime, and Evaluation Order

ES.42: Keep use of pointers simple and straightforward.
ES.43/44: Avoid expressions with undefined order of evaluation; dont depend on argument evaluation order.
ES.57–ES.59: Never return a reference/pointer to a local variable or temporary.
ES.60: Avoid `new` and `delete` outside resourcemanagement code.
ES.61: Match `delete[]` with `new[]`, `delete` with `new`.
ES.65–ES.68: Dont dereference invalid pointers or pasttheend iterators; avoid pointer arithmetic that escapes object bounds.

Control Flow

ES.70: Prefer `switch` over long `if/else if` chains.
ES.71: Prefer rangefor to traditional `for` loops when iterating all elements.
ES.72/73: Prefer `for` when there is a loop variable; `while` otherwise.
ES.74: Declare loop variables in the `for`-initializer.
ES.75: Avoid `do/while` statements.
ES.76: Avoid `goto`.
ES.77: Minimize `break` and `continue`.
ES.78: No implicit fallthrough in `switch`; use `[[fallthrough]]` if intentional.
ES.86: Dont modify the loop control variable inside the loop body.

Arithmetic and Error Handling

ES.95: Avoid signed integer overflow (undefined).
ES.96/97: Avoid unsigned wraparound as arithmetic; dont mix signed and unsigned.
ES.102: Prefer signed types for arithmetic.
ES.105: Dont divide by zero.
ES.88: Dont use exceptions for normal control flow.
ES.89: Never throw from destructors.

CHAPTER 51

Safe Expressions

One idea per expression. Prefer simple, singlepurpose expressions. Separate side effects and value computation.

Example

```
// Bad: multiple side effects, precedence confusion
if (a & 1 == 0 && ++i < n) { ... }

// Good: explicit steps
bool odd = ((a & 1) != 0);
++i;
if (odd && i < n) { ... }
```

Sequencing: Never depend on unspecified evaluation order. Separate side effects from value use. For function arguments, do not assume lefttoright evaluation.

Precedence: When in doubt, parenthesize. Prefer clarity over cleverness.

Conversions: Avoid narrowing; use braced initialization or explicit checked conversion. Never mix signed and unsigned in the same expression.

Pointer validity: Always ensure pointers are nonnull and not dangling before dereferencing. Use references for mandatory objects, `span` for contiguous ranges with size.

Casts: Avoid casts; if unavoidable, use named casts. Never cast away `const`. Prefer `static_cast` over Cstyle casts.

Arithmetic: Signed overflow is undefined; check or use wider types. Division by zero is undefined; guard it.

Side effects: Keep side effects in separate statements. Avoid assignment inside conditions (except in modern `if`withinitializer).

Note

Safe Expression Checklist

1. Does the expression depend on evaluation order? Split.
2. Multiple side effects? Separate into statements.
3. Precedence unclear? Parenthesize.
4. Implicit narrowing? Make explicit, validate.
5. Signed/unsigned mixed? Choose one domain.
6. Dangling/null pointer? Check, use references/span.
7. Cast present? Use named cast, reconsider design.
8. Overflow/division by zero possible? Guard.

CHAPTER 52

Control Flow

Structured, readable flow: Prefer guard clauses (early returns), shallow nesting, and RAII for cleanup. Every exit path must be safe.

Selection:

- Use `switch` for multiway selection on integers/enums (ES.70).
- Never allow implicit fallthrough: mark intentional cases with `[[fallthrough]]` (ES.78).
- Use `default` only for genuine catchall cases; prefer explicit enumerator handling for closed sets (ES.79).
- Use `if`withinitializer (C++17) to limit scope.

Iteration:

- Prefer range`for` for simple element iteration (ES.71).
- Use `for` when an obvious index exists (ES.72); `while` otherwise (ES.73).
- Declare loop variables in the initializer (ES.74).
- Avoid `do/while` (ES.75).
- Never modify the loop control variable inside the loop body (ES.86).

Exits and Jumps:

- Avoid `goto` (ES.76).
- Minimize `break` and `continue`; use only when they clearly improve readability.
- Prefer early returns over deep nesting RAII guarantees cleanup.

Empty statements: Make intentional empty loops obvious (ES.85); add a comment or use `continue`.

Note

ControlFlow Checklist

1. All paths easy to see? (limit nesting, use guard clauses)
2. `switch` cases complete and safe? (no accidental fallthrough)
3. Loops have clear termination and correct bounds (halfopen ranges)
4. Loop variables scoped locally, never modified in body
5. RAII ensures cleanup on all exits; no manual resource release
6. `goto` avoided; `break/continue` used sparingly

Part IX

Expressions and Statements (ES)

CHAPTER 53

ES Quick Reference

The ES rules (ES.1ES.107) cover declarations, expressions, conversions, control flow, and arithmetic safety. ES.108ES.114 are unassigned.

Declarations, Scope, Initialization

- **ES.20: Always initialize.** **ES.23: Prefer `{}`initialization.**
- **ES.25: `const`/`constexpr` by default.**
- **ES.5: Keep scopes small.** **ES.6: Declare loop variables inside `for`.**
- **ES.11: Use `auto` to avoid repetition.**
- **ES.28: Use lambdas for complex `const` initialization.**

Names, Macros, Constants

- **ES.14: No magic numbers; use named constants.**
- **ES.15: `constexpr` for compiletime values.**
- **ES.30/31: Avoid macros for constants, functions, or program logic.**
- **ES.32: `ALL_CAPS` only for macros.** **ES.33: Unique macro names.**

Expressions and Conversions

- **ES.40: Avoid complicated expressions. ES.41: Parenthesize when precedence is unclear.**
- **ES.35/36: No implicit narrowing; use `{}` to reject it.**
- **ES.47: `nullptr` instead of `0`/`NULL`.**
- **ES.48: Avoid casts. ES.49: If you must, use named casts.**
- **ES.50: Never cast away `const`. ES.56: `std::move` only for explicit moves.**

Lifetime, Pointers, Evaluation Order

- **ES.42: Keep pointer usage simple and straightforward.**
- **ES.43/44: No undefined/unspecified evaluation order; separate side effects.**
- **ES.5759: Never return a reference/pointer to a local or temporary.**
- **ES.60: Avoid `new`/`delete` outside resource management.**
- **ES.61: Match `delete[]` with `new[]`, `delete` with `new`.**
- **ES.6568: No dereference of invalid pointers/pasttheend; no pointer arithmetic that escapes object bounds.**

Control Flow

- **ES.70: Prefer `switch` for multiway dispatch.**
- **ES.71: Range`for` for simple iteration.**
- **ES.72/73: `for` when there is a loop variable; `while` otherwise.**
- **ES.74: Declare loop variable in the `for`initializer.**
- **ES.75: Avoid `do`/`while`. ES.76: Avoid `goto`.**
- **ES.77: Minimize `break`/`continue`.**
- **ES.78: No implicit fallthrough; use `[[fallthrough]]`.**
- **ES.86: Do not modify the loop control variable inside the body.**

Arithmetic and Error Handling

- **ES.95: No signed overflow. ES.105: No division by zero.**
- **ES.96/97: Avoid unsigned wraparound as arithmetic; dont mix signed/unsigned.**
- **ES.102: Prefer signed types for arithmetic.**
- **ES.88: Dont use exceptions for normal control flow.**
- **ES.89: Never throw from destructors.**

CHAPTER 54

Safe Expressions

Keep expressions simple and selfexplanatory. One idea per expression; separate side effects from value computation.

Example

```
// Bad: multiple side effects, precedence confusion
if (a & 1 == 0 && ++i < n) { ... }

// Good: explicit steps
bool odd = ((a & 1) != 0);
++i;
if (odd && i < n) { ... }
```

Sequencing: Never depend on unspecified evaluation order. Separate sideeffect operations from their usage.

```
// Bad: f(i++, i) ; g(x=next(), use(x));
// Good:
int old = i; ++i; f(old, i);
auto t = next(); x = t; g(x, use(x));
```

Conversions: No narrowing. Prefer braced initialization. Use explicit, checked conversion when necessary. Avoid signed/unsigned mixing.

Pointer safety: Ensure nonnull and nondangling before dereference. Use references for mandatory objects, `span` for contiguous ranges.

Casts: Avoid casts; if unavoidable, use named casts. Never cast away `const`.

Arithmetic safety: Signed overflow is undefined; guard division by zero.

Assignments and side effects: No assignment inside conditions (except modern `if`withinitializer). Keep side effects visible in statements.

Note

Safe Expression Checklist

1. Unclear evaluation order? split.
2. Multiple side effects? separate statements.
3. Precedence ambiguous? parenthesize.
4. Implicit narrowing? use `{}` or explicit checked conversion.
5. Signed/unsigned mixed? pick one domain.
6. Dangling/null pointer? use references/`span`.
7. Cast present? named cast, reconsider design.
8. Overflow/division by zero? guard.

CHAPTER 55

Control Flow

Keep flow structured and shallow. Use guard clauses (early returns) to reduce nesting. RAII guarantees cleanup on every path.

Selection:

- `switch` for multiway integer/enum dispatch (ES.70).
- No implicit fallthrough: mark intentional cases with `[[fallthrough]]` (ES.78).
- `default` only for genuine catchall; for closed enums handle all enumerators explicitly (ES.79).
- `if`withinitializer (C++17) limits scope.

Iteration:

- Range`for` for simple element traversal (ES.71).
- `for` with an obvious loop variable; `while` otherwise (ES.72/73).
- Declare loop variable in the `for`initializer (ES.74).
- Avoid `do/while` (ES.75).
- Never modify loop control variable inside the body (ES.86).

Jumps: No `goto` (ES.76). Minimize `break`/`continue` (ES.77). Early returns are fine when RAII handles cleanup.

Empty loops must be clearly intentional (ES.85).

Note

ControlFlow Checklist

1. All paths visible? (limit nesting, guard clauses)
2. `switch` cases complete and fallthrough intentional?
3. Loop bounds correct? (halfopen ranges)
4. Loop variables local and unmodified in body?
5. RAII ensures cleanup on every exit?
6. No `goto`; `break`/`continue` used sparingly.

CHAPTER 56

`if` and `switch`

`if`: Use for general conditions, range checks, and small branching. Keep conditions clear and parenthesized when needed. Avoid assignmentincondition mistakes. Use guard clauses (early returns) to flatten logic.

Example

```
// guard clause pattern
if (s.empty()) return false;
if (!looks_valid(s)) return false;
return process(s);
```

`switch`: Use for multiway dispatch on one discrete value, especially enumerations (ES.70). Make all paths explicit.

Example

```
switch (m) {
case Mode::fast:  run_fast();  break;
case Mode::safe:  run_safe();  break;
case Mode::debug: run_debug(); break;
}
```

Fallthrough: Never accidental. Use `[[fallthrough]]` if intentional (ES.78). Group empty cases for shared actions, but ensure every case terminates explicitly.

`default`: Only for unexpected/invalid input (e.g., external data). For closed enums, handle all enumerators explicitly to catch missing cases.

Variable declarations inside `case`: Wrap in `{}` to limit scope and avoid initialization order issues.

Note

`if`/`switch` Checklist

1. `if` conditions free of precedence surprises?
2. Assignment avoided in conditions (use `if`withinitializer if needed)?
3. Multiway dispatch via `switch`? (not long `if/else if` chain)
4. Every `case` ends with `break` or explicit `[[fallthrough]]`?
5. `default` used only when appropriate (not hiding missing enum cases)?
6. Variables in case blocks safely scoped?

CHAPTER 57

Loops

Choose the right loop:

- Range`for` for iterating all elements of a container/range (ES.71).
- `for` when an index is part of the logic (ES.72).
- `while` for sentinel/streaming loops (ES.73).
- Avoid `do/while` unless a postcondition loop is clearly better (ES.75).

Prefer standard algorithms (ES.1, ES.2) when they express intent: `std::find`, `std::accumulate`, `std::any_of`, etc. They eliminate indexing bugs.

Bounds and Index Safety:

- Use halfopen ranges `[0, n)`.
- Avoid unsigned indices for subscripts (ES.107); prefer signed index type to prevent underflow and wraparound.
- Dont mix signed/unsigned in loop conditions (ES.100).

Pointer/Iterator Validity: Never dereference invalid pointers/iterators. Do not hold pointers/references across container operations that may reallocate or erase. Reobtain iterators after mutation.

Loop control clarity:

- Minimize `break`/`continue` (ES.77).
- No `goto` (ES.76).
- Do not modify the loop control variable inside the body (ES.86).
- Empty loops must be clearly intentional (ES.85).

Note

Loop Checklist

1. Could an algorithm replace the loop?
2. Range`for` if iterating a whole container?
3. Bounds clear and halfopen?
4. Signed types used for indices, no unsigned underflow risk?
5. Pointers/iterators valid throughout the loop?
6. Loop variable modified only in header?
7. Exits (`break`/`continue`) minimal and clear?

CHAPTER 58

Implicit Conversions

Implicit conversions can silently change values, overload resolution, and safety. The rule: control them; make risky conversions explicit.

Narrowing (loss of information): Use braced initialization `{}` to reject narrowing. If intentional, use `static_cast` or a checked helper.

Example

```
double d = 3.14;
int a = d;      // allowed, narrows
// int b{d};    // error: narrowing (safer)
int c = static_cast<int>(d); // explicit
```

Signed/Unsigned mixing: Avoid. Choose one domain and convert explicitly. For indices, prefer signed `ptrdiff_t` with bounds checking.

Userdefined conversions:

- Constructors: mark singleargument constructors `explicit` unless implicit conversion is intended.
- Conversion operators: prefer named accessors (`c_str()`) over implicit conversion operators. Use `explicit operator bool()` for pointerlike types.
- Strong enums: `enum class` prevents implicit conversions to integers.

Boolean contexts: Avoid using integers as booleans when it hides intent. Pointers as conditions are idiomatic (null check).

Note

Implicit Conversion Checklist

1. Could this expression narrow? use `{}` or explicit checked conversion.
2. Signed/unsigned mixed? unify domain.
3. Are overloads sensitive to conversions? redesign or make conversions explicit.
4. Are userdefined converting constructors/operators appropriately `explicit`?
5. Are conditions clearly boolean?

CHAPTER 59

Operator Precedence

ES.41: If in doubt about operator precedence, parenthesize. Clarity beats cleverness.

Most common traps:

- `flags & mask == 0` parsed as `flags & (mask == 0)`. Write `(flags & mask) == 0`.
- `1 << n + 1` parsed as `1 << (n + 1)`. Parenthesize shifts with arithmetic.
- Use `&&` / `||` for logical operations, not `&` / `|`.
- Assignment inside conditions (`if (x = y)`) accidental vs. comparison. Prefer separate statement or `if`withinitializer.
- Nested `?:` hard to read; use `if` statements.
- Comma operator has lowest precedence; avoid in general code.

Safe patterns:

- Always parenthesize bitwise+comparison: `(x & mask) != 0`.
- Name intermediate results for complex conditions.
- Separate side effects from expressions to avoid sequencing surprises.

Note

Precedence Checklist

1. Precedence ambiguous? add parentheses.
2. Bitwise operators mixed with comparisons? parenthesize explicitly.
3. Shifts with arithmetic? parenthesize.
4. `&&`/`||` used correctly, not `&`/`|`?
5. `?:` nested? rewrite as `if`.
6. Side effects in expression? split.

Part X

Naming and Layout (NL)

CHAPTER 60

NL Guidelines (Condensed)

The NL rules are suggested defaults; consistency is the real goal. Numbers NL.6, NL.12–14, NL.22–24 are not present in the current revision.

Comments

- **NL.1: Dont say in comments what can be clearly stated in code.**
- **NL.2: State intent in comments.** Tell what the code is meant to do.
- **NL.3: Keep comments crisp.**

Indentation and Layout

- **NL.4: Maintain a consistent indentation style.** (tools preferred)
- **NL.17: Use K&Rderived layout** (Stroustrup style) as default.
- **NL.20: Dont place two statements on the same line.**
- **NL.21: Declare one name (only) per declaration.**

Naming

- **NL.5: Avoid encoding type information in names.** (no Hungarian notation)
- **NL.7: Make name length proportional to scope.**
- **NL.8: Use a consistent naming style.** (underscore_style recommended)
- **NL.9: Use `ALL_CAPS` for macro names only.**
- **NL.10: Prefer `underscore_style` names.**
- **NL.19: Avoid names that are easily misread.** (O0, l1, similar words)
- **NL.11: Make literals readable.** (digit separators, suffixes)

Member Declaration Order and Layout

- **NL.16: Use a conventional class member declaration order.** (public before protected before private; types, ctors/dtors, functions, data)
- **NL.18: Use C++style declarator layout.** (`T& operator[](size_t);`)

Interface Notation

- **NL.25: Dont use `void` as an argument type.** (C++: `void f();`)
- **NL.26: Use conventional `const` notation.** (`const int x = 7;`)

Unassigned: NL.6, NL.12–14, NL.22–24.

CHAPTER 61

Naming Rules

Naming should express meaning, not mechanics. Good names reduce review time and prevent misuse.

Key principles

- **Name intent, not implementation.** Use verbs for actions, predicate forms for booleans, plural for collections.
- **Include units** in names where relevant: `timeout_ms`, `size_bytes`.
- **Avoid abbreviations** unless they are universal (e.g., `id`, `url`).
- **Do not use reserved identifier patterns** (leading underscore followed by uppercase, double underscores).

Overload sets: All overloads with the same name must represent the same conceptual operation. Different semantics require different names.

Note

Naming Checklist

1. Does the name express purpose, not type? (NL.5)
2. Is the name length appropriate for its scope? (NL.7)
3. Is naming style consistent across the project? (NL.8)
4. Are macros the only `ALL_CAPS` identifiers? (NL.9)
5. Are names readable and unambiguous? (NL.19)
6. Are literals readable and magic numbers replaced by named constants? (NL.11)

CHAPTER 62

Consistency

Consistency is the real goal of the NL rules. It reduces cognitive load, speeds review, and prevents bugs.

What to keep consistent

- **Indentation and braces** (NL.4, NL.17) enforce with automated formatter.
- **Naming style** (NL.8, NL.10) pick one, apply projectwide.
- **Literals and units** (NL.11) digit separators, standard suffixes.
- **Ownership and lifetime vocabulary** use `unique_ptr`, `span`, etc. uniformly in interfaces.
- **Errorhandling strategy** choose exceptions or status codes and stick to it across a layer.

How to achieve consistency

- Document a short, practical house style.
- Automate formatting (clangformat, etc.) in CI and editors.
- Use linters for naming patterns and suspicious constructs.
- Isolate thirdparty code behind adapters; do not reformat it.

When to break consistency: established platform conventions, standard library patterns, or domainspecific readability (math). But then stay consistent within that domain.

Note

Consistency Checklist

1. Is naming style uniform across modules?
2. Are indentation and braces toolenforced?
3. Are macros clearly marked (only them in ALL_CAPS)?
4. Are units explicit in names and literals readable?
5. Are interface conventions for ownership and errors consistent?
6. Is thirdparty code isolated behind adapters?

CHAPTER 63

Layout and Formatting

Layout and formatting prevent misreading and reduce review time. Consistency is more important than which style you choose.

Indentation and Braces (NL.4, NL.17)

- Always use consistent indentation. Prefer braces even when optionalthey prevent fragile edits and accidental scope changes.
- Suggested default brace style: K&Rderived (Stroustrup). Opening brace on the same line as the control statement.
- No misleading whitespace: avoid trailing semicolons on loops that create empty bodies.

Spacing (NL.15)

- Use spaces sparingly. Standard style: spaces around binary operators, no space after unary operators, no spaces inside parentheses/brackets, one space after commas.
- Avoid nonstandard token spacing like `#include < vector >`.

Declarations (NL.18, NL.21)

- Declare only one name per declarationprevents pointerdeclarator traps.
- Use conventional declarator layout: `T& operator[](size_t);` not confusing positioning.
- Prefer `using` aliases for complex pointer types.

Statements per Line (NL.20)

- Exactly one statement per line. Multiple statements on a line are easy to overlook.

Class Layout (NL.16)

- Predictable member order: `public` before `protected` before `private`.
- Within each: types, special member functions, other functions, data members.
- Data members initialized in declaration orderkeep this consistent.

Line Wrapping

- Keep lines reasonable for diffs; break at logical boundaries, align wrapped arguments consistently.

Automation

- Use automated formatters to enforce style; reduces subjective debates.
- Do not reformat thirdparty code; wrap behind adapters with your house style.

Note

Layout Checklist

1. Indentation and braces consistent?
2. Spacing conventional and not distracting?
3. Only one statement per line?
4. Declarations one name each; pointer/reference clear?
5. Class members ordered predictably?
6. Lines wrapped consistently?
7. Formatting automated?

CHAPTER 64

Readability

Readability is a correctness and productivity multiplier. Make intent obvious; reduce mental decoding.

Comments (NL.1NL.3)

- Do not restate code in comments; express intent, invariants, preconditions.
- Keep comments short and technical.

Naming for Readability (NL.5, NL.7NL.11, NL.19)

- Names describe meaning, not type.
- Length proportional to scope: descriptive for globals, short for local loops.
- Use `underscore_style` consistently.
- Avoid visually confusing names (e.g., `lI1OO0`) and nearidentical spellings.
- Make literals readable: digit separators, meaningful suffixes, named constants.

Structure and Visual Clarity (NL.4, NL.17, NL.20)

- Consistent indentation and braces make control flow obvious.
- Guard clauses (early returns) reduce nesting; RAII keeps cleanup automatic.
- Break complex expressions into named steps; one idea per line.

Prefer Standard Algorithms and Small Functions

- Use algorithms like `std::any_of`, `std::find` to say *what* rather than *how*.
- Small, wellnamed functions replace long blocks and increase testability.

Note

Readability Checklist

1. Can a new reader understand intent quickly?
2. Comments add intent, not just repeat code?
3. Indentation and bracing unambiguous?
4. One statement per line?
5. Names meaningful and consistent?
6. Literals readable, units visible?
7. Complex expressions decomposed into steps?

CHAPTER 65

Avoiding Ambiguity

Eliminate any situation where a reader can reasonably interpret code in more than one way.

Avoid Ambiguous Naming (NL.5, NL.19)

- No visually similar identifiers; no nearidentical spellings.
- Avoid typeencoded names that become lies after refactoring.
- Use standard abbreviations only (`id`, `url`); no personal abbreviations.

Avoid Ambiguous Structure (NL.4, NL.17, NL.20)

- Consistent braces prevent danglingelse problems.
- No misleading indentation; one statement per line prevents missed operations.
- Use spacing that does not break tokenization.

Avoid Ambiguous Declarations (NL.18, NL.21)

- One name per declaration; avoid `int* p, q;` traps.
- Use `using` for complex pointers.

Expressions (ES.41, conversion rules)

- Parenthesize when mixing bitwise/logical operators and comparisons.
- Avoid signed/unsigned mixing; convert explicitly and validate.
- Make switch fallthrough explicit with `[[fallthrough]]`.

API and Overload Ambiguity

- All overloads of a name must represent the same conceptual operation.
- Avoid overloads that invite surprising implicit conversions; use `enum class` or distinct names.

Note

Ambiguity Checklist

1. Identifiers visually distinct? (NL.19)
2. Naming consistent and meaningbased?
3. Braces and indentation unambiguous?
4. One statement per line?
5. Declarations simple and unambiguous?
6. Complex expressions parenthesized and conversionsafe?
7. Switch fallthrough explicit?
8. APIs resistant to overload ambiguity?

CHAPTER 66

T.1–T.83: Rule Overview

The **T** rules cover templates, concepts, and generic programming. Many numbers are reserved or not yet assigned. The existing rules focus on writing clear, wellconstrained generic interfaces.

Present rules (selection)

- **T.1–T.5, T.10–T.13, T.20–T.26, T.30–T.31, T.40–T.49, T.60–T.69, T.80–T.83** are present.
- **T.64**: Use class template specialization for alternative implementations.
- **T.65**: Use tag dispatch for function behaviour selection (no partial specialisation of functions).
- **T.67**: Specialise for irregular types.
- **T.83**: Do not declare member function templates `virtual`.

Unassigned ranges

- T.6–T.9, T.14–T.19, T.27–T.29, T.32–T.39, T.45, T.50–T.59, T.63, T.66, T.70–T.79 are **not present**.

CHAPTER 67

Template Design

Design templates as contracts. Express requirements with concepts; require only the operations you use.

Concepts (T.10–T.13)

- Specify concepts for all template parameters (T.10). Prefer standard concepts (T.11).
- Define concepts by use patterns, not member names.
- Document semantic axioms (equality, ordering, etc.) in comments.

Interface Design (T.40–T.49)

- Use function objects (callables) instead of flags to pass operations.
- Require only essential properties; avoid overconstraining.
- Use template aliases (`using`) to simplify types.
- Prefer CTAD or factory functions when deduction improves usability, but keep deduction unambiguous.
- Avoid unconstrained templates with common namesthey hijack overloads (T.47).

Customization and ADL (T.69)

- Qualify nonmember calls unless ADL is explicitly intended as a customization point.
- For intentional customisation, provide a default in your namespace and call unqualified after a `using` declaration.

Implementation and Special Member Functions

- Minimise template context dependencies; place nondependent members in nontemplate bases (T.62).
- Prefer `{}` over `()` in templates to avoid vexing parses.
- Keep error messages short by using wellnamed concepts.

Templates and Hierarchies (T.80–T.83)

- Do not naively templatize a class hierarchy; prefer composition or static polymorphism (CRTP) when virtual dispatch is not needed.
- Member function templates cannot be `virtual`; use visitor or type erasure patterns instead.

Note

Template Design Checklist

1. Are template requirements explicit and minimal? (concepts)
2. Are ownership/lifetime and parameter passing clear from types?
3. Is the interface consistent and predictable?
4. Are customization points intentional and documented?
5. Is compiletime cost controlled (no unnecessary parameters/instantiations)?
6. Are initializations safe (`{}`) and conversions explicit?

CHAPTER 68

Specialization

Specialize class templates only when representation or algorithm fundamentally differs. For function behaviour, prefer constrained overloads or tag dispatch (T.65).

Class Template Specialisation (T.64, T.67)

- Use explicit/full specialisation for a specific type with a different representation (e.g., `std::hash` for user types).
- Partial specialisation is allowed for class templates onlyuse it for pattern-based classification.
- Specialisations must preserve the interface contract and be placed in the same namespace as the primary template.

Function Behaviour Selection (alternatives to specialisation)

- **Constrained overloads** (concepts) first choice when the distinction is a requirement.
- **Tag dispatch** explicit selection via a tag type (e.g., `std::true_type`/`std::false_type`).
- **`if constexpr`** for small internal branching in a single function body.

Example

```cpp
// tag dispatch example
template<class T>
void serialize_impl(const T& x, std::true_type) { /* trivially copyable */ }

template<class T>
void serialize_impl(const T& x, std::false_type) { /* general */ }

template<class T>
void serialize(const T& x) {
    serialize_impl(x, std::is_trivially_copyable<T>{});
}
```

Danger Zones

- Do not specialise function templates (partial specialisation is illegal). Use overloads.
- Overlapping partial specialisations cause ambiguity; keep patterns simple and nonoverlapping.
- Specialisations must be visible consistently across translation units (ODR). Place them in headers.
- Specialising standard library templates is allowed only for userdefined types at explicitly permitted points (e.g., `std::hash`) and must follow required semantics.

Note

Specialisation Checklist

1. Is specialisation truly required (different representation/algorithm)?
2. Does it preserve the contract of the primary template?
3. Could traits + `if constexpr` or constrained overloads replace it?
4. Are partial specialisations nonoverlapping?
5. Are specialisations declared in the correct namespace and visible where needed?
6. No illegal modifications of `std`?

CHAPTER 69

SFINAE

SFINAE (Substitution Failure Is Not An Error) enables compiletime selection of template overloads or specializations. If substituting template arguments fails in the *immediate context*, the candidate is discarded rather than causing a hard error.

Where SFINAE applies

- Forming types, return types (including trailing returns), default template arguments.
- Expression wellformedness in those contexts (expression SFINAE).
- *Not* inside function bodies failure there is a hard error.

Classic patterns

- **Detection with `decltype` and overloads**: use the comma operator to test an expression, fallback with ellipsis.
- **`std::void_t`** with partial specialization the canonical detection idiom for traits.
- **`std::enable_if`** enable/disable overloads via a boolean condition; now superseded by concepts.

Example

Detection idiom with `void_t`:

```
template<class, class = void>
struct has_begin_end : std::false_type {};

template<class T>
struct has_begin_end<T, std::void_t<
    decltype(std::declval<T&>().begin()),
    decltype(std::declval<T&>().end())>> : std::true_type {};
```

Guideline guidance (T.48): Use concepts/constraints instead of SFINAE in public interfaces. Keep SFINAE for compatibility or internal traits. Isolate it behind named aliases.

Common pitfalls

- Leaking SFINAE into unreadable signatures prefer named concepts.
- Ambiguous overloads when multiple candidates remain viable.
- Negationheavy SFINAE creating fragile overload sets.
- ADL surprises altering which expressions are wellformed.
- Hard errors mistaken for SFINAE only immediatecontext failures count.

Modern alternatives

- `requires`clauses and concepts (C++20) cleaner, better diagnostics.
- `if constexpr` for internal branching based on traits.
- Tag dispatch for function selection.
- Partial specialization for class templates (where structure differs).

Note

SFINAE Checklist

1. Is SFINAE only used where concepts are unavailable or for compatibility?
2. Are conditions in the immediate context, not inside function bodies?
3. Are overload sets designed to avoid ambiguity?
4. Are public APIs readable (no cryptic `enable_if`)?
5. Are traits/detection idioms preferred over adhoc expression checks?

CHAPTER 70

Constraints

Constraints (C++20) are the modern way to express template requirements. They replace most SFINAE with explicit, readable, and diagnosable contracts.

Forms

- `template< std::integral T >` constrained template parameter.
- `std::integral auto f(...)` abbreviated function template.
- `requires`clause on a template/function.
- `requires`expression to define a concept by use patterns.

Guideline recommendations

- **T.10:** Specify concepts for all template arguments.
- **T.11:** Prefer standard concepts.
- **T.41:** Require only essential properties avoid overconstraining.
- **T.47:** Avoid unconstrained public templates with common names.
- **T.48:** Use SFINAE only as a fallback.

Example

```
template<std::integral I>
I inc(I x) { return x + 1; }

template<class T>
requires std::movable<T> && std::default_initializable<T>
T make_default() { return T{}; }
```

Benefits

- Early rejection: failure at the call site, not deep inside instantiation.
- Better overload resolution: more structured than SFINAE, subsumption rules prefer moreconstrained overloads.
- Selfdocumenting interfaces.

Designing constraints

- Use standard concepts where applicable (`std::integral`, `std::equality_comparable`, etc.).
- Define custom concepts as reusable names for requirements.
- Keep concepts coherent; document semantic axioms (e.g., equivalence relation for equality).
- Avoid negation/disjunctionheavy overload sets they become fragile.
- If a constraint set becomes complex, split into smaller named concepts.

Note

Constraints Checklist

1. Are template parameters constrained with concepts? (T.10)
2. Are standard concepts used when available? (T.11)
3. Are constraints minimal but sufficient? (T.41)
4. Are concept names used, not inline boolean expressions?
5. Are overload sets unambiguous, with a clear best match?

CHAPTER 71

Concepts

Concepts are named compiletime predicates that constrain template parameters. They define a coherent set of requirements (syntactic and semantic) for a type.

Syntax

```
template<class T>
concept HasSize = requires(const T& t) {
    { t.size() } -> std::integral;
};
```

Core Guidelines on Concepts

- **T.20:** Concepts must be meaningful; dont create a concept for random syntax.
- **T.21:** Require a complete set of related operations (e.g., `==` and `!=` for equality).
- **T.22:** Document semantic laws (axioms) C++ cannot enforce them, but code and tests must respect them.
- **T.24:** Use tags or strong types when identical syntax has different semantics.
- **T.26:** Define concepts by use patterns, not by member names.

Standard concepts (use them!)

- Type properties: `std::movable`, `std::copyable`, `std::semiregular`, `std::regular`
- Comparisons: `std::equality_comparable`, `std::totally_ordered`
- Callables: `std::invocable`, `std::predicate`
- Iterators/ranges: `std::input_iterator`, `std::forward_iterator`, `std::ranges::range`
- Arithmetic: `std::integral`, `std::floating_point`

Refinement and composition

- A refined concept adds further constraints: `concept AddableAndOrdered = Addable<T> && std::totally_ordered<T>;`.
- Negation (`!`) and disjunction (`||`) should be used sparingly they can create confusing overload sets.

Overload resolution When multiple overloads are viable, the **more constrained** overload (via subsumption) is preferred. Design overload sets so the best match is unambiguous for every intended call.

Note

Concepts Checklist

1. Is each concept meaningful and coherent? (T.20)
2. Does it require all needed operations? (T.21)
3. Are semantic axioms documented? (T.22)
4. Are semantic differences separated with strong types? (T.24)
5. Is the concept defined by use patterns? (T.26)
6. Are negation/disjunction used only when they improve clarity? (T.30, T.31)

CHAPTER 72

Reducing Template Error Complexity

Goal: Make template failure messages short, local, and actionable.

Core techniques (guided by T.* rules)

- **Constrain interfaces (T.10)** use concepts/requires to reject invalid types at the call site, before instantiation.
- **Name requirements (T.20T.26)** use named concepts instead of long inline boolean expressions.
- **Require only what you need (T.41)** minimal constraints reduce cryptic failures for types that could otherwise work.
- **Avoid unconstrained public templates with common names (T.47)** they participate in too many overloads and produce huge error sets.
- **Prefer concepts over SFINAE (T.48)** concepts yield clearer diagnostics; isolate SFINAE when unavoidable.
- **Use `static_assert` with a clear message** for semantic requirements that concepts cant express.
- **Control ADL (T.69)** qualify nonmember calls unless you intend an explicit customization point.
- **Minimize context dependencies (T.60)** keep includes explicit, avoid includeorder surprises.
- **Reduce type noise** use aliases for long template types.
- **Prefer simple patterns:** `if constexpr`, tag dispatch, partial specialization over deep SFINAE chains.

Example

```
// Instead of an unconstrained print:
template<class T>
void print(const T&); // dangerous

// Use a constrained version:
template<class T>
concept StringLike = requires(const T& t) { std::string_view{t}; };

template<StringLike T>
void print_text(const T& t);  // clear, early rejection
```

Errorreduction patterns

- Place `static_assert` at the top of a template to give a userfriendly message for missing requirements.
- Use `requires`expressions in concepts so the missing operation is named in the error.
- Keep overload sets small: if multiple overloads are needed, ensure exactly one is the best match for every intended type.

Note

Template Error Complexity Checklist

1. Are templates constrained at the interface? (T.10)
2. Are constraints named, minimal, and meaningful? (T.41, T.20)
3. Are overload sets small, constrained, and unambiguous? (T.47)
4. Is SFINAE isolated and used only as fallback? (T.48)
5. Are ADL and unqualified calls controlled? (T.69)
6. Are dependencies explicit, and headers selfcontained? (T.60)
7. Are `static_assert` messages provided for semantic requirements?

Part XI

Error Handling and Exceptions (E)

CHAPTER 73

E.1–E.30: Rules Overview

The **E** rules define how to signal and handle failure while preserving invariants and resources. E.9–E.11, E.20–E.24, and E.29 are not assigned in the current revision.

E.1: Develop an errorhandling strategy early. Decide: exceptions vs. error codes, what is recoverable, how RAII is used, and how invariants are maintained.

E.2: Throw to signal inability to perform the required task. Use exceptions when a function cannot fulfil its contract (constructor fails, resource acquisition fails). Do not use for normal control flow.

E.3: Use exceptions for error handling only. Exceptions are for rare failures, not ordinary outcomes.

E.4: Design errorhandling around invariants. After a failure, every surviving object must remain valid (basic guarantee). RAII and value semantics help.

E.5: Let constructors establish invariants; throw if they cant. No partially valid objects.

E.6: Use RAII to prevent leaks. All resources must be owned by RAII handles; cleanup happens automatically even during stack unwinding.

E.7, E.8: State preconditions and postconditions. Document what must hold before and after a call; test them.

E.12: Use `noexcept` when throwing is impossible or unacceptable. Destructors, swap, and many move operations must not throw.

E.13: Never throw while being the direct owner of a raw resource. Wrap raw resources in RAII immediately to avoid leaks.

E.14, E.15: Use purposedesigned exception types; throw by value, catch by reference. Avoid builtin types for exceptions; catch `const` references to prevent slicing.

E.16: Destructors, deallocation, `swap`, and exception copy/move must never fail. They are fundamental to exception safety.

E.17, E.18: Dont catch every exception in every function; minimize explicit `try/catch`. Rely on RAII for cleanup; catch at architectural boundaries (thread entry, API boundaries).

E.19: Use a `final_action` (scope guard) for cleanup when no RAII handle exists. A small callable that runs on scope exit.

E.25–E.27: When exceptions are disabled: Still use RAII (E.25); consider failfast (E.26); otherwise, use systematic error codes (E.27). Avoid global state like `errno` (E.28).

E.30: Do not use exception specifications. Dynamic exception specifications are removed from the standard; use `noexcept`.

Note

ErrorHandling Checklist

1. Is a consistent strategy documented? (exceptions vs. codes, boundaries)
2. Do functions throw only for contract failure, not normal outcomes?
3. Are invariants preserved on all failure paths?
4. Are all resources managed by RAII or scope guards?
5. Are `noexcept` decisions correct (destructors, swap, moves)?
6. Are exceptions typed, thrown by value, caught by reference?
7. Is `try/catch` concentrated at boundaries, not everywhere?
8. In noexception builds: RAII, systematic error codes or failfast.

CHAPTER 74

When to Use Exceptions

Use exceptions when:

- A function **cannot complete its assigned task** while preserving its contract (E.2).
- The failure is **rare / exceptional** (E.3).
- A constructor **cannot establish an invariant** (E.5).
- Resource acquisition fails, but RAII guarantees cleanup (E.6).
- You need to propagate failure across multiple layers without manual boilerplate.

Do not use exceptions for:

- **Normal, expected outcomes** (e.g., search miss) use `std::optional`, `expected`, error codes.
- **Performancecritical frequent failure paths** exceptions have overhead.
- **In destructors** must never throw (E.16).
- **When the environment forbids them** (kernel, embedded, ABI boundaries) use RAII and systematic error codes (E.25–E.27).
- **Replacing a consistent precondition failure policy** choose failfast or throw, but be consistent.

Practical Patterns

- Throw purposedesigned types derived from `std::exception`; catch by `const&`.
- Let RAII do cleanup; write `try/catch` only at boundaries (E.17, E.18).
- Mark functions `noexcept` when they cannot reasonably throw (destructors, swap, simple accessors) (E.12).

Example

```
// Exception safe constructor + RAII
class File {
    std::FILE* f_ = nullptr;
public:
    explicit File(const char* path, const char* mode) {
        if (!(f_ = std::fopen(path, mode)))
            throw std::runtime_error{"open failed"};
    }
    ~File() noexcept { if (f_) std::fclose(f_); }
    // ...
};

// Catch at boundary
int main() {
    try {
        File f("data.txt", "r");
        process(f);
    } catch (const std::exception& e) {
        log_fatal(e.what());
        return 1;
    }
}
```

Note

ExceptionUse Checklist

1. Is failure truly exceptional and not a common outcome?
2. Does the failure prevent the function from fulfilling its contract?
3. Will exceptions cross forbidden boundaries? (translate at boundary if yes)
4. Are all resources RAIImanaged so throws cannot leak?
5. Can invariants be preserved even if throws propagate?
6. Are `noexcept` guarantees upheld on destructors/moves/swaps?

Part XII

Error Handling and Exceptions (E)

CHAPTER 75

E Rules – Condensed

The **E** rules cover error strategy, exceptions, guarantees, and resource safety. E.9–E.11, E.20–E.24, E.29 are unassigned.

E.1: Design an errorhandling strategy early. Decide: exceptions vs. error codes, recoverable vs. fatal, RAII everywhere.

E.2: Throw to signal inability to perform the required task. Use only when a function cannot fulfill its contract.

E.3: Use exceptions for error handling only. Do not use for normal control flow.

E.4: Design error handling around invariants. After a failure, every surviving object must remain valid.

E.5: Let constructors establish invariants; throw if they cannot. No halfconstructed objects.

E.6: Use RAII to prevent leaks. Resources must be owned by RAII handles; cleanup is automatic on any exit.

E.7/E.8: State preconditions and postconditions. Document what must be true before/after a call.

E.12: Use `noexcept` when throwing is impossible or unacceptable. Apply to destructors, swap, and many moves.

E.13: Never throw while directly owning a raw resource. Immediately wrap raw acquisitions into RAII.

E.14/E.15: Throw userdefined types; throw by value, catch by `const&`. Avoid builtin types and slicing.

E.16: Destructors, deallocation, `swap`, and exception copy/move must never fail. Foundation of exception safety.

E.17/E.18: Dont catch everywhere; minimize `try/catch`. Let RAII handle cleanup; catch only at boundaries.

E.19: Use a `final_action` (scope guard) for custom cleanup. When no RAII handle exists.

E.25–E.27: When exceptions disabled: still use RAII; choose systematic error codes or failfast. Avoid global state like `errno` (E.28).

E.30: Dont use dynamic exception specifications. Use `noexcept`.

Note

ErrorHandling Checklist

1. A documented, consistent strategy exists.
2. Functions throw only for contract failure, not normal outcomes.
3. Invariants preserved on all failure paths; RAII used for all resources.
4. `noexcept` correct on destructors, swap, moves.
5. Exceptions caught by reference at architectural boundaries.
6. In noexception builds: RAII + systematic error codes or failfast.

CHAPTER 76

When to Use Exceptions

Use exceptions when:

- A function **cannot perform its assigned task** (contract failure) (E.2).
- Failure is **rare and exceptional** (E.3).
- A constructor **cannot establish an invariant** (E.5).
- Resource acquisition fails, but RAII ensures cleanup (E.6).
- Clean propagation across multiple layers without boilerplate is needed.

Do not use exceptions:

- For **normal, frequent outcomes** (e.g., search miss) use `std::optional`, error codes.
- In **performancecritical** loops where failure is common.
- In **destructors** must never throw (E.16).
- When the environment **disables exceptions** use alternatives (E.25–E.27).
- To replace a consistent preconditionfailure policy; choose failfast or throw and stick to it.

Exceptionsafe patterns:

- Throw types derived from `std::exception`; catch by `const&`.
- Rely on RAII for cleanup; write `try/catch` only at boundaries (E.17/E.18).
- Mark `noexcept` where throwing is impossible (destructors, swap, trivial accessors) (E.12).

Example

```
class File {
    std::FILE* f_ = nullptr;
public:
    explicit File(const char* path, const char* mode) {
        if (!(f_ = std::fopen(path, mode))) throw std::runtime_error{"open failed"};
    }
    ~File() noexcept { if (f_) std::fclose(f_); }
    // ...
};

int main() {
    try {
        File f{"data.txt", "r"};
        process(f);
    } catch (const std::exception& e) { log_fatal(e.what()); return 1; }
}
```

Note

Exception Use Checklist

1. Is failure truly exceptional, not a common outcome?
2. Does failure violate the functions contract?
3. Will exceptions cross forbidden boundaries? Translate at boundary if needed.
4. Are all resources RAIImanaged so throws cannot leak?
5. Are invariants preserved on all failure paths?
6. Are `noexcept` promises upheld on destructors/moves/swaps?

CHAPTER 77

Exception Guarantees

Guarantees describe what remains valid after an exception. The four levels:

1. Nothrow guarantee (`noexcept`) – Operation never throws. Essential for destructors, swap, moves.
2. Strong guarantee (commitorrollback) – No observable state change if an exception occurs.
3. Basic guarantee – Invariants preserved, no leaks, but state may have changed. Minimum acceptable.
4. No guarantee – Unacceptable for most code.

Achieving Strong / Basic Guarantees

- **Copyandswap:** Copy state, modify copy, swap (nothrow) – strong guarantee.
- **Buildthencommit:** Construct new state in temporaries, then commit with nothrow steps.
- **RAII for resources:** Prevents leaks; enables basic guarantee automatically.
- **Nothrow swaps and moves:** Enable efficient commit steps.
- **Maintain invariants:** Even if strong guarantee is too expensive, preserve basic.

Example

```cpp
// Strong guarantee via copy-and-swap
class VecBox {
    std::vector<int> data_;
public:
    void assign(std::vector<int> v) { // copy/move may throw
        data_.swap(v);                // commit (nonthrowing)
    }
};
```

Constructors and Destructors

- Constructor establishes invariant or throws (E.5) – no partial objects.
- Destructor must not throw (E.16). If cleanup can fail, provide explicit `close()/commit()`; destructor does besteffort.

Note

Guarantee Checklist

1. Is `noexcept` used correctly on operations that cannot throw?
2. Can you provide strong guarantee where needed via copyandswap or buildthencommit?
3. Does every operation provide at least the basic guarantee (invariants + no leaks)?
4. Are destructors nonthrowing, and fallible cleanup exposed via explicit functions?
5. Are RAII and nothrow swaps/moves used to simplify safety?

CHAPTER 78

No Exceptions from Destructors

Rule (E.16): Destructors must never allow exceptions to escape. They are implicitly `noexcept`.

Why its critical

- During stack unwinding (another exception active), a throwing destructor calls `std::terminate`.
- Destructors run in many essential contexts: normal scope exit, partial construction, static destruction.
- RAII reliability depends on nonthrowing destructors.

Handling Cleanup That Can Fail

- **Provide explicit errorreporting function** (`close`, `flush`, `commit`) that may throw or return error.
- **Destructor does besteffort cleanup only** – never throws. Swallow/log errors internally.
- **Twophase pattern:** Destructor rolls back (nothrow); user must call `commit()` for error reporting.

Example

```
class OutFile {
    std::FILE* f_ = nullptr;
public:
    explicit OutFile(const char* path) {
        if (!(f_ = std::fopen(path, "wb"))) throw std::runtime_error{"open failed"};
    }
    void close() { // may throw
        if (f_ && std::fclose(f_) != 0) { f_ = nullptr; throw std::runtime_error{"close
        ↪  failed"}; }
        f_ = nullptr;
    }
    ~OutFile() noexcept { if (f_) (void)std::fclose(f_); } // best-effort
};
```

Never:

- Throw from a destructor directly.
- Call potentially throwing code inside a destructor unless caught and handled internally.

Note

Destructor Safety Checklist

1. Every destructor effectively `noexcept` (explicit or implicit).
2. Fallible cleanup provided via named functions, not destructor.
3. If catching inside destructor, policy is explicit (log, set flag, terminate).
4. RAII wrappers ensure no resource leaks even in destructor failure.

CHAPTER 79

Alternatives to Exceptions

When exceptions are disabled, maintain safety and clarity with these strategies (E.25–E.27).

Mandatory foundations regardless:

- **RAII still rules** (E.25) – guards against leaks on any return/error path.
- **Invariants must be preserved** (E.4) – even with error codes.
- **Systematic, consistent error propagation** (E.27) – avoid adhoc styles.

Primary alternatives

1. **Status codes + outparameters** – for functions where failu
   - Use a clear error enum/struct; define a single convention.
   - Enforce checking with `[[nodiscard]]`, code review, static

2. **`std::optional<T>`** – when absence is a normal outcome, not an error. No extra error info.

3. **Result type (valueorerror)** – returns either value or error details.
   - Combine a value and an error code/message in one struct.
   - Essential: must check before accessing value; use `[[nodiscard]]`.

Example

```cpp
template<class T>
class result {
public:
    static result success(T v);
    static result failure(errc e, std::string msg = {});
    bool ok() const noexcept;
    T& value();
    // ...
};
```

4. **Fail fast (terminate)** – when recovery is unsafe/infeasible (E.26).
   - Use `std::abort` or `std::terminate` on contract violations.
   - Appropriate for severe invariant breaches, no meaningful recovery.

5. **Error boundaries and translation** – wrap C APIs that use `errno` (E.28).
   - Capture `errno` immediately; return a local error object.
   - Centralize conversions at a single boundary layer.

Decision Guide

- Normal absence `std::optional`.
- Need error reason result type.
- Critical invariant violation fail fast.
- Wrapping C APIs translate to project error codes.

Note

Alternatives Checklist

1. Is RAII used everywhere for resources?
2. Is there a single, documented error code/result convention?
3. Are return values enforced as checked (`[[nodiscard]]`, tooling)?
4. Is failfast used only where recovery is truly unsafe?
5. Are Cstyle `errno` errors translated at the boundary?

Part XIII

Concurrency and Parallelism (CP)

CHAPTER 80

CP Guidelines – Condensed

The **CP** rules ensure correctness in multithreaded code. They cover sharedstate safety, tasks, coroutines, and message passing. Unassigned ranges: CP.57, CP.1019, CP.2730, CP.3339, CP.4549, CP.5459.

Baseline Rules (CP.1CP.4, CP.8CP.9)

- **CP.1: Assume multithreaded execution.** Write code that remains correct even when called concurrently.
- **CP.2: Avoid data races.** Any unsynchronized concurrent access where at least one is a write undefined behaviour.
- **CP.3: Minimize shared writable data.** Prefer immutable data, tasklocal state, and message passing.
- **CP.4: Think in tasks, not raw threads.** Use `std::async`, thread pools, and parallel algorithms.
- **CP.8: Never use `volatile` for synchronisation.** `volatile` does not establish happensbefore.
- **CP.9: Use concurrency validation tools.** Thread sanitizers, stress tests, deadlock detection.

SharedState Concurrency (CP.20CP.50)

- **CP.20: RAII locks only.** Never call `lock()`/`unlock()` manually; use `std::lock_guard`, `std::scoped_lock`.
- **CP.21: Lock multiple mutexes safely.** Use `std::scoped_lock` or `std::lock` to avoid deadlock.
- **CP.22: Never call unknown code (callbacks, virtuals) while holding a lock.** Copy state and call outside the critical section.
- **CP.23: A joined thread is a scoped container.** It must finish before scope exit; prefer `std::jthread` (C++20).
- **CP.24: An escaping thread is like a global object.** Its lifetimes must be explicitly managed.
- **CP.25: Prefer a joining thread.** Use `std::jthread` or equivalent; never raw `std::thread` that may be forgotten.
- **CP.26: Do not `detach()`.** Detached threads cause lifetime and shutdown hazards.

Ownership and Data Passing

- **CP.31: Pass small data by value.** Avoid references that may dangle when the task outlives the scope.
- **CP.32: Share ownership with `shared_ptr`.** For shared heap objects across threads; does not protect the objects internals.

Performance and Critical Sections

- **CP.40: Minimise context switching.** Batch work; avoid excessive finegrained locking.
- **CP.41: Minimise thread creation/destruction.** Use pools instead of threadperjob.
- **CP.42: Always wait with a predicate.** `cv.wait(lock, [&]{ return ready; });`
- **CP.43: Keep critical sections small.** Dont hold locks across I/O or heavy work.
- **CP.44: Name your lock objects.** A temporary `lock_guard` unlocks immediately give it a name.
- **CP.50: Bundle the mutex with the guarded data.** Encapsulate in a class; use `synchronized_value<T>` where available.

Coroutines (CP.51CP.53)

- **CP.51: Avoid capturing lambdas that are coroutines.** Captures can dangle; pass state by value.
- **CP.52: Do not hold locks across suspension points.** Release locks before `co_await`.
- **CP.53: Pass parameters to coroutines by value.** References may become dangling after suspension.

Message Passing and Task Results (CP.60CP.61)

- **CP.60: Use `std::future` to return a result or exception from a concurrent task.**
- **CP.61: Spawn tasks with `std::async`.** Prefer over manual `thread` + `promise`.

Example

Task with `std::async` and future

```
auto fut = std::async(std::launch::async, read_value, "v.txt");
// ... other work ...
int value = fut.get(); // synchronisation point, may rethrow
```

Note

Concurrency Review Checklist

1. Every shared writable object has a clear synchronisation strategy (mutex, atomic, task ownership).
2. No data races (CP.2) all accesses are properly ordered.
3. Locks are RAII only; no manual `lock/unlock`.
4. No callbacks while holding a lock.
5. Condition variables always wait with a predicate.
6. Threads are joined (prefer `jthread`); no `detach()`.
7. Lifetime of shared objects is managed (by value, `shared_ptr`, or clear ownership).
8. Concurrency tests and sanitizers are part of CI.

CHAPTER 81

Data Races

A data race occurs when two or more threads access the same memory location concurrently, at least one access is a write, and there is **no happens-before** ordering. The result is **undefined behaviour** not merely wrong values, but arbitrarily broken execution.

Common causes

- `++x` on an unsynchronized, nonatomic variable.
- Reading a shared variable while another thread writes it without synchronisation.
- Unsafe publication: a pointer/reference is made visible without a synchronising operation (lock release, atomic storerelease).
- `volatile` used instead of atomic/mutex `volatile` does not establish interthread ordering.

Example

Nonatomic increment data race

```
int counter = 0;
void worker() { for(int i=0;i<1'000'000;++i) ++counter; } // race
```

Fix: `std::atomic<int> counter{0};` or protect with a mutex.

Prevention strategies

1. **Eliminate sharing** give each thread its own data, share only immutable objects.
2. **Mutex protection** guard all accesses with a mutex; keep mutex and data together in one class (CP.50).
3. **Atomics** use `std::atomic` for simple shared scalars; combine with correct memory ordering (releaseacquire) for safe publication.
4. **Taskbased concurrency** use `std::async` and futures to return values instead of sharing mutable state.

Misconceptions

- `const` does **not** guarantee thread safety the object may be mutated through another reference.
- A single machine instruction does **not** guarantee absence of a data race the C++ memory model requires happensbefore.
- `volatile` is **not** a synchronisation mechanism.

Note

Data Race Prevention Checklist

1. Any shared, mutable data? eliminate sharing or add synchronisation.
2. All synchronisation uses RAII locks or correctly ordered atomics.
3. No concurrent read/write without a happensbefore relationship.
4. Tools: thread sanitizer enabled, stress tests pass.

CHAPTER 82

Mutexes

Core Rules (CP.20–CP.50)

- **CP.20:** RAII locks only never manual `lock()`/`unlock()`.
- **CP.21:** Acquire multiple mutexes with `std::scoped_lock` or `std::lock`.
- **CP.22:** Never call unknown code (callbacks, virtuals) while holding a lock.
- **CP.43:** Keep critical sections small.
- **CP.44:** Name your lock objects (prevent accidental temporary unlocks).
- **CP.50:** Bundle mutex and guarded data in the same class.

Choosing a Mutex Type

- `std::mutex` default, nonrecursive.
- `std::shared_mutex` for readheavy workloads (shared reads, exclusive writes).
- Avoid `std::recursive_mutex` usually indicates a design problem.
- Timed mutexes only for bounded waiting, not to fix deadlocks.

RAII Locking

- `std::lock_guard` simple scope lock.
- `std::scoped_lock` lock one or more mutexes (C++17).
- `std::unique_lock` deferred, manual unlock, required for condition variables.

Example

Encapsulation (CP.50)

```
class BankAccount {
    mutable std::mutex m_;
    int balance_ = 0;
public:
    void deposit(int amt) {
        std::lock_guard lk(m_);
        balance_ += amt;
    }
    int balance() const {
        std::lock_guard lk(m_);
        return balance_;
    }
};
```

Deadlock Avoidance

- **Consistent lock ordering** if manually locking multiple mutexes.
- **Use `std::scoped_lock` or `std::lock`** (CP.21) to acquire all at once.
- **No callbacks / unknown code** while holding a lock (CP.22). Copy state first, then invoke outside lock.

Example

Safe multimutex with `scoped_lock`

```
std::mutex m1, m2;
void transfer(Account& a, Account& b, int amount) {
    std::scoped_lock lk(m1, m2);
    a.debit(amount);
    b.credit(amount);
}
```

Condition Variables

- Always wait with a predicate overload (protects against spurious wake-ups).
- Predicate must check shared state protected by the **same mutex**.

Example

```
std::mutex m;
std::condition_variable cv;
std::queue<int> q;

void pop() {
    std::unique_lock lk(m);
    cv.wait(lk, [&]{ return !q.empty(); });
    int item = q.front(); q.pop();
    // ...
}
```

Pitfalls

- **Unnamed temporary:** `std::lock_guard(m);` unlocks immediately.
- **Returning references/pointers** to protected data escapes lock.
- **Holding lock across I/O** or heavy work (violates CP.43).
- **volatile** for synchronisation does not provide interthread ordering.

Note

Mutex Checklist

1. RAII locks only; named lock objects.
2. Critical sections as short as possible.
3. Multiple mutexes acquired with `scoped_lock` / `std::lock`.
4. No unknown code called under a lock.
5. Condition variable waits use a predicate.
6. Mutex and data colocated in the same class.

CHAPTER 83

Atomics

What `std::atomic` guarantees

- No data races on the atomic object itself.
- Interthread ordering and visibility when requested with memory orders.
- Does **not** automatically protect surrounding nonatomic data or complex invariants.

Atomics vs. Mutexes

- **Atomics:** single variable, simple operations, lockfree coordination.
- **Mutexes:** multivariable invariants, complex mutations, easier to reason about.

Essential Operations

- `load()` / `store()` read/write.
- `fetch_add()`, `exchange()` readmodifywrite.
- `compare_exchange_strong/weak()` conditional update (CAS).

Memory Ordering

- `relaxed` atomicity only; no ordering. Good for counters.
- `release` / `acquire` safe publication: release stores, acquire loads.
- `seq_cst` strongest; simplest model, global total order. Default.

Example

Safe publication (release/acquire)

```
std::atomic<bool> ready{false};
Payload data; // non-atomic

void producer() {
    data.x = 10; data.y = 20;
    ready.store(true, std::memory_order_release);
}
void consumer() {
    while (!ready.load(std::memory_order_acquire)) {}
    // now sees data.x == 10, data.y == 20
}
```

C++20 Additions

- `wait()`, `notify_one()` efficient blocking instead of spin loops.
- `std::atomic_ref<T>` atomic view on existing nonatomic object; must be the exclusive access path.

Common Mistakes

- `volatile` for synchronisation **not** threadsafe (CP.8).
- Mixing atomic and nonatomic accesses to the same object.
- Using `relaxed` when ordering is needed (e.g., publication).
- Complex lockfree algorithms without formal reasoning.

Note

Atomics Checklist

1. Use `seq_cst` as baseline; relax only with justification.
2. Publish data with release store + acquire load.
3. Counters: `relaxed` is sufficient for atomicity.
4. `atomic_ref`: ensure exclusive access while any reference exists.
5. Favour mutexes when protecting multiple variables.

CHAPTER 84

Deadlocks

Deadlock = cycle of waiting. Each thread holds a resource the other needs; no progress possible.

- Data race is undefined behaviour; deadlock is a liveness failure program freezes.
- Root causes: inconsistent lock order, calling unknown code under lock, waiting while holding unrelated locks, joining threads under locks.

Core Guidelines Prevention Rules

- **CP.20:** RAII locks only never manual `lock/unlock`.
- **CP.21:** Acquire multiple mutexes with `std::scoped_lock` or `std::lock`.
- **CP.22:** Never call unknown code (callbacks, virtuals, logging) while holding a lock.
- **CP.43:** Keep critical sections minimal.
- **CP.44:** Name your lock objects temporary locks unlock immediately.
- **CP.50:** Bundle mutex with guarded data.

Example

Inconsistent lock order deadlock

```
std::mutex m1, m2;
void f() { std::lock_guard l1(m1); std::lock_guard l2(m2); } // m1 then m2
void g() { std::lock_guard l2(m2); std::lock_guard l1(m1); } // m2 then m1  deadlock

// Fix: scoped_lock (C++17) or std::lock + adopt_lock
void safe() { std::scoped_lock lock(m1, m2); /* ... */ }
```

Example

Calling unknown code under lock (CP.22)

```
// Bad: callback called while holding lock
void for_each(const std::function<void(int)>& cb) {
    std::lock_guard lk(m_);
    for (int x : v_) cb(x); // cp.22 violation
}
// Good: snapshot under lock, call outside
void for_each_safe(const std::function<void(int)>& cb) {
    std::vector<int> snap;
    { std::lock_guard lk(m_); snap = v_; }
    for (int x : snap) cb(x);
}
```

Other deadly patterns

- **Condition variable without predicate**: waits may wake spuriously; always use predicate overload.
- **Waiting while holding another lock**: blocks producer from acquiring second lock deadlock. Only hold the CVs mutex during `wait()`.
- **Joining thread under lock**: if worker needs the lock to exit, join blocks forever. Release lock before join.

Engineering prevention strategies

- Minimise shared mutable state (immutable, message passing).
- One mutex per invariant; encapsulate with data (CP.50).
- Lock multiple mutexes safely: `scoped_lock`, or strict global lock order.
- Keep critical sections short (CP.43); no I/O, no unknown calls.
- Structured shutdown with `std::jthread` and `stop_token` never join under locks.

Note

Deadlock Prevention Checklist

1. Are all mutex acquisitions via RAII (named objects)?
2. Multiple mutexes locked with `scoped_lock` or consistent order?
3. No unknown code (callbacks, virtuals) called under lock?
4. Condition variables always used with predicate wait?
5. Never `wait()` while holding a second unrelated lock?
6. Thread joins performed only after releasing necessary locks?

CHAPTER 85

Concurrent System Design

Guiding Principles (CP.1CP.4, CP.20CP.22, CP.43CP.44, CP.50)

- Assume multithreaded (CP.1); avoid data races (CP.2).
- Minimise shared writable state (CP.3); think in tasks, not threads (CP.4).
- Encapsulate mutex with data (CP.50); RAII locks, short critical sections, no unknown code under locks.
- Use futures/async for task results; prefer message passing over shared mutable state.

Design Axes

- **Sharedstate vs. message passing**: prefer message passing / immutable sharing; if sharing, encapsulate locks.
- **Unstructured vs. structured concurrency**: use `std::jthread` (C++20) for RAII joining and cooperative cancellation (`stop_token`).
- **Blocking vs. nonblocking**: blocking (mutexes, CVs) for clarity; nonblocking (atomics) only when required and wellreasoned.

Component Safety Contract

- **Threadsafety level**: compatible, safe, or confined.
- **Ownership and lifetime**: clear owner, no dangling references.
- **Shutdown protocol**: stop request, drain policy, destructor behaviour.
- **Error propagation**: via futures, status codes, or explicit error messages.

Example

Structured shutdown with jthread

```
void worker(std::stop_token st) {
    while (!st.stop_requested()) { /* do work */ }
    // clean exit
}
int main() {
    std::jthread t(worker);  // RAII join
    t.request_stop();        // cooperative cancellation
}
```

Choosing Primitives

- **Mutex + CV**: for protecting invariants and waiting for state.
- **Atomics**: single shared flags/counters; releaseacquire for publication.
- **Atomic wait/notify** (C++20): efficient blocking on atomics.
- **Latches/barriers** (C++20): coordinate phases; `semaphore` for resource limits.
- **Parallel algorithms**: for dataparallel work (ensure grain size sufficient).
- **Thread pool**: amortise thread creation; bound queue; structured shutdown.

Performance Essentials

- Reduce contention first (shard locks, perthread data).
- Avoid false sharing (pad cache lines).
- Bound concurrency to cores; measure, dont guess.
- Clear correctness > clever lockfree tricks.

Note

Design Review Checklist

1. No data races; every shared writable object has a single synchronisation strategy.
2. Ownership explicit; no dangling references into async work.
3. No detached threads; all work stoppable.
4. RAII locks, safe multilock, no callbacks under locks, short critical sections.
5. Shutdown designed (stop, drain, join).
6. Error propagation exists (futures, error channels).
7. Contention measured; scaling bottlenecks identified.

CHAPTER 86

Message Passing

Core Guidelines (CP.60, CP.61, CP.2, CP.3)

- **CP.60:** Use `std::future` to return a value from a concurrent task.
- **CP.61:** Use `std::async()` to spawn concurrent tasks (specify `launch::async` for guaranteed concurrency).
- Avoid shared writable state; send messages (values) instead.

Building Blocks

- **`std::promise/future`** oneshot result channel; `get()` blocks; exceptions propagated.
- **`std::packaged_task`** wrap callable + future; decouple execution from result delivery.
- **Queue channels** multimessage; mutex + condition variable with predicate wait; explicit close/drain logic.
- **Actor model** singlethreaded owner processes messages sequentially; state never shared.
- **Atomic wait/notify (C++20)** efficient signal without spinning.

Example

Actor skeleton (C++20 jthread)

```
class SumActor {
    std::mutex m_; std::condition_variable cv_;
    std::queue<Msg> q_; std::jthread thr_;
    int sum_ = 0; // owned state, accessed only in loop()

    void loop(std::stop_token st) {
        while (!st.stop_requested()) {
            Msg msg;
            {
                std::unique_lock lk(m_);
                cv_.wait(lk, [&]{ return st.stop_requested() || !q_.empty(); });
                if (st.stop_requested()) break;
                msg = std::move(q_.front()); q_.pop();
            }
            // process msg, update sum_
        }
    }
public:
    void send(Msg m) {
        { std::lock_guard lk(m_); q_.push(std::move(m)); }
        cv_.notify_one();
    }
    // ...
};
```

Message Design Rules

- Messages are **values** move them; avoid borrowed references.
- Ownership transfers via `std::unique_ptr` within messages if needed.
- Error reporting: either use `std::variant<Result, Error>` or `std::promise` to set exception.

Shutdown and Cancellation

- Combine `std::stop_token` (cooperative stop) with queue `close()` (unblock waiters).
- Draining policy: process remaining messages or discard? document explicitly.
- Never join/block on threads while holding locks they might need.

Note

Message Passing Checklist

1. No shared writable state outside channels.
2. Messages are selfcontained values (no dangling pointers).
3. Channel contract: ordering, capacity, close semantics documented.
4. Error strategy explicit (exceptions via futures, error objects via messages).
5. Shutdown: stop request + channel close; no detached threads.
6. No callbacks under locks.

Part XIV

Performance (Per)

CHAPTER 87

Per.1–Per.14: Rules Condensed

The Performance rules apply when you have a *real, measurable* performance requirement. Correctness and clarity come first.

Core Principles

- **Per.1 – Don't optimize without reason.** Have a measurable goal.
- **Per.2 – Don't optimize prematurely.** Simple code first; optimize only when needed.
- **Per.3 – Focus on performancecritical paths.** Profile to find hotspots.
- **Per.4 – Complicated code is not automatically faster.** Compilers optimise simple code well.
- **Per.5 – Highlevel code can be as fast as lowlevel code.** Use standard algorithms, `std::sort` over `qsort`.
- **Per.6 – Measure, don't guess.** Never claim performance without data. Use `std::chrono::steady_clock`.

Enabling Optimisation (Design for Performance)

- **Per.7 – Design to enable optimisation.** Keep interfaces typed; avoid `void*`; make costs predictable.
- **Per.10 – Rely on the static type system.** Templates, strong enums, `std::variant` give the compiler more information.
- **Per.11 – Move computation to compile time.** Use `constexpr`, `consteval`, constant initialisation.

Removing Unnecessary Overhead

- **Per.12 – Eliminate redundant aliases.** Reduce cases where two pointers/references might target the same object; pass `span` and return values.
- **Per.13 – Eliminate redundant indirections.** Avoid pointerchasing and virtual dispatch in hot loops; prefer contiguous storage.
- **Per.14 – Minimise allocations/deallocations.** Use `reserve()`, reuse buffers, prefer stack allocation; `std::pmr` when central control is needed.

Note

Performance Workflow Checklist

1. Define the performance metric and goal.
2. Measure baseline (Per.6).
3. Identify hotspots with a profiler (Per.3).
4. Apply optimisations: start with algorithms/data structures, then reduce allocations (Per.14), indirections (Per.13), aliasing (Per.12).
5. Remeasure; keep only changes that improve the metric.

CHAPTER 88

When to Optimize

Optimise only when:

- A **measurable performance requirement** exists (latency, throughput, memory limit). (Per.1)
- **Measurements show a real bottleneck** (profiler data, production telemetry). (Per.3, Per.6)
- A **scalability bottleneck** is proven (e.g., `O(n²)` algorithm on critical path).

Do *not* optimise when:

- No measurable bottleneck exists.
- The system is I/O bound or limited externally.
- You cannot reproduce the workload reliably (Per.6).

The reliable workflow

1. State the goal in measurable terms.
2. Establish a baseline measurement (`steady_clock`, repeated runs).
3. Profile to locate the dominant cost.
4. Improve algorithms/data structures first they give the largest, cleanest wins.
5. Apply microoptimisations only if needed, keeping code maintainable.
6. Remeasure and compare to baseline; discard changes that dont help.

Note

Measurement Fundamentals

- Use `std::chrono::steady_clock` for interval timing.
- Benchmark release builds with optimisations enabled.
- Run multiple iterations; report min/median/max.
- Isolate from noise (other processes, variable CPU frequency).

CHAPTER 89

Measure Before Optimizing

Rule Per.6: Never make performance claims without measurements. Intuition and folklore are not substitutes for data.

Good Measurement Practices

- Use a monotonic clock (`steady_clock`).
- Warm up caches and the allocator; discard first runs if measuring steady state.
- Repeat many times; analyse statistical distribution.
- Compare before/after under exactly the same conditions (compiler flags, input, hardware).

Profiling vs. Benchmarking

- **Profiling** find where time is spent (sampling, instrumentation).
- **Microbenchmarks** measure isolated operations; useful for comparing alternatives.
- **System benchmarks** measure endtoend behaviour under realistic load.

Example

```cpp
template<class F>
auto measure(F&& f, std::size_t iters) {
    using clock = std::chrono::steady_clock;
    auto start = clock::now();
    for (std::size_t i = 0; i < iters; ++i) f();
    return std::chrono::duration_cast<std::chrono::nanoseconds>(clock::now() - start);
}
```

Note

Common Pitfalls

- Measuring debug builds.
- Measuring onetime initialisation instead of steady state.
- Too few iterations noise dominates.
- Changing compiler flags or environment between runs.
- Ignoring variability; always capture multiple samples.

CHAPTER 90

Cost Models

A cost model helps you *predict* where time and memory go before measuring. It guides design but must be validated (Per.6).

Two Layers

- **Algorithmic** BigO, amortised complexity. Choose the right algorithm family.
- **Machine** cache, allocations, branching, synchronisation. Explain constantfactor differences.

Standard Library Complexity Guarantees

- `std::sort` `O(N log N)`.
- `std::vector::push_back` amortised `O(1)`; reallocation can be `O(N)`.
- `std::unordered_map` average `O(1)` lookup; worstcase and rehashing cost.

Machine Costs to Model

- **Memory locality:** contiguous storage few cache misses; pointer chasing many.
- **Allocations:** expensive, variable; minimise on hot paths (Per.14).
- **Branching:** unpredictable branches stall pipelines; prefer branchless code only when measured.
- **Synchronisation:** contention on locks/atomics causes serialisation and cacheline bouncing.

API Cost Transparency

- Nonowning views (`span`, `string_view`) cheap observation.
- Returning by value efficient with move semantics and copy elision.
- Explicit ownership transfer (`unique_ptr`) clear lifetime and allocation cost.

Note

Cost Model Checklist

1. What scales with input? Identify `O(...)`.
2. Where are the allocations? Can they be removed? (Per.14)
3. What is the memory access pattern? Contiguous or pointerchasing?
4. Any contended locks/atomics?
5. Does the interface hide expensive work?
6. Confirm predictions with measurement (Per.6).

CHAPTER 91

Cache-Friendly Code

Memory access often dominates execution time. Cachefriendly code respects locality and minimises cache misses.

Core Guidelines

- **Per.18: Space is time** smaller, compact data improves cache utilisation.
- **Per.19: Access memory predictably** linear, contiguous access patterns are ideal.

Rules of Thumb

- **Prefer contiguous storage** `std::vector` for hot data; avoid linked lists for traversal.
- **Iterate in memory order** rowmajor traversal for multidimensional arrays.
- **Block (tile) large loops** reuse small chunks to keep data in cache.
- **Split hot and cold fields** keep frequently used members together in a compact struct; move rarely used data elsewhere.
- **Structure of Arrays (SoA)** when processing one field across many objects, store fields in separate contiguous arrays.
- **Avoid false sharing** pad threadlocal counters to cacheline boundaries (`alignas(64)` typical).

Example

SoA for fieldwise processing

```
struct Particles {
    std::vector<float> x, y, z;
    std::vector<float> vx, vy, vz;
};
void update_x(Particles& p, float dt) {
    for (size_t i = 0; i < p.x.size(); ++i)
        p.x[i] += p.vx[i] * dt; // linear, cachefriendly
}
```

Common Antipatterns

- Pointerbased linked lists for hot sequential traversal.
- Columnmajor access on rowmajor data.
- Allocating many small objects separately hurts locality and increases fragmentation.
- False sharing between threads writing to adjacent variables.

Note

CacheFriendly Checklist

1. Are hot data structures stored contiguously? (`vector`, `array`)
2. Are critical loops traversing memory in linear order?
3. Can working sets be shrunk or blocked?
4. Are hot and cold fields separated?
5. In concurrent code, are perthread counters padded to avoid false sharing?
6. Validate improvements with measurement (Per.6).

Part XV

The Standard Library (SL)

CHAPTER 92

SL.1–SL.16: Guidelines Condensed

The **SL** rules map to the most essential standard library guidance. For sequential numbering, we assign:

- SL.1–SL.4 = toplevel rules
- SL.5–SL.8 = SL.con.1–SL.con.4 (containers)
- SL.9–SL.13 = SL.str.1–SL.str.5 (strings/bytes)
- SL.14–SL.16 = SL.io.1–SL.io.3 (iostreams)

SL.1 (SL.1): Use libraries wherever possible. Prefer standard algorithms and containers over handwritten code.
SL.2 (SL.2): Prefer the standard library to other libraries. It is stable, widely known, and portable.
SL.3 (SL.3): Do not add entities to `std`. Except where the standard explicitly permits specialisation for userdefined types.
SL.4 (SL.4): Use the standard library in a typesafe manner. No rawmemory tricks on nontrivial objects.

SL.5 (SL.con.1): Prefer `std::array` or `std::vector` over C arrays.
SL.6 (SL.con.2): `std::vector` is the default container.
SL.7 (SL.con.3): Avoid bounds errors; use range`for`, algorithms, and `.at()` at boundaries.
SL.8 (SL.con.4): No `memset`/`memcpy` on nontriviallycopyable types; use `std::fill`/`std::copy`.

SL.9 (SL.str.1): `std::string` owns character sequences.
SL.10 (SL.str.2): `std::string_view` or `span<char>` for nonowning views.
SL.11 (SL.str.3): Use `zstring`/`czstring` for Cstyle zeroterminated strings when required.
SL.12 (SL.str.4): `char*` refers to a single character, not a string.
SL.13 (SL.str.5): `std::byte` for noncharacter byte values.

SL.14 (SL.io.1): Use formatted I/O; manual characterlevel input only when necessary.
SL.15 (SL.io.2): Always validate input treat all external data as potentially malformed.
SL.16 (SL.io.3): Prefer iostreams for I/O in C++; use `sync_with_stdio(false)` for performance when mixing is not needed.

Note

Standard Library Quick Checklist

1. Use standard facilities; do not reinvent.
2. Never add to `std`; specialise only where allowed.
3. Default container: `std::vector`.
4. Use views (`string_view`, `span`) for nonowning access; ensure lifetime.
5. Avoid raw byte operations on nontrivial types.
6. Validate input early.

CHAPTER 93

Proper Use of the STL

STL mental model: Containers hold data, algorithms process ranges, iterators/views connect them. Prefer `algorithm(range, ...)` over manual loops.

Algorithms over raw loops

- Intent is explicit: `sort`, `find`, `transform`.
- Reduces offbyone and indexing bugs.
- Use eraseremove idiom: `v.erase(remove(begin,end,val), end);`
- Prefer `std::transform` instead of manual mapping loops.

Container choice

- Default: `std::vector` (contiguous, cachefriendly).
- Fixed size: `std::array`.
- Ordered keys: `std::map/set` (logarithmic complexity).
- Hashbased: `std::unordered_map/set` (average O(1)).
- Understand invalidation rules: `vector` reallocation invalidates all iterators/references.

Views and lifetime

- `std::string_view` readonly, nonowning; caller guarantees lifetime.
- `std::span` nonowning contiguous view; safer than pointer+size.
- Ranges (C++20) pipeline transformations; still nonowning views.

Value semantics and RAII

- Owning via `unique_ptr`/`shared_ptr` (use shared only when needed).
- No manual `new`/`delete`; use `make_unique`.
- `reserve()` to avoid reallocations.

Common pitfalls

- `std::vector<bool>` is a bitset, not a normal container; prefer `vector<unsigned char>` or `vector<std::byte>`.
- Ignoring iterator invalidation after push/erase/rehash.
- Raw byte ops (`memset`/`memcpy`) on nontrivial objects.
- Accidental `O(n^2)` loops when `O(n)` is possible.

Note

STL Usage Checklist

1. Default: `vector`. Fixed: `array`.
2. Replace loops with algorithms where clearer.
3. Nonowning parameters: `string_view` / `span`.
4. Understand iterator invalidation for chosen container.
5. No raw memory ops on nontrivial types.
6. `reserve` when size predictable; avoid quadratic growth.

CHAPTER 94

Containers

Standard containers ensure safe, performant data storage. Choose by access pattern and lifetime needs.

Container selection flow

1. **Contiguous storage needed?** `vector` (default), `array` (fixed).
2. **Stable references across insert/erase?** consider `deque` (end ops) or nodebased `list` / `map` (but measure).
3. **Fast random access?** `vector`, `array`, `deque`.
4. **Ordered by key?** `map`/`set`.
5. **Hashbased lookup?** `unordered_map`/`set`.

Core Guidelines (SL.con)

- **SL.con.1:** `array`/`vector` over C arrays they carry size.
- **SL.con.2:** `vector` as default contiguous, simple.
- **SL.con.3:** No bounds errors range`for`, algorithms, `.at()` at boundaries.
- **SL.con.4:** No raw byte functions on nontriviallycopyable types.

Example

```
// Avoid keeping references across growth
std::vector<int> v{1,2,3};
int& r = v[0];
v.push_back(4); // may invalidate r
```

Erase patterns

- `vector`: eraseremove idiom.
- `map`/`unordered_map`: `it = m.erase(it)` while iterating.

Performance tips

- `reserve()` when final size is known.
- Contiguous storage wins for traversal; nodebased structures may hurt locality.
- `std::pmr` containers for custom allocation strategies.

Ownership

- Store values by default.
- Use `unique_ptr` in containers for exclusive ownership.
- `shared_ptr` only when shared ownership is genuinely required.

`std::span` safe view

```
int sum(std::span<const int> a) { // size travels with pointer
    int s=0; for(int x:a) s+=x; return s;
}
```

Note

Container Checklist

1. Default: `vector`; fixed size: `array`.
2. Avoid `vector<bool>` when standard element references are needed.
3. Know and respect iterator/reference invalidation rules.
4. Reserve capacity when growth is predictable.
5. Prefer contiguous layouts for hot paths; validate with measurement.

CHAPTER 95

Algorithms

Use standard algorithms instead of handwritten loops. They express intent, reduce bounds/index mistakes, and are highly optimised.

The Algorithm Model

- Classic: halfopen iterator pair `[first, last)`.
- C++20 ranges: `std::ranges::sort(v)` (accept a whole range), support **projections** to extract keys.

Example

Projection with `std::ranges::sort`

```
struct Item { int id; int score; };
std::ranges::sort(items, {}, &Item::score); // compare by score
```

Core Principles

- Prefer algorithms over raw loops names document intent.
- Respect contracts: valid iterators, no overlap where forbidden, strict weak ordering for comparators.
- Understand iterator invalidation dont keep iterators across container modifications.

Algorithm Categories

- **Nonmodifying:** `find`, `any_of`, `count`
- **Modifying:** `transform`, `replace`, `remove/remove_if`
- **Sorting:** `sort`, `stable_sort`, `nth_element`, `partial_sort`
- **Binary search** on sorted ranges: `lower_bound`, `upper_bound`
- **Numeric:** `accumulate` (ordered), `reduce` (may reorder, C++17)

EraseRemove Idiom

```
v.erase(std::remove(v.begin(), v.end(), value), v.end());
```

Parallel Algorithms (C++17+)

- Use `std::execution::par` for parallel, `par_unseq` for parallel+vectorised.
- **Critical:** no data races; callables must be safe for concurrent invocation.
- Exceptions from user callables cause `std::terminate`.

Ranges Pipelines (C++20) Lazy view composition with `|`: `v | filter(...) | transform(...)`. Materialise into a container when ownership is needed. Views are nonowning; underlying data must outlive the view.

Note

Algorithms Checklist

1. Replace manual loops with named algorithms where they express intent.
2. Use `std::ranges` and projections to reduce boilerplate.
3. Ensure comparators satisfy strict weak ordering.
4. Use eraseremove idiom for sequence containers.
5. No shared mutable state in parallel/unsequenced callbacks.
6. Measure; dont assume performance.

CHAPTER 96

Ranges

C++20 Ranges generalise the STL: a **range** is anything with `begin()` / `end()`. A **view** is a lazy, nonowning, cheaptocopy range. Constrained algorithms accept a whole range and support projections.

Why Ranges?

- **Safer:** no accidental iterator mismatches, borrowedrange/dangling safety.
- **Cleaner:** `std::ranges::sort(v)` instead of `std::sort(v.begin(), v.end())`.
- **Composable:** pipelines with `|`, each stage a small, testable view.
- **Lazy:** no intermediate allocations unless you materialise.

Example

Filter + Transform Pipeline

```
auto view = v | std::views::filter([](int x){ return x%2==0; })
              | std::views::transform([](int x){ return x*x; });
// materialise: std::vector<int> res(view.begin(), view.end());
```

Projections (C++20) eliminate custom comparators:

```
std::ranges::sort(users, {}, &User::id);   // sort by id
std::ranges::find(users, 42, &User::id);   // find by id
```

Lifetime Rules

- Views refer to their underlying data; do not let the data be destroyed while the view is alive.
- Returning a view of a local variable dangling.
- `std::ranges::borrowed_range`: ranges where iterators remain valid even when the range is a temporary. Otherwise, constrained algorithms may return `dangling` to prevent use of invalid iterators.

`std::ranges::subrange` bundles an iterator and sentinel, often returned by algorithms (e.g., `remove`).

```
auto r = std::ranges::remove(v, x); // returns subrange
v.erase(r.begin(), r.end());
```

Contracts

- Predicates/projections must be sideeffect free.
- Comparators must form a strict weak ordering.
- Iterator category must meet algorithm requirements (e.g., `sort` needs randomaccess).

Performance Views are cheap objects, but iterator complexity may be higher. Measure before microoptimising.

Note

Ranges Checklist

1. Prefer `std::ranges` algorithms for direct range usage and projections.
2. Use views for pipelines; materialise when ownership or repeated passes are needed.
3. Ensure underlying data outlives all views.
4. Keep predicates and projections pure.
5. Be aware of `dangling` when passing temporaries.

CHAPTER 97

Avoiding Reinvention

Core Guidelines: SL.1 Use libraries wherever possible. **SL.2** Prefer the standard library.

Why Reinvention is Harmful

- **Correctness:** handwritten loops/containers miss edge cases, invalidation, and UB.
- **Security:** manual buffer/ownership management invites overflows, leaks, doublefree.
- **Performance:** standard implementations are heavily optimised; custom versions often slower.
- **Maintenance:** nonstandard utilities require internal docs, onboarding cost, and ongoing fixes.

Common Reinventions Replacements

- **Dynamic array** `std::vector`
- **Fixedsize array** `std::array`
- **Raw owning pointer** `std::unique_ptr` / `std::shared_ptr`
- **Search/loop** `std::find`, `std::any_of`, `std::transform`
- **Remove elements** eraseremove idiom
- **Sort** `std::sort`, `std::partial_sort`, `std::nth_element`
- **Owned text** `std::string`; **readonly text** `std::string_view`
- **Pointer+length** `std::span`
- **Raw bytes** `std::byte`
- **Timing** `std::chrono`
- **Concurrency primitives** `std::mutex`, `std::lock_guard`, `std::jthread`, etc.

When Reinvention is Justified Only when the standard library cannot meet a **proven** requirement (specific data structure, certified subset, hard performance constraint). Even then, build on standard types and idioms, keep custom code minimal, and document the reason.

AntiReinvention Workflow

1. Identify custom utilities that duplicate standard features.
2. Replace internal interfaces with standard vocabulary types.
3. Replace raw loops with algorithms/ranges where clarity improves.
4. Test before and after; measure performance if relevant.

Note

Avoid Reinvention Checklist

1. Is there a standard library equivalent?
2. Can you compose standard components to solve the problem?
3. Use standard vocabulary types for ownership, ranges, and views.
4. Prefer algorithms and ranges over raw loops.
5. Write custom code only when there is a documented, unmeetable need.

CHAPTER 98

File Organization

Separate interface (headers) from implementation (source files) to prevent ODR violations and enable modular builds.

Scalable Directory Layout

```
project-root/
  include/mylib/   # public headers
  src/             # .cpp + private headers
  tests/
  examples/
```

Header Content Rules Headers should contain only:

- Declarations, templates, `inline`/`constexpr` functions, `extern` declarations.

Never place noninline function definitions or mutable global object definitions in a header.

SelfContained Headers Every header must `#include` everything it uses. It must compile when included alone (verify with a CI test).

Include Guards Use traditional guards with a unique, projectprefixed name. Portable, ISO C++ compliant.

```
#ifndef MYLIB_FOO_H
#define MYLIB_FOO_H
// ...
#endif
```

Include Order

- `#include` all headers at the top of the file.
- In a `.cpp`, include its own header first.
- Do not rely on transitive includes; include what you use.

Avoid Cycles and Transitive Dependencies

- No circular includes. Break cycles with forward declarations, smaller common headers, or PImpl.
- Never depend on headers that are only pulled in transitively; explicitly include what you need.

Forward Declarations Use forward declarations for pointer/reference members and parameters, and for PImpl. Do not forwarddeclare if you need the complete type (value members, inheritance, `sizeof`).

Example

PImpl with Forward Declaration

```
// widget.h
class widget_impl;
class widget {
    std::unique_ptr<widget_impl> p_;
public:
    explicit widget(std::string name);
    ~widget();
    // ...
};
// widget.cpp defines widget_impl
```

No `using namespace` in Headers It pollutes every includers scope. Use fully qualified names in headers. In `.cpp` files, limited `using` may be acceptable.

Build Hygiene

- Compile every public header in isolation (CI check).
- Detect include cycles.
- For precompiled headers / unity builds, keep a separate build that compiles each translation unit independently to catch missing includes.

Note

File Organization Checklist

1. Public headers under `include/`, sources in `src/`.
2. Headers contain only declarations, templates, inline, and constexpr.
3. Every header is selfcontained (includes what it uses).
4. Include guards present, with unique names.
5. `.cpp` includes its own header first; no reliance on transitive includes.
6. No `using namespace` in headers; no cyclic includes.
7. Forward declarations used where sufficient; otherwise full include.
8. CI verifies headers compile alone and cycles are absent.

Part XVI

Source Files and Build Structure (SF)

CHAPTER 99

Header Files, Guards, and Modules

The SF rules ensure robust, selfcontained headers and a modular build structure. Below, the essentials of header design, include guards, and modern modules.

Header Files

Headers contain interfaces, not definitions (SF.2). Place declarations, templates, `inline`, `constexpr`, and `extern` in headers. Never put noninline function definitions or mutable global objects there they cause ODR violations.

Selfcontained headers (SF.10, SF.11). Every header must include all its prerequisites. It must compile when included alone. No reliance on transitive includes.

Example

```cpp
// widget.h
#ifndef MYLIB_WIDGET_H
#define MYLIB_WIDGET_H
#include <string>
#include <vector>

namespace mylib {
class widget {
    std::string name_;
    std::vector<std::string> tags_;
public:
    explicit widget(std::string name);
    // ...
};
}
#endif
```

Include order and consistency. Place all `#include`s at the top. In `.cpp`, include its own header first. This catches missing includes early (SF.5).

No `using namespace` in headers (SF.7). It pollutes every includers scope. Fully qualify names in headers (`std::string`). Narrow `using` declarations inside functions are acceptable.

Forward declarations and PImpl. Forwarddeclare classes used as pointers/references to reduce header dependencies. Use the PImpl idiom to hide implementation details and heavy includes.

Macros in headers. Avoid macros that change interfaces. Prefer `constexpr`, `enum`, inline functions. If unavoidable, use a unique project prefix and `#undef` after use.

Note

Header Checklist

1. Includes what it uses; compiles alone.
2. No noninline definitions; no mutable global objects.
3. Include guard present, unique, projectprefixed.
4. No `using namespace` at global scope.
5. Forward declarations used where sufficient; PImpl for heavy internals.

Include Guards

Every header must have an include guard (SF.8). A standard, portable pattern:

```
#ifndef MYLIB_WIDGET_H
#define MYLIB_WIDGET_H
// header content
#endif
```

Guard macro naming. Use a unique, projectprefixed name derived from the include path (e.g., `MYLIB_DETAIL_WIDGET_IMPL_H`). Avoid collisions that can silently skip a header.

`#pragma once` is not ISO C++. The Core Guidelines prefer portable ISO C++ guards. Use `#pragma once` only if you accept the nonstandard semantics and test thoroughly.

Guards plus selfcontained headers. Guards prevent repeated inclusion, but you still need SF.10 and SF.11: include what you use, and headers that compile alone.

Note

Guard Checklist

1. One guard per header, near the top.
2. Unique macro name (project prefix + path).
3. No reliance on `#pragma once` for portability.

Modules (C++20+)

Modules provide a languagesupported alternative to textual inclusion. They reduce repeated parsing, macro leakage, and fragile transitive includes.

Named Modules

- **Module interface unit** (`export module name;`) exports declarations.
- **Module implementation unit** (`module name;`) provides definitions.
- Partitions allow splitting a module while keeping one logical name.

Example

```
// math.ixx
export module math;
export int add(int a, int b);

// math.cpp
module math;
int add(int a, int b) { return a + b; }

// main.cpp
import math;
```

Header Units. `import <vector>;` treats a standard header as an importable unit (requires build system support). Header hygiene still matters.

Global Module Fragment. For mixed code, include legacy headers before the module declaration:

```
module;
#include <cstdint>
export module net;
```

Standard Library Modules. `import std;` is available on some implementations (e.g., MSVC). Use only when your toolchain and build system fully support module dependency scanning.

Build system requirements. Modules need compilation order: interfaces before consumers. Use a build system with module scanning (CMake 3.28, `CMAKE_EXPERIMENTAL_CXX_MODULE_CMAKE_API`).

Pragmatic adoption:

- First enforce strict header hygiene (selfcontained, no transitiveinclude dependencies).
- Convert stable, frequently used interfaces to named modules.
- Keep compatibility headers; expect a mixed codebase for years.

Note

Modules Checklist

1. Use named modules for new, stable library boundaries.
2. Keep headers as fallback; ensure they remain selfcontained.
3. Build system configured for module dependency scanning.
4. Avoid macroheavy interfaces in modules; use typed alternatives.

Part XVII

Const Correctness (Con)

CHAPTER 100

Con.1 – Con.4, Objects, Member Functions, Logical vs. Bitwise

Core Rules (Con.1–Con.4)

Con.1: By default, make objects immutable. Declare variables `const` unless they must change. Do not apply to passbyvalue parameters or returned locals where `const` would inhibit moves.

Con.2: By default, make member functions `const`. Mark member functions `const` unless they change the objects observable state. This enables use on `const` objects and documents intent.

Con.3: By default, pass pointers and references to `const`. Use `const T&` / `const T*` when the callee does not modify the argument. Never cast away const (except for knownsafe legacy wrappers).

Con.4: Use `const` for values that never change after construction. Eliminates reader uncertainty and prevents accidental modification.

`const` Objects

Meaning: A `const` object cannot be modified after construction. Modification through a cast is undefined behaviour if the original object is truly const.

Initialization

- `const` objects must be initialized. `const` data members must be initialized in the member initializer list.
- Prefer `constexpr` for compiletime constants; `constinit` to enforce static initialization of static/thread storage objects.

Linkage in Headers

- Plain `const` at namespace scope has internal linkage. For a shared constant in a header, use `inline constexpr` (C++17).
- `constexpr` does not automatically make a variable `inline`; use `inline constexpr` to avoid ODR issues.

Const and Member Functions

- `const` objects can only call `const` member functions (Con.2).
- Logical constness may use `mutable` for caches (see below).

Const is not deep: `const` does not make pointedto data const. Modifying `*p` through a `const` object is allowed by the language but must respect the types logical constness contract.

Note

`const` Objects Checklist

1. Are variables that never change declared `const` (Con.1, Con.4)?
2. Are namespacescope header constants `inline constexpr`?
3. Are `const` data members initialized in the member initializer list?
4. No `const_cast` writes to truly const objects (undefined behaviour).
5. `constinit` used where dynamic initialization must be avoided.

`const` Member Functions

Inside a `const` member function, `*this` is `const`. Non`mutable` members cannot be modified, and only `const` member functions can be called. A `const` object can only call `const` members.

Overloading on Const

- Provide `const` overload returning `const T&` (readonly) and non`const` overload returning `T&` (mutable) for accessors.
- The `const` overload is selected for `const` objects.

Refqualifiers (`&` and `&&`) prevent dangerous references to temporaries. Example: `const std::string& get() const &;` for lvalues, `std::string get() &&;` for rvalues.

Logical Constness and `mutable`

- `mutable` allows modification in `const` functions. Use only for caches, instrumentation, or synchronisation primitives never to change observable logical state.
- If internal writes occur, ensure thread safety (mutex/atomics) when the object may be used concurrently.

Composing with other specifiers

- Mark observers `const` + `noexcept` + `[[nodiscard]]` where appropriate.
- `constexpr` can be combined with `const` for compiletime evaluation.

C++23 explicit object parameter can replace multiple cv/refqualified overloads with a single template, deducing the objects type.

Note

`const` Member Functions Checklist

1. Are observer/query functions marked `const` (and usually `noexcept`)?
2. Are `const` and non`const` overloads provided for accessors returning references?
3. Is `mutable` used only for nonsemantic data (caches, locks)?
4. If a `const` function writes `mutable` state, is it threadsafe?
5. No non`const` references returned from `const` functions.

Logical vs. Bitwise Const

Bitwise const the objects bits do not change. Simple, but too strict for caches. **Logical const** the objects *observable meaning* does not change, even if internal bits (caches, locks) are modified. C++ supports logical const via `mutable`.

Design Contract

- `mutable` separates physical state from logical state. The caller must not be able to detect the write through the public interface (except possibly via timing).
- If a `const` function writes internal state, ensure:
  - No observable semantic change.
  - Thread safety (mutex or atomic) if concurrency is possible.

Example

```
class Date {
    mutable std::string cache_;
    mutable bool cache_valid_ = false;
    // ...
public:
    const std::string& text() const {
        if (!cache_valid_) { cache_ = build(); cache_valid_ = true; }
        return cache_;
    }
};
```

Pitfalls

- Using `mutable` to change observable state (e.g., decrementing a balance) breaks the const contract.
- Transitive writes through pointers: `const` does not protect pointedto objects; ensure such writes are logically const or document carefully.
- "`const` == threadsafe" is false; you must add synchronisation if internal writes occur in a shared object.

Note

Logical/Constness Checklist

1. For every `const` function performing writes: is the data truly nonobservable (cache, stats)?
2. Can the caller detect the write besides performance?
3. If used concurrently, is the mutable state synchronised?
4. Could caching be removed in favour of recomputation (simpler)?

Part XVIII

Type Safety (Type)

CHAPTER 101

Type Safety Rules (Condensed)

Core Rules (Type.1–Type.7)

Type.1: Avoid casts. Use construction (especially `T{e}`) for conversions. When narrowing is intentional, use `gsl::narrow` (checked) or `gsl::narrow_cast` (explicit).

Type.2: Do not use `static_cast` to downcast; use `dynamic_cast` instead. `static_cast` downcast is unchecked undefined behaviour if wrong. `dynamic_cast` returns `nullptr` (pointer) or throws `std::bad_cast` (reference) on failure.

Type.3: Do not use `const_cast` to cast away `const`. Modifying an originally `const` object is undefined behaviour. If forced by legacy APIs, isolate the cast in a small wrapper.

Type.4: Do not use Cstyle or functional casts. Prefer construction or named casts (`static_cast`, `dynamic_cast`, etc.). Cstyle casts hide intent and can perform multiple dangerous conversions.

Type.5: Do not use a variable before it has been initialized. Always initialize objects. Prefer `T{}` (valueinitialization) or explicit initial values.

Type.6: Always initialize a data member. Use default member initializers, or initialize in every constructors member initializer list.

Type.7: Avoid naked `union`; use `std::variant` instead. `std::variant` tracks the active alternative; `std::get` / `std::visit` provide safe access.

Practical Cast Guidance

General rule: avoid casts. Design interfaces that dont require them. If unavoidable, use the safest named cast.

Example

Prefer braceinitialization for numeric conversions

```
double d = 3.14;
int a{d};   // narrowing rejected (preferred)
int b = d;  // silent narrowing (bad)
```

Example

Checked downcast (Type.2)

```
struct Base { virtual ~Base() = default; };
struct Der : Base { void f(); };

void use(Base& b) {
    if (auto* p = dynamic_cast<Der*>(&b)) {
        p->f(); // safe
    }
}
```

Never cast away const to modify (Type.3) undefined behaviour if original object is const. Legacy workaround: wrap, keep cast local, document.

Avoid Cstyle casts (Type.4). Use `static_cast` for explicit conversions, `dynamic_cast` for safe downcasts.

`static_cast` Best Practices

`static_cast` is for compiletime checked conversions (numeric, upcast, `void*` back to typed pointer). **Do not use for downcasts** (Type.2). For arithmetic narrowing, prefer braceinit or a checked helper.

Example

Valid `static_cast` uses

```
double d = 3.7;
int i = static_cast<int>(d); // explicit, but allows silent narrowing
// Prefer: int i{d}; (narrowing diagnosed)

Base* b = new Derived();
Derived* d = static_cast<Derived*>(b); // BAD (unchecked downcast)
// Good: auto* d = dynamic_cast<Derived*>(b);
```

`dynamic_cast` Safe Downcast

Requires polymorphic base (at least one virtual function). **Pointer form:** returns `nullptr` on failure check it. **Reference form:** throws `std::bad_cast` use only when failure is truly exceptional.

Design note: frequent downcasts are a design smell; prefer virtual functions, visitor, or `std::variant`.

Example

```
void process(Base* p) {
    if (auto* d = dynamic_cast<Derived*>(p)) {
        d->specific(); // safe
    } else {
        // handle other types
    }
}
```

Avoiding `reinterpret_cast`

The typesafety profile bans `reinterpret_cast`. It is the most dangerous cast: violates aliasing, alignment, and object lifetime rules.

Alternatives to typepunning

- **`std::bit_cast`** (C++20) safe bitwise copy between trivially copyable types of same size.
- **`std::memcpy`** portable fallback.
- **Explicit serialisation** read bytes, construct value.

Example

```
float u32_to_float(uint32_t u) {
    return std::bit_cast<float>(u); // defined behaviour
}
```

If you must use `reinterpret_cast` (rare):

- Localise it to a tiny FFI / lowlevel boundary.
- Document alignment, lifetime, and aliasing assumptions.
- Prefer readonly views; never write through reinterpreted pointers.
- Validate with static assertions and thorough tests.

Initialization & Sum Types

Type.5: Always initialise. `T{}` valueinitialises; avoids indeterminate values.
Type.6: Every member initialised; default member initialisers scale best.

Example

```
class Config {
    int port_{8080};            // default member initialiser
    std::string host_{"localhost"};
public:
    Config() = default;
    explicit Config(int port) : port_{port} {}
};
```

Type.7: `std::variant` replaces raw unions. Tracks active member, provides safe access (`get_if`, `visit`).

Example

```
using Value = std::variant<int, double, std::string>;
Value v = 3.14;
if (auto* p = std::get_if<double>(&v)) { /* use *p */ }
```

Note

Type Safety Checklist

1. Every variable and member initialised (Type.5, Type.6).
2. Narrowing conversions made explicit (Type.1).
3. Cstyle casts absent; named casts used where unavoidable (Type.4).
4. Downcasts use `dynamic_cast` with checked failure (Type.2).
5. `const` never cast away to modify (Type.3).
6. Sum types modeled with `std::variant`, not raw unions (Type.7).
7. Bit reinterpretation uses `std::bit_cast` or `memcpy`, not `reinterpret_cast`.

Part XIX

Object Lifetime Safety (Lifetime)

CHAPTER 102

Lifetime Safety – Condensed

The Lifetime profile enforces one rule: **never access an object through a pointer/reference/view that is not guaranteed to be valid.**

Lifetime.1: Dont dereference a possibly invalid pointer.

- Ensure every dereferenced pointer/reference/view is initialised, nonnull (if required), inbounds, not dangling, and not invalidated.
- Common invalid sources: uninitialised pointers, `nullptr`, address of a local that died, deleted object, or iterators/views invalidated by container mutation.

Lifetime.2: Dont create dangling pointers or references.

- Never store/return a pointer/reference that outlives the referredto object.
- Never bind a view (`std::string_view`, `std::span`) to a temporary; the view will dangle when the temporary dies.

Example

```
// BAD: returning pointer to local
int* bad() { int x=42; return &x; }

// BAD: view bound to temporary
std::string_view sv = "temp"s; // temporary std::string dies  dangle

// GOOD: return by value / own the data
std::string good() { return "hello"; }
std::string s = good();
std::string_view sv2 = s; // OK: s lives longer
```

Lifetime.3: Dont pass a potentially dangling pointer/reference to another function.

- If a callee might store the pointer beyond the call, pass an **owner** (`std::unique_ptr`, `std::shared_ptr`) or guarantee the object outlives the stored reference.
- For nonstoring use, pass `const T&`, `span`, or `string_view` with clear lifetime documentation.

Lifetime.4: Dont return a potentially dangling pointer/reference/view.

- Never return a reference/pointer to a local or temporary.
- Return views only when the underlying storage is guaranteed to outlive the returned view (e.g., a view into a parameter that the caller owns).
- Prefer returning by value or an owning smart pointer.

Lifetime.5: Avoid invalidation by mutation.

- Do not keep iterators, pointers, or references into a container across operations that may invalidate them (reallocation, insert, erase).
- Reacquire iterators after mutation, or use designs that dont require holding them across mutating calls.

Example

```
std::vector<int> v{1,2,3};
auto it = v.begin();
v.push_back(4);  // may reallocate  it invalid
*it = 0;         // undefined behaviour
```

Lifetime.6: Make ownership and lifetime explicit.

- Use RAII owners: `std::vector`, `std::string`, `std::unique_ptr`, `std::shared_ptr`.
- Raw pointers are nonowning observers; if you must use owning raw pointers (legacy), annotate with `gsl::owner<T*>`.
- Nonowning views: `T&`, `T*`, `std::span`, `std::string_view`. Document how long the borrowed object lives.

Dangling References

Common Causes and Fixes

- **Returning reference to local:** return by value.
- **Binding reference to temporary without lifetime extension:** temporary dies at end of fullexpression; use `const T&` local binding only when direct.
- **View (`string_view`/`span`) of temporary:** own the data first, then create view.
- **Container mutation invalidating references:** dont hold references across mutating ops.

Example

```
// Dangerous: reference through function return
const std::string& bad = id(std::string("tmp")); // dangles

// Safe: own the string
std::string s = "tmp";
const std::string& good = s;
```

Lifetime Extension

When a temporarys lifetime is extended: Only when directly bound to a local `const T&` or `T&&` in a valid initialization.

- `const std::string& r = std::string("hello");` // OK, lives until end of scope.
- Passing a temporary to a function parameter **does not** extend lifetime beyond the fullexpression.
- Returning a reference to a temporary **never** extends lifetime; it dangles.
- Views (`string_view`, `span`) **never** extend lifetime they are nonowning.
- Reference members initialized via `new` or certain aggregate forms may not extend lifetime as expected.

Temporaries

Temporaries are destroyed at the end of the fullexpression unless directly bound to a reference as above. Key traps:

- `std::string_view v = "hello"s;` temporary `std::string` destroyed at `;`.
- `const char* p = std::string("x").c_str();` `p` dangles after the statement.
- `return new Holder{ std::string("tmp") };` the temporary does not live as long as the dynamic object; reference member dangles.
- `std::initializer_list<int> il = {1,2,3}; const int* p = il.begin();` `p` points to temporary backing array (destroyed with `il`).

Prefer owning values instead of borrowing from temporaries.

Ownership Clarity

Make ownership visible in types.

- **Owner:** `std::unique_ptr`, `std::shared_ptr`, `std::vector`, `std::string`, value objects.
- **Borrower (nonowning):** `T&`, `T*`, `std::span`, `std::string_view`.
- **Avoid naked owning raw pointers.** If legacy, annotate with `gsl::owner<T*>`.

Interface rules

- **Pass/return ownership** via `unique_ptr` by value, or return by value.
- **Borrow** via `const T&`, `T*`, `span`, `string_view`; document that the object outlives the call.
- **Never store borrowed pointers/views** without a proven lifetime guarantee (e.g., parentchild relationship).

Example

```cpp
// Ownership transfer
std::unique_ptr<Widget> create();        // caller owns
void consume(std::unique_ptr<Widget> w); // takes ownership

// Borrowing
void print(const Config& cfg);           // nonnull, nonowning
void maybe_log(Logger* logger);          // nullable observer

// Nonowning range
void process(std::span<const int> arr);  // borrows
```

Note

Lifetime Safety Checklist

1. Every dereference proven safe (initialised, not null if required, in bounds, not dangling).
2. No pointers/references/views stored beyond the lifetime of the referent.
3. No views bound to temporaries; own the data first.
4. Iterators/pointers into containers reacquired after mutations.
5. Ownership expressed via RAII types; borrowing clearly indicated and temporary.
6. No owning raw pointers except annotated legacy boundaries.

Part XX

Bounds Safety (Bounds)

CHAPTER 103

Bounds Safety – Condensed

The **Bounds Safety** profile prevents outofbounds access by eliminating pointer arithmetic, arraytopointer decay, and unchecked indexing. The core rules (Bounds.1Bounds.4) enforce explicit range representation and safe access.

Core Rules

Bounds.1: Dont use pointer arithmetic. Use `span` instead. Pointer arithmetic turns a pointer into an implicit range without bounds. A `span` carries both address and size, enabling safe iteration and tool enforcement.

Bounds.2: Only index into arrays using constant expressions. Dynamic indexing into raw arrays is a major source of outofbounds bugs. For variable indices, use safer abstractions (`span`, `vector` with `at()`, or validated access).

Bounds.3: No arraytopointer decay. Implicit decay loses size information. Pass arrays as `span`, `std::array`, or a container to preserve the bound.

Bounds.4: Dont use standardlibrary functions and types that are not boundschecked. Prefer checked access (`at()`, `gsl::at`) over unchecked `operator[]`, especially at module boundaries, parsing, and securitysensitive code.

OutofBounds Access

Common Causes

- Offbyone in loop bounds.
- Unchecked index from user input or computations.
- Pointer arithmetic walking past the object.
- Signed/unsigned conversions (negative index becoming huge unsigned).
- Arraytopointer decay followed by a mismatched length.

Prevention Patterns

- **Rangefor** and standard algorithms no indexing, no bounds risk.
- **Validationonce then safe indexing:** check `i < size()` before using `operator[]`.
- **Checked access:** `at()` member functions (throws `std::out_of_range`).
- **Use `span`** to keep pointer and size together; wrap legacy `T* + size` pairs at boundaries.

Example

```
// Bad: pointer arithmetic + separate length
void fill_bad(int* p, int n) {
    for (int* it = p; it != p+n; ++it) *it = 0;
}

// Good: span carries bounds
void fill(std::span<int> s) {
    for (int& x : s) x = 0;
}
```

`std::span` The Safe Range View

`std::span<T>` is a **nonowning** view over contiguous elements. It stores a pointer and size (or uses compiletime extent).

- Replaces pointer+length arguments.
- Accepts C arrays, `std::array`, `std::vector`, etc.
- Supports slicing (`first`, `last`, `subspan`) without pointer arithmetic.
- Lifetime must be managed: a span dangles if the underlying storage is destroyed.

Example

```
void process(std::span<const int> data) {
    if (data.size() < 2) return;
    auto head = data.first(2);
    auto tail = data.subspan(2);
    // ...
}
```

Access and checking

- `span::operator[]` is **unchecked** (undefined behaviour if index out of range) in C++20/23.
- C++26 introduces `span::at()` boundschecked access.
- Use `gsl::at(span, index)` or `gsl::at(container, index)` for uniform checked access in projects adopting GSL.

Byte views

- `std::as_bytes(span<T>)` `span<const std::byte>`
- `std::as_writable_bytes(span<T>)` `span<std::byte>` (T nonconst)
- Useful for serialisation, hashing, and lowlevel operations.

Safe Indexing

Strategies in order of preference

1. **Eliminate indexing** use rangefor, algorithms.
2. **Checked access** `at()` or `gsl::at` (throws on invalid index).
3. **Validate once, then index** check `i < size()` early, then use `operator[]` safely.

Example

```cpp
// Checked access via at()
int read_checked(const std::vector<int>& v, size_t i) {
    return v.at(i); // throws std::out_of_range
}

// Validate once, then safe
int read_validated(std::span<const int> s, size_t i) {
    if (i >= s.size()) return 0; // handle error
    return s[i]; // safe: i in [0, size())
}
```

Index type safety

- Avoid signed/unsigned pitfalls: validate negative indices before converting to `size_t`.
- Use `std::ssize()` (C++20) for signed size, especially in reverse loops.

Tooling and Enforcement

Key diagnostics (MSVC C++ Core Guidelines checkers)

- C26481 Dont use pointer arithmetic. Use span instead (bounds.1).
- C26482 Only index into arrays using constant expressions (bounds.2).
- C26485 No arraytopointer decay (bounds.3).
- C26446 Prefer gsl::at() or unchecked subscript with bounds checking (bounds.4).

Clangtidy also provides checks aligned with Bounds.1Bounds.4.

Suppression (use sparingly)

```
[[gsl::suppress("bounds.1", justification: "legacy C ABI requires pointer arithmetic")]]
void legacy_bridge(char* p, int n);
```

Note

Bounds Safety Checklist

1. Pass contiguous ranges as `span`, not pointer+length.
2. No pointer arithmetic in application code; isolate in lowlevel wrappers.
3. No arraytopointer decay at interfaces; use `span` or containers.
4. Prefer rangefor and algorithms; eliminate manual indexing where possible.
5. When indexing, use checked access (`at()`) or explicit validation.
6. Wrap Cstyle buffers into `span` at boundaries.
7. Enable static analysis rules for Bounds profile.

Part XXI

GSL (Guidelines Support Library)

CHAPTER 104

GSL Guidelines (Condensed)

GSL.1: Use GSL vocabulary types to make safety assumptions explicit. Express intent in types, not comments: `not_null` for nonnullness, `span` for contiguous ranges, `owner` for owning raw pointers. Prefer RAII; annotate only when necessary.

GSL.2: Use `gsl::not_null` when null is invalid. Eliminates redundant null checks and documents contracts. Use for parameters, returns, and members that must never be null.

GSL.3: Use `gsl::span` for contiguous sequences. Replaces pointer+length pairs. Carries bounds; supports slicing (`first`, `subspan`) instead of pointer arithmetic. Use `gsl::span` where bounds checking is desired; `std::span` for standard portability.

GSL.4: Mark ownership explicitly. Prefer `std::unique_ptr` and RAII. Use `gsl::owner<T*>` only when you cannot upgrade (legacy, ABI). Annotate owning raw pointers to make ownership visible and toolcheckable.

GSL.5: Use `Expects` and `Ensures` for pre and postconditions. Document interface assumptions that types cannot express. `Expects(expr)` for preconditions, `Ensures(expr)` for postconditions.

GSL.6: Use GSL utilities and tooling thoughtfully.

- `finally` RAII cleanup action.
- `narrow<T>` (checked) / `narrow_cast<T>` (explicit) for safe narrowing.
- `gsl::at` checked element access (Bounds.4).
- `gsl::index` signed index type.
- Suppress warnings locally with `[[gsl::suppress("rule")]]` and a justification only when a legacy constraint cannot be lifted.

Note

GSL Adoption Checklist

1. Replace pointer+length arguments with `gsl::span` (GSL.3).
2. Use `not_null` for nonnull pointers (GSL.2).
3. Prefer RAII owners; use `owner<T*>` only in legacy interfaces (GSL.4).
4. Add `Expects`/`Ensures` for remaining contracts (GSL.5).
5. Integrate static analysis; suppress only with justification (GSL.6).

CHAPTER 105

span (GSL)

Definition: `gsl::span<T>` is a nonowning view of a contiguous sequence. It stores a pointer and size; does not allocate or free. **Purpose:** Replace pointer+length pairs and pointer arithmetic; make ranges explicit for bounds safety.

gsl::span vs std::span

- `std::span` (C++20) standard, unchecked `operator[]`.
- `gsl::span` Core Guidelines focus; often boundschecked in implementations, recommended by tooling (Bounds.1).

Use `gsl::span` where the Bounds profile is enforced; `std::span` otherwise.

Construction

- `gsl::span<T>(ptr, count)` wrap a raw pointer+length.
- From C arrays, `std::array`, contiguous containers (`vector::data()`, `size()`).
- Subspans via `first(n)`, `last(n)`, `subspan(offset, count)` avoids arithmetic.

Bounds Safety

- `gsl::at(span, i)` checked access (recommended by Bounds.4).
- Validate once, then index safely under invariant.
- Rangefor and algorithms remove indexing risk entirely.

Byte Views Convert to bytes for hashing/serialisation:

```
auto bytes = gsl::as_bytes(span_of_ints);   // span<const byte>
auto wbytes = gsl::as_writable_bytes(span); // span<byte>
```

Lifetime: `span` never extends lifetime; must not outlive the underlying storage. Dangling spans are a common pitfall.

CHAPTER 106

not_null (GSL)

Definition: `gsl::not_null<T>` wraps a pointerlike value and guarantees it is not null. Construction fails fast if given null. No runtime overhead for access.
Purpose: Express nonnull contracts in interfaces; eliminate defensive null checks.

When to Use

- Function parameters that require a valid object pointer.
- Return values that must never be null.
- Data members that must always refer to an object.

Do not use when null is a legitimate state; use `T*` or `std::optional` instead.

not_null vs references

- References are nonnull but nonrebindable.
- `not_null<T*>` allows rebinding (unless `const`), while still being nonnull.

Patterns

```
void log(gsl::not_null<Logger*> lg, const std::string& msg); // nonnull param

class Controller {
    gsl::not_null<Engine*> eng_; // member, never null
public:
    explicit Controller(gsl::not_null<Engine*> e) : eng_(e) {}
};
```

Tooling: Static analysis can recommend `not_null` when a pointer is never checked for null, and warn on redundant null checks. (MSVC C26429)

CHAPTER 107

`owner` (GSL)

Definition: `gsl::owner<T*>` is an annotation for **owning raw pointers**. It marks that the pointer must be released (`delete`) or transferred to another owner. It does **not** automatically delete. **Purpose:** Make ownership explicit when RAII owners cannot be used (legacy, ABI, handle internals).

Core Rules

- Plain `T*` is nonowning by default.
- Owning raw pointers must be annotated `owner<T*>`.
- Prefer `std::unique_ptr` / `std::shared_ptr` in new code.
- `owner<T*>` may be null; combine with design rules if nonnull is required.

Usage Patterns

```
gsl::owner<X*> make_x() { return new X{42}; } // ownership out
void consume(gsl::owner<X*> p) { delete p; }  // ownership in

// Inside a resource handle:
class XHandle {
    gsl::owner<int*> p_ = nullptr;
public:
    ~XHandle() { delete p_; }
    // ...
};
```

Static Analysis (MSVC Core Check)

- **C26400** assign `new` to `owner<T*>` (not plain `T*`).
- **C26401** do not delete a nonowner.
- **C26406** do not assign a raw pointer to an owner without a clear transfer.

clangtidy: `cppcoreguidelines-owning-memory` enforces similar rules.

Note

owner Checklist

1. Replace owning raw pointers with `unique_ptr` / `shared_ptr` where possible.
2. Annotate unavoidable owning raw pointers with `owner<T*>`.
3. Never delete a pointer unless it is marked `owner`.
4. Do not assign a raw pointer to an `owner` without a proven ownership transfer.

CHAPTER 108

index / Expects / Ensures (GSL)

gsl::index – Safe Index Type

Definition: `gsl::index` is the GSL vocabulary type for indexing, typically `std::ptrdiff_t`. It is a signed integer, avoiding many signed/unsigned mismatch bugs.

Why Signed?

- Prevents negative index converting to huge unsigned value.
- Enables clean reverse loops (`i >= 0`) without underflow.
- Supports not found sentinel values like `-1`.

When to Use

- Loop counters that index into containers.
- Return values for not found (negative sentinel).
- Reverse loops and arithmetic where negative values are meaningful.
- In performancesensitive indexing still prefer range`for` or algorithms when possible.

Example

Safe reverse loop with `gsl::index`

```cpp
#include <gsl/util>
#include <vector>

int last_positive(const std::vector<int>& v) {
    if (v.empty()) return -1;
    for (gsl::index i = static_cast<gsl::index>(v.size())-1; i >= 0; --i) {
        if (v[static_cast<std::size_t>(i)] > 0) return static_cast<int>(i);
    }
    return -1;
}
```

Safe Conversion from `size_t`

- Convert `v.size()` to `gsl::index` once, explicitly: `auto n = static_cast<gsl::index>(v.size());`
- Validate negative external indices *before* converting to `size_t` for subscript.
- Use `std::ssize()` (C++20) as a standard alternative for signed size.

Note

`gsl::index` Checklist

1. Use `gsl::index` for all index variables and indexreturning functions.
2. Keep conversions from `size_t` explicit and local.
3. Eliminate indexing entirely with range`for` and algorithms where possible.
4. Validate negative user indices before conversion.

`gsl::Expects` / `gsl::Ensures` – Contract Assertions

Purpose: Document and enforce preconditions (`Expects`) and postconditions (`Ensures`). **Behaviour:** Violation `std::terminate` (failfast). Not for recoverable runtime errors.

Usage

- Place `Expects(condition)` at the top of the function body.
- Place `Ensures(condition)` before normal return statements (often at end).
- Keep conditions sideeffect free, fast, and deterministic.
- Prefer expressing contracts with types (`not_null`, `span`) first; use `Expects/Ensures` for the rest.

Example

```
#include <gsl/assert>
#include <gsl/span>

int max_val(gsl::span<const int> a) {
    Expects(!a.empty());          // precondition: range not empty

    int m = a[0];
    for (int x : a) if (x > m) m = x;

    Ensures(m >= a[0]);          // postcondition: max is at least first element
    return m;
}
```

What Not to Use Them For

- Do not use for validating external input (file/network/user) use explicit error handling.
- Do not put side effects inside contract checks.
- Do not rely on contract checks for normal program logic treat them as fatal debugging aids.

Implementation Note Microsofts GSL reference implementation always terminates on violation (no configurable throw). Use a custom `std::terminate_handler` for logging before exit.

Note

Contract Checklist

1. Use `Expects` for preconditions that must hold for safe execution.
2. Use `Ensures` for guarantees that the function provides on normal return.
3. Keep conditions simple, pure, and cheap.
4. Prefer stronger types over contract checks where possible.
5. Do not use for recoverable errors separate validation from contracts.

Part XXII

Profiles

CHAPTER 109

Type Safety Profile (Pro.safety)

Goal: Eliminate type violations unsafe casts, uninitialised use, raw unions, and varargs via toolenforceable rules.

Type.1: Avoid casts.

- **Type.1.1** No `reinterpret_cast`.
- **Type.1.2** No `static_cast` for arithmetic narrowing (use `T{e}` or GSL helpers).
- **Type.1.3** No pointer casts where source == target.
- **Type.1.4** No pointer casts that could be implicit.

Type.2: Dont use `static_cast` to downcast. Use `dynamic_cast` when hierarchy navigation is unavoidable; otherwise redesign.

Type.3: Dont use `const_cast` to cast away `const`. Modifying an originally `const` object is undefined behaviour. Fix the API or isolate in a wrapper.

Type.4: Dont use Cstyle or functional casts. Prefer construction (braces) or named casts (`static_cast`, `dynamic_cast`, etc.).

Type.5: Dont use a variable before it has been initialised. Always initialise; prefer `T{}`.

Type.6: Always initialise a data member. Use default member initialisers or constructor initialiser lists.

Type.7: Avoid naked `union`; use `std::variant` instead. Variant tracks the active member and provides safe access.

Type.8: Avoid varargs. Use variadic templates, overloads, or typesafe formatting.

Example

Before (Type.1): `auto q = reinterpret_cast<const uint32_t*>(p);` **After:** `uint32_t x; std::memcpy(&x, p, sizeof(x));`

Tool enforcement: ClangTidy `cppcoreguidelines-pro-type-*` checks; MSVC CppCoreCheck diagnostics.

CHAPTER 110

Bounds Safety Profile (Pro.bounds)

Goal: Prevent outofbounds access by removing pointer arithmetic, arraytopointer decay, and unchecked indexing.

Bounds.1: Dont use pointer arithmetic. Use `span` instead. Carry pointer + size together; iterate with range`for` or algorithms.

Bounds.2: Only index into arrays using constant expressions. Dynamic indexing on raw arrays is disallowed; use `span`, `at()`, or validated access.

Bounds.3: No arraytopointer decay. Pass arrays as `span`, `std::array`, or containers never let size be lost.

Bounds.4: Dont use unchecked standardlibrary access. Prefer `at()` or `gsl::at()` for checked access; eliminate indexing with algorithms.

Example

Before (Bounds.1): `void f(int* p, int n){ /* p+i */ }` **After:** `void f(gsl::span<int> s){ for(int& x:s) ... }`

Tool enforcement: ClangTidy `cppcoreguidelines-pro-bounds-*` checks; MSVC C26481 (no pointer arithmetic), C26482, C26485.

CHAPTER 111

Lifetime Safety Profile (Pro.lifetime)

Goal: Never dereference a possibly invalid pointer (dangling, useafterfree, invalidated iterator, view of temporary).

Lifetime.1: Dont dereference a possibly invalid pointer detect or avoid. Focus on the most common local patterns:

- Pointer to local that escaped scope.
- Use after `delete` (fix: RAII).
- Iterator/reference invalidated by container mutation.
- `std::string_view` / `std::span` bound to a temporary.

Example

Bad: `int* p = new int(7); delete p; return *p;` **Good:** `auto p = std::make_unique<int>(7); return *p;`
Bad: `std::string_view sv = "temp"s; // dangles` **Good:** `std::string s = "temp"; std::string_view sv = s;`

Enforcement: Visual Studio lifetime analysis (local static detection); ASan/UBSan for runtime useafterfree/scoping bugs. Prefer RAII, avoid returning views of locals, and document lifetime contracts.

CHAPTER 112

Enforcement and Tooling (Profiles)

Profiles are designed for **tool enforcement**. Combine multiple layers:

Layer 1 Compiler Warnings: Elevate important warnings to errors; treat them as part of a clean build.
Layer 2 Static Analysis: ClangTidy (`cppcoreguidelines-pro-type-*`, `pro-bounds-*`) and MSVC CppCoreCheck map directly to profiles.
Layer 3 Dynamic Analysis: Sanitizers (AddressSanitizer, UndefinedBehaviorSanitizer, ThreadSanitizer) catch runtime violations during testing.
Layer 4 Code Review: Verify ownership, invariants, and lifetime contracts that tools cannot prove.

Adoption Roadmap

1. **Baseline:** Run tools, fix highimpact bugs (bounds, lifetime).
2. **Prevent Regressions:** CI blocks new profile violations.
3. **Refactor:** Use `span`, `not_null`, RAII, checked access.
4. **Isolate Unsafe Code:** Keep lowlevel operations in small, audited modules with safe wrappers.

Key Tool Mapping

- **Type Safety:** ClangTidy `cppcoreguidelines-pro-type-*`, MSVC CppCoreCheck (casts, init).
- **Bounds:** ClangTidy `cppcoreguidelines-pro-bounds-*`, MSVC C26481C26485.
- **Lifetime:** Visual Studio lifetime analysis, ASan, UBSan.

Note

Golden Rule: Treat profile violations as defects. Use local suppressions with justification only when a legacy constraint cannot be lifted.

Part XXIII

Enforcement and Tooling

CHAPTER 113

Tooling Summary

`clang-tidy` – Core Guidelines Linter

`clang-tidy` is the primary crossplatform tool for enforcing the Core Guidelines. It requires a `compile_commands.json` (generated by CMake with `CMAKE_EXPORT_COMPILE_COMMANDS=ON`).

Key Commands

- `clang-tidy -p build file.cpp` – run with compilation database.
- `-checks='-*,cppcoreguidelines-pro-type-*, cppcoreguidelines-pro-bounds-*'` – enable specific profiles.
- `-fix` – apply automatic fixes; `-export-fixes=fixes.yaml` for controlled application.
- `-warnings-as-errors='...'` – treat selected checks as errors.
- `run-clang-tidy.py -p build -j 8` – parallel fullproject run.
- `clang-tidy-diff.py` – filter diagnostics to changed lines (PR gate).

Configuration (`.clang-tidy`)

```
Checks: '-*,cppcoreguidelines-pro-type-*, cppcoreguidelines-pro-bounds-*,cppcoreguidelines-*'
WarningsAsErrors: ''
HeaderFilterRegex: '^(src|include)/'
FormatStyle: file
```

Suppression Use `// NOLINT(cppcoreguidelines-...)` on the offending line, or `// NOLINTNEXTLINE`. Always add a justification comment.

Static Analysis Layers

Layered approach:

1. **Compiler warnings** – fast, strict baseline (treat warnings as errors in CI).
2. **Rulebased linting** (`clang-tidy`) – enforces profiles, modernisation, and bugprone patterns.
3. **Pathsensitive analysis** (Clang Static Analyzer, GCC `-fanalyzer`) – finds leaks, null dereferences, useafterfree (soundness not guaranteed).
4. **Security scanning** (CodeQL) – querybased detection of vulnerabilities.

Clang Static Analyzer: `scan-build cmake -build build` runs wholeproject checks. **GCC `-fanalyzer`:** `gcc -fanalyzer file.c` (primarily C; expensive, not complete). **MSVC Code Analysis:** Set `<RunCodeAnalysis>true</RunCodeAnalysis>` and `<EnableCppCoreCheck>true</EnableCppCoreCheck>` in MSBuild.

CI Integration

Golden Rule: Generate `compile_commands.json` once and use it for all analysis tools. **Policy:** *No new warnings* – establish a baseline, then block regressions.

Recommended CI Job Matrix

- **PR gate (fast):** build + tests + `clang-tidy` on changed lines.
- **Nightly / release:** full `clang-tidy` + sanitizers (ASan, UBSan) + optional CodeQL.
- **Windows:** MSVC build + MSVC Code Analysis (CppCoreCheck).

Example

Sanitizer Build (CMake)

```
cmake -S . -B build-asan -DCMAKE_BUILD_TYPE=RelWithDebInfo \
  -DCMAKE_CXX_FLAGS="-fsanitize=address,undefined -fno-omit-frame-pointer"
cmake --build build-asan
ctest --test-dir build-asan
```

Artifacts: store build logs, test results, sanitizer reports, and SARIF output (for code scanning).

Gradual Rule Enforcement

Adoption Strategy

1. **Baseline:** run tools, collect existing findings, fix critical safety bugs.
2. **Stop regressions:** block new warnings on changed code (diffbased analysis).
3. **Expand rules:** start with `Pro.bounds` and `Pro.safety`; later add lifetime, modernise, etc.
4. **Full enforcement:** require clean profile runs for release; keep suppressions rare and justified.

Suppression Policy

- Prefer code fixes over suppressions.
- Suppress only at the narrowest scope (`NOLINTNEXTLINE(specific-check)`).
- Always provide a short reason comment.
- Exclude thirdparty headers via `HeaderFilterRegex`.

Note

Enforcement Checklist

1. Generate and use `compile_commands.json` for all analysis.
2. Configure `.clang-tidy` with the safety profiles first.
3. Run `clang-tidy` in CI (PR diff gate + nightly full run).
4. Add sanitizer builds (ASan/UBSan) to CI tests.
5. Establish no new warnings policy; continuously shrink baseline.
6. Keep suppressions minimal, documented, and timebound.

Appendices

A. Complete Rule Index

The Core Guidelines are organized into families. The full official titles are in the canonical document; this summary lists the rule ranges by family for quick navigation.

Rule Families and Ranges

- **Bounds Safety:** Bounds.1–4
- **Constants/Immutability:** Con.1–5
- **Classes:** C.1–36
- **Cstyle Programming:** CPL.1–6
- **Concurrency/Parallelism:** CP.1–15
- **Error Handling:** E.1–14
- **Enumerations:** Enum.1–8
- **Expressions/Statements:** ES.1–34
- **Functions:** F.1–60
- **Interfaces:** I.1–30
- **Philosophy:** P.1–13
- **Performance:** Per.1–21
- **Resource Management:** R.1–17
- **Standard Library:** SL.1–10
- **Templates/Generic Prog.:** T.1–10
- **Source Files:** SF.1–10
- **Profiles:** Pro.bounds, Pro.lifetime, Pro.type
- **Type Safety:** Type.1–8
- **Lifetime Safety:** Lifetime.1–4

B. Rule Classification by Risk

Risk Levels

- **Critical** – Undefined behaviour, memory corruption, data races (e.g., Bounds, Type, Lifetime, CP.2).
- **High** – Leaks, crashes, ambiguous ownership (e.g., R.*, E.*, some I.*).
- **Medium** – Interface misuse, maintainability, some expression traps.
- **Low** – Readability, style consistency.
- **Contextdependent** – Performance rules, domainspecific constraints.

Adoption Priority (RiskDriven)

1. Bounds, Lifetime, Type safety
2. Concurrency (datarace freedom)
3. Resource management, Error handling
4. Interfaces, Classes, Functions, Expressions
5. Library and performance guidance

C. Comparison with MISRA and AUTOSAR

HighLevel Differences

- **Core Guidelines** – Generalpurpose modern C++, profiles for enforceable safety.
- **MISRA C++:2023** – Highintegrity subset for C++17, processdriven compliance.
- **AUTOSAR C++14** – Automotive safety, strict subset of C++14, updated MISRA 2008.

Common Ground All three emphasise: avoid undefined behaviour, clear ownership, minimal global mutable state, restricted casts/unions, and deterministic analyzability.

Key Differences

- **Profiles vs. compliance:** Core Guidelines use toolenforceable profiles; MISRA/AUTOSAR require formal deviation handling.
- **Language baseline:** Core Guidelines evolve with modern C++; MISRA 2023 targets C++17, AUTOSAR C++14.
- **Exceptions and RTTI:** AUTOSAR restricts `dynamic_cast` and uncaught exceptions; Core Guidelines prefer safe design but are less prescriptive.

Integration Strategy

1. Follow the mandated standard (MISRA/AUTOSAR) where compliance is required.
2. Add Core Guidelines **profiles** (bounds, lifetime, type) for measurable safety guarantees.
3. Document deviations; keep them small, reviewed, and timebound.

D. When and Why to Break a Rule

Principles

- Rules are defaults; exceptions require a higherpriority constraint (correctness, ABI, performance).
- Make exceptions explicit, local, and reviewable.
- Safetyprofile violations need formal justification and strong containment.

Decision Checklist (Mandatory for Every Exception)

1. Which rule is being broken?
2. What is the constraint (ABI, realtime, legacy)?
3. Risk of violation (critical/high/medium)?
4. Conforming alternative and why it was rejected.
5. How is the exception contained (narrow scope, wrapper)?
6. Verification plan (tests, sanitizers, static analysis).
7. Removal plan / revisit date.

Safe Exception Patterns

- Isolate lowlevel code behind a safe, ruleconforming API.
- Use `[[suppress("...")]]` for profile checks only when absolutely necessary, with a justification.
- Add compensating guardrails: assertions, bounds checks, fuzz tests.

Documentation Template Rule ID(s) – Location – Reason – Risk – Rejected alternative – Containment – Verification – Removal plan – Reviewer.

E. RealWorld Industrial Examples

1. Embedded Buffer Parsing *Problem:* Raw pointer arithmetic on device frames.
Solution: Use `std::span` and explicit bounds checks.
Sketch: `std::optional<Frame> parse(span<const uint8_t>);`

2. Hidden Data Races in a Library *Problem:* Functionlocal static cache causing races.
Solution: Move cache into a dedicated object; caller controls sharing.
Sketch: `struct ComputationCache { ... double compute(int x); };`

3. Ownership at ABI Boundaries *Problem:* Raw pointers crossing C plugin boundaries.
Solution: RAII wrapper with explicit create/destroy, `owner<T*>` annotation.
Sketch: `class PluginHandle { unique_ptr<...>; };`

4. Smart Pointer Parameters *Problem:* Misusing `shared_ptr` for observation.
Solution: `T&` for borrowing, `unique_ptr` for transfer.
Sketch: `void use(Widget& w); void sink(unique_ptr<Widget>);`

5. Virtual Destructors *Problem:* Deleting derived object via base without virtual destructor.
Solution: `virtual Base() = default;` (or protected nonvirtual if never deleted through base).

6. NonNull Contracts *Problem:* Defensive null checks everywhere.
Solution: Use a `not_null<T>` wrapper; enforce at construction.
Sketch: `int read_value(not_null<const Node*> n);`

7. Deliberate LowLevel Escape Hatch *Problem:* Hot path must use pointer arithmetic, cannot be refactored immediately.
Solution: Isolate in a small, tested function; suppress profile checks with justification.
Sketch: `[[suppress("bounds")]] char* raw_find(char* p, int n, char x);`

Industrial Adoption Playbook

1. Start with one profile (Bounds, Lifetime, or Type) and keep it warningfree.
2. Isolate unavoidable lowlevel code behind safe interfaces.
3. Clarify ownership at boundaries (ABI, plugins) with RAII or `owner<T*>` annotation.
4. Harden concurrency by eliminating shared mutable state.

References

This book draws on the following sources:

Primary Rule Source

- **C++ Core Guidelines** Bjarne Stroustrup, Herb Sutter, and contributors. The canonical guideline text (snapshot used in this book).

ISO/IEC C++ Standards

- ISO/IEC 14882:2014 (C++14)
- ISO/IEC 14882:2017 (C++17)
- ISO/IEC 14882:2020 (C++20)
- ISO/IEC 14882:2024 (C++23)

SafetyCritical Coding Standards

- **MISRA C++:2023** Motor Industry Software Reliability Association.
- **AUTOSAR C++14** AUTOSAR Adaptive Platform, Guidelines for C++14 in critical and safetyrelated systems.

Guidelines Support Library (GSL)

- Microsoft GSL repository implementation of `not_null`, `owner`, `span`, `Expects/Ensures`, etc.

Tooling Documentation

- **Microsoft C++ Core Guidelines Checkers** Visual Studio / MSVC CppCoreCheck.
- **clangtidy** LLVM extra tools; `cppcoreguidelines-*` check families.

Standards Status and Community Sources

- **WG21 (ISO C++ Committee)** official standards status page.
- **Standard C++ Foundation (isocpp.org)** *The Standard* page.

All references are publicly available and correspond to the versions used for rule citations and tooling discussions throughout this book.