https://simplifycpp.org

DRAFT

Inside the ISO C++ Core Guidelines

An Engineer's Reference and Companion

Prepared by Ayman alheraki

First Edition

Inside the ISO C++ Core Guidelines

An Engineer's Reference and Companion

Prepared by Ayman Alheraki

simplifycpp.org

December 2025

Contents

Contents **2**

Author's Introduction **164**

Book's Introduction 169

What Are the ISO C++ Core Guidelines? **169**

Objectives of This Reference **176**

How to Use This Book as a Reference **182**

Guidelines vs. Mandatory Rules **191**

When and Why a Rule May Be Broken **199**

I Philosophy (P) 208

1 P.1 – P.12 **210**

1.1 Purpose of the Philosophy Rules 210

1.2 How to Read P-Rules in Practice 210

1.2.1 They Are Principles, Not Micro-Rules 210

1.2.2 They Lead to Checkable Code . . . 211
1.3 P.1: Express Ideas Directly in Code . . . 211
1.3.1 Rule . . . 211
1.3.2 Reason . . . 211
1.3.3 Example: Make Meaning Explicit . . . 211
1.3.4 Prefer Library Expressions of Intent . . . 211
1.3.5 Enforcement Notes . . . 212
1.4 P.2: Write in ISO Standard C++ . . . 212
1.4.1 Rule . . . 212
1.4.2 Reason . . . 212
1.4.3 Guidance: Localize Extensions . . . 212
1.4.4 Enforcement Notes . . . 213
1.5 P.3: Express Intent . . . 213
1.5.1 Rule . . . 213
1.5.2 Reason . . . 213
1.5.3 Example: Prefer Range-Based Loops . . . 213
1.5.4 Enforcement Notes . . . 213
1.6 P.4: Ideally, a Program Should Be Statically Type Safe . . . 214
1.6.1 Rule . . . 214
1.6.2 Reason . . . 214
1.6.3 Preferred Alternatives (Examples) . . . 214
1.7 P.5: Prefer Compile-Time Checking to Run-Time Checking . . . 215
1.7.1 Rule . . . 215
1.7.2 Reason . . . 215
1.7.3 Example: Replace a Run-Time Check with `static_assert` . . . 215
1.7.4 Example: Prefer Interfaces the Compiler Can Help With . . . 215
1.7.5 Enforcement Notes . . . 216

1.8 P.6: What Cannot Be Checked at Compile Time Should Be Checkable at Run Time . . . 216
1.8.1 Rule . . . 216
1.8.2 Reason . . . 216
1.8.3 Example: Do Not Hide the Size Information . . . 216
1.8.4 Better: Pass a Size-Carrying Type or a View . . . 217
1.8.5 Enforcement Notes . . . 217
1.9 P.7: Catch Run-Time Errors Early . . . 217
1.9.1 Rule . . . 217
1.9.2 Reason . . . 217
1.9.3 Example: Prefer Early, Central Validation . . . 218
1.9.4 Performance Note . . . 218
1.10 P.8: Don't Leak Any Resources . . . 218
1.10.1 Rule . . . 218
1.10.2 Reason . . . 218
1.10.3 Example: Prefer RAII . . . 218
1.10.4 Important Clarification from the Guidelines . . . 219
1.10.5 Enforcement Notes . . . 219
1.11 P.9: Don't Waste Time or Space . . . 219
1.11.1 Rule . . . 219
1.11.2 Reason . . . 220
1.11.3 Example: Avoid Gratuitous Work . . . 220
1.11.4 Enforcement Notes . . . 220
1.12 P.10: Prefer Immutable Data to Mutable Data . . . 220
1.12.1 Rule . . . 220
1.12.2 Reason . . . 220
1.12.3 Practical Guidance . . . 220

1.13 P.11: Encapsulate Messy Constructs, Rather Than Spreading Through the Code 221
1.13.1 Rule . . . 221
1.13.2 Reason . . . 221
1.13.3 Example: Replace Manual Allocation Logic with `std::vector` . . . 221
1.13.4 Enforcement Notes . . . 222
1.14 P.12: Use Supporting Tools as Appropriate . . . 222
1.14.1 Rule . . . 222
1.14.2 Reason . . . 222
1.14.3 Example: Use Static Analysis Intentionally . . . 222
1.14.4 Caution . . . 223
1.15 A Short Summary Checklist for P.1–P.12 . . . 223

2 Design Philosophy of C++ 224
2.1 What This Chapter Covers . . . 224
2.2 Core Design Goals of C++ . . . 224
2.2.1 Direct, Efficient Use of Hardware . . . 224
2.2.2 Abstraction Without Losing Performance . . . 225
2.2.3 A Minimal Assumed Run-Time Environment . . . 225
2.2.4 Long-Term Evolution and Stability . . . 225
2.3 The Zero-Overhead Principle . . . 225
2.3.1 What It Means . . . 225
2.3.2 Why It Matters . . . 226
2.3.3 Example: Zero-Overhead Abstraction via Templates . . . 226
2.4 Deterministic Resource Management (RAII) . . . 226
2.4.1 Why RAII Is Central . . . 226
2.4.2 Example: Resource Safety by Construction . . . 227
2.5 Expressiveness with Strong Static Types . . . 227
2.5.1 Make Intent and Invariants Visible . . . 227

2.5.2 Example: Use Types to Express Intent . . . 227
2.6 Multi-Paradigm by Design . . . 228
2.6.1 Not One Style, but Controlled Choice . . . 228
2.6.2 Example: Algorithms Express Intent . . . 228
2.7 Prefer Compile-Time Guarantees When Possible . . . 229
2.7.1 Shift Errors Left . . . 229
2.7.2 Example: Enforce Assumptions with `static_assert` . . . 229
2.8 Safety as an Engineering Goal, Not a Separate Language . . . 229
2.8.1 C++ Improves Safety Through Defaults, Libraries, and Tooling . . . 229
2.8.2 Design Implication . . . 230
2.9 Encapsulation of Low-Level Complexity . . . 230
2.9.1 Abstractions Should Contain the Mess . . . 230
2.9.2 Example: Hide Raw Memory Behind a Type . . . 230
2.10 Summary: The C++ Design Philosophy in One Page . . . 231

3 Zero-overhead Abstraction 232
3.1 Definition and Meaning . . . 232
3.1.1 The Zero-overhead Principle . . . 232
3.1.2 What the Principle Is (and Is Not) . . . 232
3.2 Why Zero-overhead Matters in C++ . . . 233
3.2.1 C++ Targets Performance-Critical Domains . . . 233
3.2.2 The Goal: Abstraction Without Surprise Costs . . . 233
3.3 How C++ Achieves Zero-overhead Abstraction . . . 234
3.3.1 Compile-time Composition (Templates and Inlining) . . . 234
3.3.2 RAII: Safety With Deterministic Cost . . . 234
3.3.3 Value Semantics and Move Operations . . . 235
3.4 Where Costs Appear: Abstraction Is Not Magic . . . 235
3.4.1 Static vs. Dynamic Polymorphism . . . 235

3.4.2 Type Erasure Can Add Indirection 236
3.4.3 Checks and Safety Are Not Automatically “Waste” 236
3.5 Zero-overhead in Practice: Rules of Thumb 236
3.5.1 Prefer Abstractions the Compiler Can See Through 236
3.5.2 Measure, Then Decide 237
3.5.3 Encapsulate “Paying” Abstractions 237
3.6 Relationship to the Core Guidelines 237

4 Expressing Intent Clearly 238
4.1 What “Express Intent” Means 238
4.2 Why Clear Intent Is a Reliability Feature 239
4.3 Principles for Expressing Intent 239
4.3.1 Prefer Code That States Meaning Directly 239
4.3.2 Prefer Interfaces That Make Misuse Hard 239
4.3.3 Prefer Designs That Are Checkable 240
4.4 Techniques: How to Make Intent Visible in C++ 240
4.4.1 Technique 1: Use Names That Communicate Semantics 240
4.4.2 Technique 2: Make Units and Domain Meaning Part of the Type . . . 240
4.4.3 Technique 3: Use `const` to Express Non-Mutation 241
4.4.4 Technique 4: Use RAII to Express Ownership and Lifetime 241
4.4.5 Technique 5: Avoid Pointer Arithmetic; Prefer Size-Carrying Views . 242
4.4.6 Technique 6: Make Non-Null a Type-Level Requirement 242
4.4.7 Technique 7: Prefer Narrow, Scoped Enumerations Over Integers . . 243
4.4.8 Technique 8: Replace “Magic Numbers” with Named Constants . . . 243
4.4.9 Technique 9: Prefer Algorithms and Range-Based Loops to Manual Control Flow 244
4.4.10 Technique 10: Keep Scopes Small and Bind Names Where They Are Used 244

4.5 Intent and Tooling: Making the Design Enforceable 244
4.5.1 clang-tidy and the `cppcoreguidelines-*` Checks 245
4.5.2 Microsoft C++ Core Guidelines Checkers 245
4.6 A Practical Checklist for Code Review 245

5 Making Invalid States Unrepresentable 246
5.1 The Core Idea 246
5.2 Why This Matters in C++ 246
5.3 Principles: How to Make Invalid States Unrepresentable 247
5.3.1 Principle 1: Put Meaning in Types, Not Comments 247
5.3.2 Principle 2: Narrow Interfaces to the Valid Domain 247
5.3.3 Principle 3: Carry Required Context Together 247
5.4 Common C++ Patterns That Encode Validity 248
5.4.1 Pattern 1: Scoped Enumerations Instead of Integers 248
5.4.2 Pattern 2: Strong Types for Units and Domain Values 248
5.4.3 Pattern 3: Non-null by Construction (`gsl::not_null`) 249
5.4.4 Pattern 4: Size-Carrying Views (`span`) Instead of (Pointer, Length) . 249
5.4.5 Pattern 5: Optional Values (`std::optional`) Instead of Sentinel Values 250
5.4.6 Pattern 6: Closed Alternatives (`std::variant`) Instead of Unions or Ad-hoc Tags 250
5.4.7 Pattern 7: Expected-or-Error (`std::expected`) Instead of Unstructured Error Codes 251
5.5 Encoding Invariants Inside Types 252
5.5.1 Private Construction + Validated Factory 252
5.6 Tooling Support and Enforceability 253
5.7 Checklist: Are You Making Invalid States Unrepresentable? 253

6 Clarity versus Cleverness 255

6.1 What This Chapter Means by “Clarity” and “Cleverness” 255
6.1.1 Clarity 255
6.1.2 Cleverness 255
6.2 Why Clarity Wins in Professional C++ 256
6.2.1 Maintainability Scales, Cleverness Does Not 256
6.2.2 Clever Code Often Hides Undefined or Unspecified Behavior 256
6.3 Guideline Anchors: Where the Core Guidelines Stand 257
6.3.1 Prefer Simplicity and Readable Structure 257
6.3.2 Prefer Abstractions Over Handcrafted Tricks 257
6.3.3 Avoid Macro Cleverness 257
6.4 Common “Clever” Patterns and Clear Alternatives 257
6.4.1 Pattern 1: Complicated Expressions 257
6.4.2 Pattern 2: Precedence Tricks 258
6.4.3 Pattern 3: Undefined or Unspecified Order of Evaluation 259
6.4.4 Pattern 4: Clever Loops That Hide Intent 260
6.4.5 Pattern 5: “Clever” Use of Loop Indices 261
6.4.6 Pattern 6: Macro Cleverness 261
6.5 When “Clever” Is Acceptable 262
6.5.1 When It Is Local, Measured, and Documented 262
6.5.2 A Good Rule of Thumb 262
6.6 Checklist: Enforcing Clarity in Daily Work 262

7 Engineering Simplicity 264
7.1 What “Engineering Simplicity” Means 264
7.2 Why Simplicity Is a Professional Requirement 264
7.2.1 Software Lives Longer Than Its Authors 265
7.2.2 Simplicity Reduces Defect Density 265
7.2.3 Simplicity Is Compatible with Performance 265

7.3 Guideline Anchors for Simplicity . . . 265
7.3.1 Express the Idea, Not the Mechanism . . . 266
7.3.2 Encapsulate Messy Constructs . . . 266
7.3.3 Prefer Suitable Library Abstractions . . . 266
7.4 Principles of Engineering Simplicity in C++ . . . 266
7.4.1 Principle 1: Make the Valid Path the Easy Path . . . 266
7.4.2 Principle 2: Keep Invariants Local . . . 266
7.4.3 Principle 3: Prefer Narrow Interfaces with Clear Contracts . . . 267
7.4.4 Principle 4: Separate Policy from Mechanism . . . 267
7.4.5 Principle 5: Use Tools to Maintain Simplicity . . . 267
7.5 Common Complexity Traps and Simpler Alternatives . . . 267
7.5.1 Trap 1: Pointer + Length Interfaces . . . 267
7.5.2 Trap 2: Hidden Ownership and Manual Cleanup . . . 268
7.5.3 Trap 3: Sentinel Values for “Not Found” or “No Value” . . . 269
7.5.4 Trap 4: Overly Clever Expressions . . . 269
7.6 Simplicity Patterns That Scale . . . 270
7.6.1 Pattern 1: Small, Single-Purpose Functions . . . 270
7.6.2 Pattern 2: Convert at the Boundary . . . 270
7.6.3 Pattern 3: Prefer Strong Types for Domain Values . . . 270
7.6.4 Pattern 4: Prefer a Standard Algorithm When It Matches the Intent . . . 270
7.6.5 Pattern 5: Make “Fast” Code Simple by Design . . . 271
7.7 A Practical Review Checklist . . . 271

II Interfaces (I) 273

8 I.1 – I.26 275
8.1 Rule Map (Current Numbering) . . . 275

8.2 Shared Design Goals for Interface Rules . . . 275
8.3 I.1: Make interfaces explicit . . . 276
8.4 I.2: Avoid non-const global variables . . . 277
8.5 I.3: Avoid singletons . . . 278
8.6 I.4: Make interfaces precisely and strongly typed . . . 278
8.7 I.5: State preconditions (if any) . . . 279
8.8 I.6: Prefer `Expects()` for expressing preconditions . . . 279
8.9 I.7: State postconditions . . . 280
8.10 I.8: Prefer `Ensures()` for expressing postconditions . . . 281
8.11 I.9: If an interface is a template, document its parameters using concepts . . . 281
8.12 I.10: Use exceptions to signal a failure to perform a required task . . . 282
8.13 I.11: Never transfer ownership by a raw pointer (`T*`) or reference (`T&`) . . . 283
8.14 I.12: Declare a pointer that must not be null as `not_null` . . . 284
8.15 I.13: Do not pass an array as a single pointer . . . 284
8.16 I.14 – I.21: Reserved / Not Present in Current Rule Set . . . 286
8.17 I.22: Avoid complex initialization of global objects . . . 286
8.18 I.23: Keep the number of function arguments low . . . 286
8.19 I.24: Avoid adjacent parameters that can be invoked by the same arguments in either order with different meaning . . . 287
8.20 I.25: Prefer empty abstract classes as interfaces to class hierarchies . . . 288
8.21 I.26: If you want a cross-compiler ABI, use a C-style subset . . . 289
8.22 Summary Checklist (Interfaces) . . . 290

9 Designing Clear Interfaces 291

9.1 Purpose of This Chapter . . . 291
9.2 What an “Interface” Is in C++ . . . 291
9.3 Design Goals of a Clear Interface . . . 292
9.4 A Practical Process for Designing an Interface . . . 292

9.4.1 Step 1: Identify the Meaning, Not the Mechanism . . . 292
9.4.2 Step 2: Encode Constraints in Types . . . 293
9.5 Parameters: Choosing the Right Shapes . . . 293
9.5.1 Prefer the Most Specific Acceptable Type (I.4) . . . 293
9.5.2 Avoid Adjacent Same-Type Parameters (I.24) . . . 294
9.5.3 Keep Parameter Counts Low (I.23) . . . 294
9.6 Bounds and Ranges: Avoid "Pointer-Only" Arrays (I.13) . . . 294
9.6.1 Use `std::span` to Carry Bounds . . . 294
9.6.2 Tool Support Reinforces This Choice . . . 295
9.7 Nullability and Ownership: Make Lifetime Rules Explicit . . . 295
9.7.1 Never Transfer Ownership with Raw Pointers (I.11) . . . 295
9.7.2 If Null Is Invalid, Say So in the Type (I.12) . . . 295
9.8 Preconditions and Postconditions (I.5–I.8) . . . 296
9.8.1 State Preconditions and Postconditions . . . 296
9.8.2 Contract Semantics Note . . . 297
9.9 Error Handling: Make Failure Explicit . . . 297
9.9.1 Exceptions for Failure to Perform a Required Task (I.10) . . . 297
9.9.2 Make "Must-Check" Results Hard to Ignore . . . 297
9.10 Read-only Views: Prefer `std::string_view` for Observation . . . 298
9.11 Generic Interfaces: Use Concepts to Document Requirements (I.9) . . . 298
9.12 Stability and ABI Boundaries (I.26) . . . 299
9.13 Tooling: Keep Interfaces Clear Over Time . . . 299
9.14 Interface Design Checklist . . . 299

10 Separation of Interface and Implementation 301
10.1 What Separation Means in C++ . . . 301
10.1.1 Interface . . . 301
10.1.2 Implementation . . . 301

10.2 Why Separation Matters . . . 302
10.2.1 Stability Under Change . . . 302
10.2.2 Build Time and Dependency Control . . . 302
10.2.3 Binary Compatibility (ABI) Considerations . . . 302
10.3 Primary Techniques in C++ . . . 303
10.4 Technique 1: Header/Source Separation (Traditional C++ Model) . . . 303
10.4.1 The Basic Rule . . . 303
10.4.2 Header Hygiene (Avoid Accidental Dependencies) . . . 304
10.5 Technique 2: Abstract Interfaces (Pure Abstract Classes) . . . 304
10.5.1 When You Need Full Runtime Substitution . . . 304
10.5.2 Trade-offs . . . 305
10.6 Technique 3: PImpl (Pointer to Implementation) . . . 305
10.6.1 What PImpl Is . . . 305
10.6.2 A Clear, Modern PImpl Pattern (with `std::unique_ptr`) . . . 306
10.6.3 What PImpl Buys You . . . 307
10.6.4 PImpl Costs . . . 307
10.7 Technique 4: Modules (C++20 and Later) . . . 307
10.7.1 Interface Units vs. Implementation Units . . . 307
10.7.2 Example Structure . . . 308
10.7.3 Why Modules Help . . . 308
10.8 Special Case: Templates and “Header-Only” Reality . . . 308
10.8.1 Why Many Templates Must Be Visible . . . 308
10.8.2 Practical Approaches . . . 309
10.9 Guideline-Oriented Checklist . . . 309

11 Minimizing Dependencies 311
11.1 Why Dependencies Matter . . . 311
11.2 Core Goals of Dependency Minimization . . . 312

11.3 The Fundamental Rule: Headers Define Your Dependency Surface 312
11.3.1 A Header Is a Contract 312
11.3.2 A Header Must Be Safe to Include Multiple Times 313
11.4 Practical Techniques to Minimize Dependencies 313
11.5 Technique 1: Prefer Forward Declarations in Headers When Possible 313
11.5.1 Idea 313
11.5.2 Example 313
11.5.3 Limits 314
11.6 Technique 2: Keep Headers “Clean” and Self-Contained 314
11.6.1 Include What You Use 315
11.6.2 Include Your Own Header First 315
11.7 Technique 3: Move Implementation Details Out of Headers 315
11.7.1 Avoid Heavy Includes in Public Headers 315
11.7.2 Move Definitions to .cpp 315
11.8 Technique 4: Use PImpl to Hide Private Data and Reduce Rebuilds 316
11.8.1 Why PImpl Helps 316
11.8.2 Modern PImpl Example 316
11.8.3 Trade-offs 317
11.9 Technique 5: Prefer Non-Owning Views Over Owning Types for Read-Only Parameters 318
11.10 Technique 6: Break Cycles by Refactoring the Interface 318
11.11 Technique 7: Use C++20 Modules to Reduce Inclusion-Based Coupling . . . 319
11.11.1 What Modules Change 319
11.11.2 Minimal Example 319
11.11.3 Practical Note on Toolchains 320
11.12 Tooling Support for Dependency Control 320
11.12.1 Guideline-Driven Static Analysis 320

11.12.2 Include-Graph and “Remove Unused Headers” Tools 320
11.13 Dependency-Minimization Checklist 320

12 Resource-safe Interfaces 322
12.1 What “Resource-safe” Means at an Interface Boundary 322
12.2 Why Resource Safety Starts With Interfaces 323
12.3 The Core Rule: RAII Handles Own Resources (R.1) 323
12.3.1 Rule 323
12.3.2 Meaning 323
12.3.3 Interface Consequence 324
12.3.4 Example: Memory as a Resource 324
12.4 Ownership in Interfaces: Never Transfer Ownership via Raw Pointers (I.11) . 324
12.4.1 Rule 324
12.4.2 Reason 324
12.4.3 Recommended Ownership Shapes 325
12.4.4 Example: Transferring Ownership 325
12.4.5 Annotation for Legacy Interfaces 325
12.5 Non-nullness as a Contract: Use `not_null` (I.12) 326
12.5.1 Rule 326
12.5.2 Why It Improves Resource Safety 326
12.5.3 Example 326
12.6 Bounds Safety for Resource-like Views: Use `span` (I.13) 326
12.6.1 Rule 326
12.6.2 Reason 327
12.6.3 Preferred Interface Forms 327
12.6.4 Example 327
12.6.5 Tool Support 327
12.7 Resource-safe Parameter Passing Patterns 328

12.7.1 Pattern A: Borrowing (Non-owning Observation) . . . 328
12.7.2 Pattern B: Taking Ownership . . . 328
12.7.3 Pattern C: Shared Ownership (Use Sparingly) . . . 328
12.8 RAII Beyond Memory: Locks, Files, and Other Handles . . . 329
12.8.1 Example: Lock as a Resource . . . 329
12.8.2 Example: File as a Resource . . . 329
12.9 Static Analysis Reinforcement for Resource-safe Interfaces . . . 330
12.9.1 Ownership and Raw Pointers . . . 330
12.9.2 Non-nullness . . . 330
12.9.3 Bounds . . . 330
12.10 Checklist: Designing a Resource-safe Interface . . . 330

13 Interfaces That Express Ownership and Lifetime 332
13.1 Purpose of This Chapter . . . 332
13.2 Core Vocabulary . . . 333
13.2.1 Ownership . . . 333
13.2.2 Lifetime . . . 333
13.2.3 Observation vs. Ownership in Raw Types . . . 333
13.3 The Primary Rules for Ownership/Lifetime Interfaces . . . 333
13.3.1 I.11: Never transfer ownership by a raw pointer or reference . . . 333
13.3.2 I.12: Declare a pointer that must not be null as `not_null` . . . 334
13.3.3 R.1, R.3, R.4: RAII handles own; raw pointers/references observe . . . 334
13.4 Interface Shapes That Make Ownership Explicit . . . 334
13.4.1 Case A: The callee does not store the argument (temporary borrow) . . . 334
13.4.2 Case B: The callee requires non-null but still does not own . . . 335
13.4.3 Case C: The callee takes ownership (exclusive ownership) . . . 335
13.4.4 Case D: The callee shares ownership (shared lifetime) . . . 336
13.4.5 Case E: Observe a shared object without extending its lifetime . . . 336

13.5 Expressing Lifetime and Bounds for Views . . . 337
13.5.1 Non-owning string views . . . 337
13.5.2 Bounds-carrying contiguous views . . . 337
13.6 GSL Annotations for Legacy Interfaces . . . 338
13.7 Tooling Reinforcement: Making Contracts Enforceable . . . 338
13.7.1 MSVC Core Guidelines checks (ownership) . . . 338
13.7.2 MSVC Core Guidelines checks (non-nullness) . . . 338
13.8 Design Checklist: Ownership and Lifetime at the Boundary . . . 339

III Functions (F) 340

14 F.1 – F.60: Function Guidelines (Complete Numbered Set) 342
14.1 Scope and edition note . . . 342
14.2 F.def: Function definition rules . . . 342
14.2.1 F.1: Package meaningful operations as carefully named functions . . 342
14.2.2 F.2: A function should perform a single logical operation . . . 343
14.2.3 F.3: Keep functions short and simple . . . 343
14.2.4 F.4: If a function might have to be evaluated at compile time, declare it `constexpr` . . . 344
14.2.5 F.5: If a function is very small and time-critical, declare it `inline` . . 344
14.2.6 F.6: If your function must not throw, declare it `noexcept` . . . 344
14.2.7 F.7: For general use, take `T*` or `T&` arguments rather than smart pointers 344
14.2.8 F.8: Prefer pure functions . . . 345
14.2.9 F.9: Unused parameters should be unnamed . . . 345
14.2.10 F.10: If an operation can be reused, give it a name . . . 346
14.2.11 F.11: Use an unnamed lambda if you need a simple function object in one place only . . . 346

14.2.12 F.12: Not assigned in the Jul 8, 2025 revision 346
14.2.13 F.13: Not assigned in the Jul 8, 2025 revision 346
14.2.14 F.14: Not assigned in the Jul 8, 2025 revision 347
14.3 F.call: Parameter passing expression rules 347
14.3.1 F.15: Prefer simple and conventional ways of passing information . . 347
14.3.2 F.16: For "in" parameters, pass cheaply-copied types by value and others by reference to const 347
14.3.3 F.17: For "in-out" parameters, pass by reference to non-const 347
14.3.4 F.18: For "will-move-from" parameters, pass by `X&&` and `std::move` the parameter 348
14.3.5 F.19: For "forward" parameters, pass by `TP&&` and only `std::forward` the parameter 348
14.3.6 F.20: For "out" output values, prefer return values to output parameters 348
14.3.7 F.21: To return multiple "out" values, prefer returning a struct 349
14.4 F.call: Parameter passing semantic rules 349
14.4.1 F.22: Use `T*` or `owner<T*>` to designate a single object 349
14.4.2 F.23: Use a `not_null<T>` to indicate that "null" is not a valid value . 350
14.4.3 F.24: Use a `span<T>` (or similar) to designate a half-open sequence . 350
14.4.4 F.25: Use a `zstring` (or `not_null<zstring>`) to designate a C-style string 351
14.4.5 F.26: Use a `unique_ptr<T>` to transfer ownership where a pointer is needed 351
14.4.6 F.27: Use a `shared_ptr<T>` to share ownership 351
14.4.7 F.28: Not assigned in the Jul 8, 2025 revision 352
14.4.8 F.29: Not assigned in the Jul 8, 2025 revision 352
14.4.9 F.30: Not assigned in the Jul 8, 2025 revision 352
14.4.10 F.31: Not assigned in the Jul 8, 2025 revision 352

14.4.11 F.32: Not assigned in the Jul 8, 2025 revision . . . 352
14.4.12 F.33: Not assigned in the Jul 8, 2025 revision . . . 352
14.4.13 F.34: Not assigned in the Jul 8, 2025 revision . . . 353
14.4.14 F.35: Not assigned in the Jul 8, 2025 revision . . . 353
14.4.15 F.36: Not assigned in the Jul 8, 2025 revision . . . 353
14.4.16 F.37: Not assigned in the Jul 8, 2025 revision . . . 353
14.4.17 F.38: Not assigned in the Jul 8, 2025 revision . . . 353
14.4.18 F.39: Not assigned in the Jul 8, 2025 revision . . . 353
14.4.19 F.40: Not assigned in the Jul 8, 2025 revision . . . 353
14.4.20 F.41: Not assigned in the Jul 8, 2025 revision . . . 353
14.5 F.ret: Value return semantic rules . . . 354
14.5.1 F.42: Return a `T*` to indicate a position (only) . . . 354
14.5.2 F.43: Never (directly or indirectly) return a pointer or a reference to a local object . . . 354
14.5.3 F.44: Return a `T&` when copy is undesirable and "returning no object" isn't needed . . . 354
14.5.4 F.45: Don't return a `T&&` . . . 355
14.5.5 F.46: `int` is the return type for `main()` . . . 355
14.5.6 F.47: Return `T&` from assignment operators . . . 355
14.5.7 F.48: Don't return `std::move(local)` . . . 355
14.5.8 F.49: Don't return `const T` . . . 356
14.6 Other function rules . . . 356
14.6.1 F.50: Use a lambda when a function won't do (capture local variables or write a local function) . . . 356
14.6.2 F.51: Where there is a choice, prefer default arguments over overloading 357
14.6.3 F.52: Prefer capturing by reference in lambdas that will be used locally 357

14.6.4 F.53: Avoid capturing by reference in lambdas that will be used non-locally . . . 357
14.6.5 F.54: When writing a lambda that captures `this` or any class data member, don't use `[=]` default capture . . . 358
14.6.6 F.55: Don't use `va_arg` arguments . . . 359
14.6.7 F.56: Avoid unnecessary condition nesting . . . 359
14.6.8 F.57: Not assigned in the Jul 8, 2025 revision . . . 359
14.6.9 F.58: Not assigned in the Jul 8, 2025 revision . . . 359
14.6.10 F.59: Not assigned in the Jul 8, 2025 revision . . . 359
14.7 F.call: Cross-cutting pointer/reference selection rule . . . 360
14.7.1 F.60: Prefer `T*` over `T&` when "no argument" is a valid option . . . 360

15 Function Design Principles 361
15.1 Introduction . . . 361
15.2 Purpose of Function Design Principles . . . 361
15.3 Fundamental Design Principles . . . 362
15.3.1 Single Responsibility: One Function, One Purpose . . . 362
15.3.2 Clear and Minimal Interfaces . . . 363
15.3.3 Prefer Return Values to Output Parameters . . . 363
15.3.4 Minimize Side Effects . . . 364
15.3.5 Express Ownership and Lifetime Correctly . . . 364
15.4 Parameter Passing Strategies . . . 365
15.4.1 Pass by Value for Small, Cheap-to-Copy Types . . . 365
15.4.2 Pass by `const&` for Read-Only Large Types . . . 365
15.4.3 Pass by Reference for Mutating Parameters . . . 365
15.4.4 Perfect Forwarding in Templates . . . 366
15.5 Function Safety Principles . . . 366
15.5.1 Use `noexcept` Where Applicable . . . 366

15.5.2 Consistent Error Reporting . . . 366
15.6 Lambda and Local Function Design . . . 367
15.6.1 Use Lambdas for Local Tasks . . . 367
15.6.2 Capture Variables Explicitly . . . 367
15.7 Summary Checklist . . . 367

16 Parameter Passing 368
16.1 Introduction . . . 368
16.2 Design goal: Make intent obvious at the call site . . . 369
16.3 F.call: Core parameter passing rules (F.15–F.21) . . . 369
16.3.1 F.15: Prefer simple and conventional ways of passing information . . . 369
16.3.2 A practical intent table (normal techniques) . . . 369
16.3.3 F.16: For "in" parameters, pass cheaply-copied types by value and others by reference to `const` . . . 370
16.3.4 F.17: For "in-out" parameters, pass by reference to non-`const` . . . 371
16.3.5 F.18: For "will-move-from" parameters, pass by `X&&` and `std::move` the parameter . . . 371
16.3.6 F.19: For "forward" parameters, pass by `TP&&` and only `std::forward` the parameter . . . 372
16.3.7 F.20: For "out" output values, prefer return values to output parameters 373
16.3.8 F.21: To return multiple "out" values, prefer returning a `struct` or `tuple` . . . 373
16.4 Parameter passing semantic rules (F.22–F.27) . . . 374
16.4.1 F.22: Use `T*` or `owner<T*>` to designate a single object . . . 374
16.4.2 F.23: Use `not_null<T>` to indicate that "null" is not a valid value . . . 375
16.4.3 F.24: Use `span<T>` to designate a half-open sequence . . . 375
16.4.4 F.25: Use `zstring` or `not_null<zstring>` to designate a C-style string 376

16.4.5 F.26: Use `unique_ptr<T>` to transfer ownership where a pointer is needed . . . 376
16.4.6 F.27: Use `shared_ptr<T>` to share ownership . . . 377
16.5 F.60: Optional arguments . . . 378
16.5.1 F.60: Prefer `T*` over `T&` when "no argument" is a valid option . . . 378
16.6 Common modern C++ patterns . . . 378
16.6.1 Strings: prefer `std::string_view` for read-only input . . . 379
16.6.2 Sequences: prefer `std::span` for contiguous ranges . . . 379
16.6.3 Avoid "smart pointer parameters" unless you mean ownership . . . 379
16.7 Pitfalls and how to avoid them . . . 380
16.7.1 Pitfall: output parameters that look like inputs . . . 380
16.7.2 Pitfall: forwarding references used "for performance" . . . 380
16.7.3 Pitfall: dangling references/views . . . 380
16.8 Practical checklist . . . 380

17 Return Values 382
17.1 Introduction . . . 382
17.2 Design goals for return values . . . 382
17.3 Return categories and when to use them . . . 383
17.3.1 Return by value (`T`) . . . 383
17.3.2 Return a pointer (`T*`) . . . 383
17.3.3 Return a reference (`T&`) . . . 383
17.3.4 Return an optional-like result . . . 384
17.4 C++ Core Guidelines: Value return semantic rules (F.42–F.49) . . . 384
17.4.1 F.42: Return a `T*` to indicate a position (only) . . . 384
17.4.2 F.43: Never (directly or indirectly) return a pointer or reference to a local object . . . 385

17.4.3 F.44: Return a T& when copy is undesirable and "returning no object" isn't needed . . . 386
17.4.4 F.45: Don't return a T&& . . . 386
17.4.5 F.46: int is the return type for main() . . . 387
17.4.6 F.47: Return T& from assignment operators . . . 387
17.4.7 F.48: Don't return std::move(local) . . . 388
17.4.8 F.49: Don't return const T . . . 388
17.5 Practical patterns for robust return-value APIs . . . 389
17.5.1 Pattern: "Maybe" results (absence is normal) . . . 389
17.5.2 Pattern: Return structured results . . . 389
17.5.3 Pattern: Error signaling . . . 390
17.6 Common mistakes and how to avoid them . . . 390
17.6.1 Returning views or pointers to temporary data . . . 390
17.6.2 Returning owning raw pointers . . . 390
17.7 Checklist . . . 391

18 noexcept Usage 392
18.1 Introduction . . . 392
18.2 What noexcept means (and what it does not mean) . . . 392
18.2.1 Guaranteed behavior when violated . . . 392
18.2.2 noexcept does not mean "no failures" . . . 393
18.3 Core Guidelines perspective (F.6 and related guidance) . . . 393
18.3.1 F.6: If your function must not throw, declare it noexcept . . . 393
18.4 Where noexcept matters most . . . 394
18.4.1 Destructors and deallocation functions . . . 394
18.4.2 Move operations and performance-critical generic code . . . 395
18.4.3 swap and other "must-not-fail" primitives . . . 395
18.5 Unconditional vs conditional noexcept . . . 396

18.5.1 Unconditional `noexcept` . . . 396
18.5.2 Conditional `noexcept` (best for templates and wrappers) . . . 396
18.6 The `noexcept` operator . . . 397
18.7 API design guidance: when to use `noexcept` . . . 397
18.7.1 Good candidates . . . 397
18.7.2 Be careful (often *do not* use `noexcept`) . . . 398
18.8 Patterns for `noexcept`-friendly design . . . 398
18.8.1 Split "can-fail" from "cannot-fail" . . . 398
18.8.2 Keep `noexcept` functions "simple" and dependency-minimal . . . 399
18.9 Common mistakes and how to avoid them . . . 399
18.9.1 Mistake: declaring `noexcept` because "it probably won't throw" . . . 399
18.9.2 Mistake: a `noexcept` function calls something that might throw . . . 399
18.9.3 Mistake: returning error information by throwing from destructors . . . 400
18.10 Checklist . . . 400

19 Small vs. Large Functions 401
19.1 Introduction . . . 401
19.2 Core design principles . . . 401
19.2.1 One logical operation (F.2) . . . 402
19.2.2 Keep functions short and simple (F.3) . . . 403
19.3 Why small functions are usually better . . . 403
19.3.1 Readability and naming . . . 403
19.3.2 Testability . . . 404
19.3.3 Maintainability and change isolation . . . 404
19.4 When a larger function can be justified . . . 405
19.4.1 Performance-critical tight loops . . . 405
19.4.2 Highly cohesive algorithms . . . 405
19.4.3 Locality and invariants . . . 405

19.5 Common failure modes of large functions . . . 406
19.6 The performance dimension: inlining and code size . . . 406
19.6.1 Inlining is not the same as `inline` . . . 406
19.6.2 F.5: Very small and time-critical functions may be `inline` . . . 406
19.6.3 Small functions and code bloat . . . 407
19.7 Technique: Extracting helpers without losing clarity . . . 407
19.7.1 Extract "decision" helpers . . . 407
19.7.2 Extract phases, not random slices . . . 408
19.8 Technique: Keep scopes small . . . 408
19.9 Practical guidelines: choosing small vs. large . . . 409
19.9.1 Use small functions when . . . 409
19.9.2 Allow larger functions when . . . 409
19.10 Checklist (adoption-friendly) . . . 409

20 Const Correctness 411
20.1 Introduction . . . 411
20.2 What `const` communicates in function design . . . 411
20.2.1 Two essential questions . . . 412
20.2.2 Const correctness improves . . . 412
20.3 Const correctness for input parameters . . . 412
20.3.1 Default: input parameters are read-only . . . 412
20.3.2 Pointers: `T*` vs. `const T*` . . . 413
20.3.3 In-out parameters must be non-const . . . 414
20.4 Const correctness for member functions . . . 414
20.4.1 Default: make member functions `const` . . . 414
20.4.2 Provide const and non-const overloads for accessors . . . 415
20.4.3 Logical constness and `mutable` . . . 416
20.5 Const correctness for return types . . . 416

20.5.1 Avoid returning `const T` by value . . . 416
20.5.2 Return `T&` vs. `const T&` . . . 417
20.6 Const correctness with ownership and non-owning access . . . 417
20.6.1 Do not use smart pointers just to get const correctness . . . 417
20.7 Common mistakes and how to avoid them . . . 418
20.7.1 Mistake: forgetting `const` on read-only member functions . . . 418
20.7.2 Mistake: using `const_cast` to call non-const functions . . . 418
20.7.3 Mistake: passing non-const references for input-only arguments . . . 418
20.7.4 Mistake: returning references to internals without a const view . . . 418
20.8 Checklist . . . 418

21 Overloads 420
21.1 Introduction . . . 420
21.2 What an overload set should mean . . . 420
21.2.1 One name, one concept . . . 421
21.3 F.51: Default arguments vs. overloading . . . 421
21.3.1 F.51: Where there is a choice, prefer default arguments over overloading 421
21.3.2 Recommended pattern: one implementation + defaults . . . 422
21.3.3 When overloading is better than defaults . . . 422
21.3.4 Critical caveat: default arguments and virtual functions . . . 422
21.4 Designing safe overload sets . . . 423
21.4.1 Avoid “almost the same” overloads . . . 423
21.4.2 Prefer explicitness over clever conversions . . . 424
21.4.3 Use `const` overloads for accessors . . . 425
21.4.4 Use ref-qualifiers when value category changes semantics . . . 425
21.5 Overloading vs. templates . . . 426
21.5.1 Do not use forwarding references unless you are truly forwarding . . . 426
21.5.2 Constrain templates when they participate in overload resolution . . . 426

21.6 Common overload-resolution traps . . . 427
21.6.1 `std::initializer_list` preference . . . 427
21.6.2 Ambiguity with user-defined conversions . . . 427
21.7 Documentation and enforcement practices . . . 428
21.7.1 Document overload set invariants . . . 428
21.7.2 Static analysis hint (Core Guidelines) . . . 428
21.8 Checklist . . . 428

22 Lambdas 430
22.1 Introduction . . . 430
22.2 What a lambda is (precise model) . . . 431
22.3 Why lambdas are preferred in modern C++ . . . 431
22.3.1 Locality of logic . . . 431
22.3.2 Better composition with algorithms . . . 432
22.3.3 Avoiding global helper functions . . . 432
22.4 Core Guidelines rules for lambdas (F.50–F.54) . . . 432
22.4.1 F.50: Use a lambda when a function won't do (to capture local variables, or write a local function) . . . 432
22.4.2 F.51: Where there is a choice, prefer default arguments over overloading 433
22.4.3 F.52: Prefer capturing by reference in lambdas that will be used locally 433
22.4.4 F.53: Avoid capturing by reference in lambdas that will be used non-locally . . . 433
22.4.5 F.54: When capturing `this` (or data members), do not use `[=]` default capture . . . 434
22.5 Capture semantics: value, reference, and lifetime . . . 435
22.5.1 Capture by value . . . 435
22.5.2 Capture by reference . . . 435
22.5.3 Capturing `this` and object lifetime . . . 436

22.6 Generic lambdas and type deduction . . . 437
22.7 Mutable lambdas . . . 437
22.8 Lambdas vs. `std::function` . . . 438
22.8.1 Prefer templates/`auto` for performance and generality . . . 438
22.8.2 Use `std::function` when you need type erasure . . . 438
22.9 Common mistakes and how to avoid them . . . 438
22.9.1 Dangling captures . . . 439
22.9.2 Overusing default capture . . . 439
22.9.3 Using lambdas where a named function improves clarity . . . 439
22.10 Checklist . . . 439

IV Classes (C) 440

23 Classes (C.1–C.183) 442
23.1 Scope and intent . . . 442
23.2 A practical mental model . . . 442
23.3 Rule directory (C.1–C.183) . . . 443
23.3.1 C: General class rules (C.1–C.9) . . . 443
23.3.2 C.concrete: Concrete types (C.10–C.49) . . . 444
23.3.3 C.ctor: Constructors, assignments, and destructors (C.50–C.99) . . . 446
23.3.4 C.con: Containers and other resource handles (C.100–C.109) . . . 449
23.3.5 C.lambdas: Function objects and lambdas (C.150–C.170) . . . 449
23.3.6 C.hier: Class hierarchies (OOP) (C.120–C.149) . . . 451
23.3.7 C.over: Overloading and overloaded operators (C.160–C.179) . . . 452
23.3.8 C.union: Unions (C.180–C.183) . . . 454
23.4 Representative examples (core patterns you will reuse) . . . 454
23.4.1 Invariants and encapsulation (C.2, C.6, C.9) . . . 454

23.4.2 Rule of zero / five with RAII (C.23, C.61, C.75) . . . 455
23.4.3 Safe polymorphic base (C.121, C.124, C.127, C.130) . . . 455
23.4.4 Tagged union alternative (C.181–C.183) . . . 456
23.5 Adoption checklist for real code bases . . . 456

24 Class Design 458
24.1 Introduction . . . 458
24.2 Choose the right kind of type . . . 459
24.2.1 Concrete/value type . . . 459
24.2.2 Resource handle (RAII type) . . . 459
24.2.3 Polymorphic hierarchy . . . 459
24.3 Invariants and encapsulation . . . 459
24.3.1 Use `class` for invariants; `struct` for simple aggregates . . . 459
24.3.2 Keep representation private . . . 460
24.3.3 Do not expose writable base-class data . . . 460
24.4 Interface vs. implementation . . . 461
24.4.1 Separate interface and representation . . . 461
24.4.2 Use Pimpl when you need ABI stability or heavy implementation hiding 461
24.5 Member vs. non-member functions . . . 462
24.5.1 Make a function a member only if it needs representation access . . . 462
24.5.2 Place helper functions in the same namespace . . . 462
24.6 Construction, invariants, and safety . . . 463
24.6.1 Constructors establish invariants . . . 463
24.6.2 Prefer initialization over assignment . . . 463
24.6.3 Never call virtual functions from constructors or destructors . . . 464
24.7 Default operations: rule of zero and rule of five . . . 464
24.7.1 Prefer the rule of zero . . . 464
24.7.2 If you manage a resource, design all special members intentionally . . . 464

24.8 Const correctness and access design . . . 465
24.8.1 Make read-only operations `const` . . . 465
24.8.2 Provide const and non-const accessors when returning references . . . 465
24.9 Operator overloading and conversions . . . 466
24.9.1 Overload only when meaning is conventional . . . 466
24.9.2 Avoid surprising implicit conversions . . . 466
24.10 Class hierarchies: safe and evolvable polymorphism . . . 466
24.10.1 Only use inheritance for polymorphism . . . 467
24.10.2 Polymorphic base checklist . . . 467
24.11 Unions, variants, and representing alternatives . . . 468
24.11.1 Avoid naked unions . . . 468
24.11.2 Do not use unions for type punning . . . 468
24.12 Practical class design workflow . . . 468
24.12.1 Step 1: Write the invariant . . . 469
24.12.2 Step 2: Decide ownership . . . 469
24.12.3 Step 3: Design the public API to prevent invalid states . . . 469
24.12.4 Step 4: Minimize coupling . . . 469
24.13 Checklist . . . 470

25 Invariants **471**
25.1 Introduction . . . 471
25.2 Why invariants matter in C++ class design . . . 472
25.2.1 Encapsulation is mainly about preserving invariants . . . 472
25.2.2 Invariants enable reliable reasoning . . . 472
25.2.3 Invariants and exception safety are linked . . . 472
25.3 Core Guidelines view of invariants (Classes section) . . . 473
25.3.1 Use `class` when an invariant exists . . . 473
25.3.2 Keep representation private to enforce the invariant . . . 473

25.3.3 Invariants are established by constructors . . . 473
25.4 Defining an invariant . . . 473
25.4.1 Write the invariant explicitly . . . 473
25.4.2 Prefer making invalid states unrepresentable . . . 474
25.5 Establishing invariants . . . 474
25.5.1 Constructors must establish the invariant or fail . . . 474
25.5.2 Use factories when construction needs multi-step validation . . . 475
25.6 Preserving invariants in member functions . . . 475
25.6.1 Public member functions must restore the invariant . . . 476
25.6.2 Keep mutating operations narrow and checked . . . 476
25.6.3 Const correctness protects invariants . . . 477
25.7 Preconditions, postconditions, and invariants . . . 477
25.7.1 How they relate . . . 477
25.7.2 Document requirements clearly . . . 477
25.8 Techniques for checking invariants . . . 478
25.8.1 Runtime checking in debug builds . . . 478
25.8.2 RAII guard to check on exit even with exceptions . . . 479
25.9 Invariants and class hierarchies . . . 480
25.9.1 Base-class invariants must be simple and stable . . . 480
25.9.2 Do not call virtual functions from constructors or destructors . . . 481
25.10 Invariants and ownership . . . 481
25.10.1 Ownership rules are invariants . . . 481
25.10.2 Move operations must preserve validity . . . 481
25.11 Common invariant failures and fixes . . . 482
25.11.1 Public data members in an invariant-bearing type . . . 482
25.11.2 Two-phase initialization . . . 482
25.11.3 Complex functions that modify many fields . . . 483

25.12 Checklist 483

26 Constructors **484**

26.1 Introduction 484
26.2 Primary rule: constructors establish invariants 485
26.2.1 Class invariant and validity 485
26.2.2 Avoid two-phase initialization 485
26.3 Prefer initialization over assignment 486
26.3.1 Member initializer lists 486
26.3.2 Initialization order is declaration order 486
26.4 Choosing the right constructor set 487
26.4.1 Keep constructors minimal and coherent 487
26.4.2 Prefer non-throwing default construction only when meaningful . . . 487
26.5 Avoid unintended implicit conversions 487
26.5.1 Use `explicit` for single-argument constructors 488
26.5.2 `explicit(bool)` (conditional explicitness) 488
26.6 Delegating and inheriting constructors 489
26.6.1 Delegating constructors 489
26.6.2 Inheriting constructors 489
26.7 Factories for “virtual behavior” during initialization 490
26.7.1 Do not rely on virtual dispatch in constructors 490
26.8 Exception safety in constructors 491
26.8.1 If construction fails, no object exists 491
26.8.2 RAII members simplify correctness 491
26.9 Copy, move, and special member constructors 492
26.9.1 Rule of zero (preferred) 492
26.9.2 Resource owners: design copying and moving intentionally 492
26.10 Performance and correctness notes 493

26.10.1 Avoid heavy work in constructors when it harms usability 493
26.10.2 Prefer `noexcept` where non-throwing is part of the design 494
26.10.3 `constexpr` constructors 494
26.11 Common constructor pitfalls 494
26.11.1 Leaving members uninitialized 494
26.11.2 Narrowing conversions 495
26.11.3 Ambiguity with `initializer_list` 495
26.11.4 Calling virtual functions in constructors/destructors 495
26.12 Checklist 495

27 Destructors **497**
27.1 Introduction 497
27.2 Destructor fundamentals (what the language guarantees) 498
27.2.1 When destructors run 498
27.2.2 Destruction order 498
27.2.3 A destructor is implicitly `noexcept` 498
27.3 Core Guidelines: the destructor rules 499
27.3.1 Do not throw from destructors 499
27.3.2 Use RAII and prefer the rule of zero 499
27.3.3 If you define a destructor, review all special members 499
27.4 Designing a correct destructor 500
27.4.1 Destructor responsibilities 500
27.4.2 Keep cleanup non-throwing 500
27.5 Destructors and ownership 501
27.5.1 RAII makes ownership explicit 501
27.5.2 Prefer `unique_ptr` for exclusive ownership 502
27.5.3 “After move” validity is part of the invariant 502
27.6 Destructors and inheritance 502

27.6.1 Virtual destructor for polymorphic base classes . . . 502
27.6.2 When a base class should *not* be deletable . . . 503
27.6.3 Do not call virtual functions from destructors . . . 503
27.7 Custom destructors and the special member set . . . 503
27.7.1 Destructor presence affects implicit move generation . . . 504
27.7.2 Prefer `=default` when you need a declared destructor for form . . . 504
27.8 Best practices for "cleanup that can fail" . . . 504
27.8.1 Provide explicit operations for error reporting . . . 504
27.8.2 Never rely on destructors for business-critical error reporting . . . 504
27.9 Common destructor mistakes . . . 505
27.9.1 Throwing exceptions . . . 505
27.9.2 Manual `delete` on owning raw pointers . . . 505
27.9.3 Polymorphic base without virtual destructor . . . 505
27.9.4 Doing too much work in destructors . . . 505
27.10 Checklist . . . 505

28 Rule of 0 / 3 / 5 507
28.1 Introduction . . . 507
28.2 Special member functions (the set you must understand) . . . 507
28.3 Rule of 0 . . . 508
28.3.1 Definition . . . 508
28.3.2 Why the Core Guidelines prefer it . . . 508
28.3.3 Example: value type with RAII members . . . 508
28.3.4 What makes Rule of 0 possible . . . 509
28.4 Rule of 3 . . . 509
28.4.1 Definition (historical and still relevant) . . . 509
28.4.2 Why it still matters today . . . 510
28.4.3 Example: deep-copying resource owner (Rule of 3) . . . 510

28.5 Rule of 5 . . . 511
28.5.1 Definition . . . 512
28.5.2 Why it exists . . . 512
28.5.3 Move-only types (unique ownership) . . . 512
28.5.4 Copyable resource owners (deep copy + move) . . . 513
28.6 Rule of 0 vs. Rule of 5: how to choose . . . 515
28.6.1 Prefer Rule of 0 whenever possible . . . 515
28.6.2 When Rule of 5 is justified . . . 515
28.7 Important modern details (that cause real bugs) . . . 516
28.7.1 A user-declared destructor can inhibit implicit moves . . . 516
28.7.2 Use `=default` and `=delete` to express intent . . . 516
28.7.3 Move operations should often be `noexcept` . . . 516
28.7.4 Destructors must not throw . . . 517
28.8 A practical decision checklist . . . 517

29 Copy and Move Semantics 518
29.1 Introduction . . . 518
29.2 The five special member functions . . . 519
29.3 Design decision: value, unique ownership, or shared ownership . . . 519
29.3.1 Value semantics (copy means independent value) . . . 519
29.3.2 Unique ownership (move-only) . . . 520
29.3.3 Shared ownership (explicit sharing) . . . 520
29.4 Rule of zero first . . . 520
29.4.1 Prefer the rule of zero . . . 520
29.5 Copy semantics . . . 521
29.5.1 Copy constructor: create a new object from an existing one . . . 521
29.5.2 Copy assignment: replace this object's value . . . 521
29.5.3 Deep copy for resource owners . . . 521

29.6 Move semantics . . . 523
29.6.1 Move constructor and move assignment . . . 523
29.6.2 Move-only type (unique ownership) example . . . 523
29.6.3 Why `noexcept` on moves matters . . . 524
29.7 Defaulted and deleted operations . . . 524
29.7.1 Express intent with `=default` and `=delete` . . . 524
29.7.2 Be aware of implicit generation rules . . . 525
29.8 Exception safety and strong invariants . . . 525
29.8.1 Copy assignment and the strong guarantee . . . 525
29.8.2 Destructors must not throw . . . 526
29.9 Copy and move in class hierarchies . . . 526
29.9.1 Avoid slicing . . . 526
29.9.2 Consider deleting base copy operations . . . 527
29.10 Practical guidance for real code bases . . . 527
29.10.1 Prefer standard RAII members . . . 527
29.10.2 Make semantics obvious at the interface . . . 527
29.11 Checklist . . . 528

30 Inheritance 529

30.1 Introduction . . . 529
30.2 When inheritance is the right tool . . . 530
30.2.1 Public inheritance for substitutability (“is-a”) . . . 530
30.2.2 Prefer composition for reuse . . . 530
30.3 The safe polymorphic base: required design elements . . . 531
30.3.1 Virtual destructor (deletion through base) . . . 531
30.3.2 No public or protected representation . . . 532
30.3.3 Use `override` consistently . . . 532
30.3.4 Use `final` when further derivation is not intended . . . 533

30.4 Object slicing and copying in hierarchies . . . 533
30.4.1 Avoid slicing . . . 533
30.4.2 If copying polymorphic objects is needed, use `clone()` . . . 534
30.4.3 Consider deleting copy operations in the base . . . 534
30.5 Virtual calls during construction and destruction . . . 535
30.5.1 Do not call virtual functions from constructors or destructors . . . 535
30.5.2 Use the Non-Virtual Interface (NVI) pattern for invariants . . . 535
30.6 Overload sets and name hiding in derived classes . . . 536
30.6.1 Use `using Base::f` to expose base overloads . . . 536
30.7 Multiple inheritance and virtual inheritance . . . 536
30.7.1 Multiple inheritance: use primarily for interfaces . . . 536
30.7.2 Virtual inheritance is for real diamond problems . . . 537
30.8 Downcasting and RTTI . . . 537
30.8.1 Avoid downcasts when possible . . . 537
30.8.2 If you must check dynamic type, be explicit and safe . . . 537
30.9 Public, protected, private inheritance . . . 538
30.9.1 Prefer public inheritance for substitutability . . . 538
30.9.2 Prefer composition over private inheritance . . . 538
30.10 Practical hierarchy checklist . . . 538

31 Polymorphism 540
31.1 Introduction . . . 540
31.2 When to use runtime polymorphism . . . 540
31.2.1 Use it only when you need dynamic behavior selection . . . 541
31.2.2 Prefer alternatives when runtime polymorphism is not required . . . 541
31.3 Core principles of safe runtime polymorphism . . . 541
31.3.1 Design for substitutability . . . 541
31.3.2 Keep base interfaces small and stable . . . 542

31.3.3 Use inheritance primarily for runtime polymorphism 542
31.4 The polymorphic base class: mandatory rules 542
31.4.1 Virtual destructor when deletion via base is possible 542
31.4.2 Do not expose representation in a base class 543
31.4.3 Use `override` and `final` 543
31.4.4 Do not call virtual functions from constructors/destructors 544
31.5 Object slicing and ownership 544
31.5.1 Avoid object slicing 544
31.5.2 Prefer `unique_ptr` for ownership 544
31.5.3 Copying polymorphic objects: use `clone()` 545
31.6 Runtime polymorphism patterns 545
31.6.1 Abstract base class (interface) 546
31.6.2 Non-Virtual Interface (NVI) 546
31.6.3 Type erasure as an alternative (`std::function` and custom wrappers) 547
31.7 Downcasting and RTTI 547
31.7.1 Avoid downcasting when possible 547
31.7.2 If you must, use `dynamic_cast` deliberately 547
31.8 Performance characteristics 548
31.8.1 What virtual dispatch costs 548
31.9 Multiple inheritance: safe use 548
31.10 Checklist for safe polymorphic design 549

32 `final` / `override` 550
32.1 Introduction 550
32.2 What `override` means 550
32.2.1 Definition 550
32.2.2 Why `override` matters 551
32.2.3 Example: a silent bug without `override` 551

32.2.4 The same code fixed with `override` . . . 551
32.2.5 Guideline practice: mark every override . . . 552
32.3 What `final` means . . . 552
32.3.1 Definition (two forms) . . . 552
32.3.2 Function `final` . . . 552
32.3.3 Class `final` . . . 553
32.3.4 Why `final` matters . . . 553
32.4 How `override` and `final` interact with class design . . . 553
32.4.1 Polymorphic base classes and safe overriding . . . 553
32.4.2 Non-Virtual Interface (NVI) with `final` . . . 554
32.5 Common pitfalls and how the specifiers prevent them . . . 555
32.5.1 Name hiding and overload sets . . . 555
32.5.2 Signature mismatches . . . 555
32.5.3 Accidental overriding vs. deliberate sealing . . . 556
32.6 Performance notes (secondary to correctness) . . . 556
32.7 Recommended style rules . . . 556
32.7.1 A disciplined, guideline-aligned policy . . . 556
32.7.2 Where to place the specifiers . . . 556
32.8 Checklist . . . 557

V Enumerations (Enum) 558

33 Enum.1 – Enum.10: Enumerations (Complete Numbered Set) 560
33.1 Scope and numbering note . . . 560
33.2 Enum.1: Prefer enumerations over macros . . . 560
33.3 Enum.2: Use enumerations to represent sets of related named constants . . . 561
33.4 Enum.3: Prefer `enum class` over “plain” `enum` . . . 562

33.5 Enum.4: Define operations on enumerations for safe and simple use 563
33.6 Enum.5: Don't use ALL_CAPS for enumerators 565
33.7 Enum.6: Avoid unnamed enumerations 565
33.8 Enum.7: Specify the underlying type only when necessary 566
33.9 Enum.8: Specify enumerator values only when necessary 567
33.10 Enum.9: Not assigned in the Jul 8, 2025 revision 568
33.11 Enum.10: Not assigned in the Jul 8, 2025 revision 568
33.12 Quick checklist (engineering use) 569

34 enum class 570
34.1 Introduction 570
34.2 Why enum class is the default choice 570
34.2.1 Scoped enumerators prevent name pollution 570
34.2.2 No implicit conversion to int 571
34.3 Syntax and core rules 571
34.3.1 Basic declaration 571
34.3.2 Underlying type (only when needed) 572
34.3.3 Forward declaration 572
34.4 Using enum class in control flow 572
34.4.1 switch over a scoped enum 572
34.5 Converting to the underlying integer safely 573
34.5.1 Prefer std::to_underlying in C++23 and later 573
34.5.2 Pre-C++23 approach: std::underlying_type 573
34.6 Flags and bitmasks with enum class 574
34.6.1 Do not rely on implicit bitwise operations 574
34.6.2 Define explicit operators for flag enums 574
34.7 Modern convenience: using enum (C++20) 575
34.8 Design guidance aligned with the Core Guidelines 575

34.8.1 Prefer `enum class` by default . . . 575
34.8.2 Avoid `ALL_CAPS` enumerators . . . 576
34.8.3 Assign explicit values only when necessary . . . 576
34.8.4 Document meaning and keep enums cohesive . . . 576
34.9 Common mistakes and how `enum class` helps . . . 576
34.9.1 Accidental mixing of unrelated enums . . . 576
34.9.2 Overusing casts . . . 576
34.10 Checklist . . . 577

35 Avoiding Unsafe Enums 578
35.1 Introduction . . . 578
35.2 What makes an enum “unsafe” in practice . . . 579
35.2.1 Unscoped enums leak names and convert implicitly . . . 579
35.2.2 Enums can carry invalid values . . . 579
35.2.3 Flags are often modeled incorrectly . . . 580
35.3 Core Guidelines strategy for safety . . . 580
35.4 Safer alternative #1: use `enum class` by default . . . 580
35.4.1 Scoped enumerators prevent name collisions . . . 580
35.4.2 No implicit conversion to `int` . . . 581
35.5 Safer alternative #2: keep enums cohesive and named . . . 581
35.5.1 Avoid unnamed enums for unrelated constants . . . 581
35.5.2 Avoid `ALL_CAPS` enumerators . . . 582
35.6 Safer alternative #3: convert explicitly at boundaries . . . 582
35.6.1 Prefer `std::to_underlying` in C++23 . . . 582
35.6.2 Pre-C++23: use `std::underlying_type` . . . 582
35.7 Safer alternative #4: validate enum values from external input . . . 583
35.7.1 Use a “try-parse” pattern . . . 583
35.7.2 Prefer mapping functions over casts . . . 583

35.8 Safer alternative #5: design flags as flags (and define operations) 583
35.8.1 Define explicit bitwise operators 584
35.8.2 Keep explicit enumerator values only when needed 584
35.9 Safer alternative #6: keep underlying types and explicit values minimal 585
35.9.1 Do not specify underlying type unless necessary 585
35.9.2 Do not assign explicit values unless necessary 585
35.10 Safer alternative #7: write safe API surfaces 585
35.10.1 Prefer enums in public APIs instead of integers 585
35.10.2 Avoid “sentinel” values like `-1` when possible 586
35.11 Safe usage patterns in real code 586
35.11.1 Provide conversion to text via total mapping 586
35.11.2 Use `using enum` locally when it improves clarity (C++20) 586
35.12 Checklist: how to avoid unsafe enums 587

36 Explicit Conversions 588
36.1 Introduction 588
36.2 Why explicit conversions matter for enumerations 589
36.2.1 The core problem: integers are too permissive 589
36.2.2 `enum class` enforces explicit conversions 589
36.3 Forms of explicit conversions in C++ 589
36.3.1 Explicit construction 590
36.3.2 `static_cast` 590
36.3.3 Library helpers for enum conversion 591
36.3.4 User-defined explicit conversion operators 592
36.4 Guideline-aligned rules for explicit conversions with enums 592
36.4.1 Prefer `enum class` to force explicit conversions 592
36.4.2 Define operations instead of relying on integer casts 593
36.4.3 Convert only at boundaries 593

36.5 The unsafe part: explicit casts can still be wrong . . . 593
36.5.1 Explicit does not guarantee correctness . . . 593
36.5.2 Casting integers to enums can create invalid values . . . 594
36.5.3 Preferred approach: validate during decoding . . . 594
36.6 Explicit conversions and narrowing . . . 594
36.6.1 Narrowing conversions are a common source of defects . . . 594
36.7 Choosing the right explicit conversion technique . . . 595
36.7.1 For enums to integers . . . 595
36.7.2 For integers to enums . . . 595
36.7.3 For domain types to primitives . . . 595
36.8 Checklist . . . 596

37 Type Compatibility 597
37.1 Introduction . . . 597
37.2 Compatibility rules: unscoped `enum` vs. scoped `enum class` . . . 598
37.2.1 Unscoped `enum`: compatible with integers in many contexts . . . 598
37.2.2 Scoped `enum class`: incompatible with integers by default . . . 599
37.3 Compatibility across different enums . . . 599
37.3.1 Different enum types are not interchangeable . . . 599
37.4 Underlying types and representation compatibility . . . 600
37.4.1 Underlying type basics . . . 600
37.4.2 When underlying type matters for compatibility . . . 600
37.4.3 Do not treat “same underlying type” as semantic compatibility . . . 601
37.5 Compatibility with integers and explicit conversions . . . 601
37.5.1 Prefer explicit conversions at boundaries . . . 601
37.5.2 Use `std::to_underlying` (C++23) for encoding . . . 601
37.5.3 Validate when decoding . . . 602
37.6 Compatibility in expressions and comparisons . . . 602

37.6.1 Comparisons are type-safe with `enum class` . . . 602
37.6.2 Arithmetic is not defined for `enum class` unless you define it . . . 603
37.7 Overload resolution and API compatibility . . . 603
37.7.1 Problem: unscoped enums can bind to unintended overloads . . . 603
37.7.2 Solution: use `enum class` and take the enum type in APIs . . . 604
37.8 Interoperability patterns (when compatibility is required) . . . 604
37.8.1 Define explicit mapping functions . . . 604
37.8.2 Avoid reintroducing implicit compatibility via helper macros . . . 605
37.9 Design rules for compatibility that stays safe . . . 605
37.9.1 Use `enum class` for domain separation . . . 605
37.9.2 Keep explicit values and underlying types minimal . . . 605
37.9.3 Prefer return/parameter types that preserve domain meaning . . . 605
37.10 Checklist . . . 606

VI Resource Management (R) 607

38 Resource Management (R.1–R.17) 609
38.1 Purpose and scope . . . 609
38.2 Rule map for this chapter (R.1–R.17) . . . 610
38.3 R.1: Manage resources automatically using RAII . . . 610
38.3.1 Rule . . . 610
38.3.2 Why . . . 610
38.3.3 What to do . . . 611
38.3.4 Example: manual cleanup (bad) vs RAII (good) . . . 611
38.3.5 Practical guidance . . . 612
38.4 R.2: In interfaces, use raw pointers to denote individual objects (only) . . . 612
38.4.1 Rule . . . 612

38.4.2 Why . . . 613
38.4.3 What to do . . . 613
38.4.4 Examples . . . 613
38.4.5 Notes . . . 613
38.5 R.3: A raw pointer (T*) is non-owning . . . 614
38.5.1 Rule . . . 614
38.5.2 Why . . . 614
38.5.3 What to do . . . 614
38.5.4 Example . . . 614
38.6 R.4: A raw reference (T&) is non-owning . . . 615
38.6.1 Rule . . . 615
38.6.2 Why . . . 615
38.6.3 What to do . . . 615
38.6.4 Example . . . 615
38.7 R.5: Prefer scoped objects; do not heap-allocate unnecessarily . . . 616
38.7.1 Rule . . . 616
38.7.2 Why . . . 616
38.7.3 What to do . . . 616
38.7.4 Example . . . 617
38.8 R.6: Avoid non-const global variables . . . 617
38.8.1 Rule . . . 617
38.8.2 Why . . . 617
38.8.3 What to do . . . 617
38.8.4 Example . . . 618
38.9 R.10: Avoid malloc() and free() . . . 618
38.9.1 Rule . . . 618
38.9.2 Why . . . 618

38.9.3 What to do . . . 618
38.9.4 Example: wrapping `malloc` with RAII . . . 619
38.10 R.11: Avoid calling `new` and `delete` explicitly . . . 619
38.10.1 Rule . . . 619
38.10.2 Why . . . 619
38.10.3 What to do . . . 619
38.10.4 Example . . . 620
38.11 R.12: Immediately give the result of an explicit allocation to a manager object 620
38.11.1 Rule . . . 620
38.11.2 Why . . . 620
38.11.3 What to do . . . 620
38.11.4 Example . . . 621
38.12 R.13: Perform at most one explicit resource allocation in a single expression . 621
38.12.1 Rule . . . 621
38.12.2 Why . . . 621
38.12.3 What to do . . . 621
38.12.4 Example . . . 622
38.13 R.14: Avoid `[]` parameters; prefer `span` . . . 622
38.13.1 Rule . . . 622
38.13.2 Why . . . 622
38.13.3 What to do . . . 622
38.13.4 Example . . . 623
38.14 R.15: Always overload matched allocation/deallocation pairs . . . 623
38.14.1 Rule . . . 623
38.14.2 Why . . . 623
38.14.3 What to do . . . 623
38.14.4 Example . . . 624

38.15 R.16 (reserved): Unassigned in current guideline set . . . 624
38.16 R.17 (reserved): Unassigned in current guideline set . . . 624
38.17 Common patterns and checklists (practical summary) . . . 625
38.17.1 Ownership and lifetimes checklist . . . 625
38.17.2 Exception safety checklist . . . 625

39 RAII (Resource Acquisition Is Initialization) 626
39.1 What RAII means in modern C++ . . . 626
39.1.1 Definition . . . 626
39.1.2 Resources covered by RAII . . . 627
39.1.3 Relationship to the Core Guidelines (Resource Management) . . . 627
39.2 Why RAII is the default resource-management model . . . 627
39.2.1 The problem with manual cleanup . . . 627
39.2.2 RAII advantages . . . 628
39.3 The RAII contract: invariants and destructor rules . . . 628
39.3.1 Constructor establishes the invariant . . . 628
39.3.2 Destructor releases the resource and must not throw . . . 629
39.4 A minimal RAII wrapper for non-memory resources . . . 629
39.4.1 Design goals . . . 629
39.4.2 Example: generic handle wrapper pattern . . . 630
39.4.3 Example: POSIX file descriptor . . . 631
39.5 RAII for memory: prefer standard owners and containers . . . 631
39.5.1 The preferred owning tools . . . 631
39.5.2 Prefer factory functions . . . 632
39.6 RAII for synchronization: scoped locking . . . 632
39.6.1 The standard pattern . . . 632
39.6.2 Example: `std::lock_guard` . . . 633
39.6.3 When to prefer `std::unique_lock` . . . 633

39.7 RAII for files and streams . . . 633
39.7.1 Streams are RAII types . . . 633
39.7.2 Example: safe file reading with exceptions enabled . . . 633
39.7.3 Policy note . . . 634
39.8 Views vs owners: `std::span` completes the RAII picture . . . 634
39.8.1 Preferred interfaces . . . 634
39.8.2 Example: API that avoids pointer + length . . . 635
39.9 Move semantics: the engine of unique ownership . . . 635
39.9.1 Why move matters for RAII . . . 635
39.9.2 Rule of zero, rule of five, and RAII . . . 635
39.9.3 Example: move-only RAII type . . . 635
39.10 Custom deleters: RAII for "foreign" resources . . . 636
39.10.1 Example: RAII with custom deleter . . . 636
39.11 Scope guards: RAII for "do this when leaving scope" . . . 637
39.11.1 Example: minimal scope guard . . . 637
39.11.2 Example: transactional rollback . . . 638
39.12 Common RAII pitfalls and how to avoid them . . . 638
39.12.1 Pitfall 1: throwing destructors . . . 638
39.12.2 Pitfall 2: shared ownership when you really have a single owner . . . 639
39.12.3 Pitfall 3: reference cycles with `shared_ptr` . . . 639
39.12.4 Pitfall 4: returning views to objects that will not outlive the view . . . 639
39.12.5 Pitfall 5: confusing "scope-based" lifetime with "resource" lifetime . 639
39.13 Practical design checklist for RAII types . . . 639
39.14 Summary . . . 640

40 Resource Ownership 641
40.1 What "ownership" means . . . 641
40.1.1 Definition . . . 641

40.1.2 Resources in scope 641
40.2 Owner vs observer (the central distinction) 642
40.2.1 Owners 642
40.2.2 Observers 642
40.2.3 Core Guidelines rule summary 643
40.3 Three ownership models you must choose from 643
40.3.1 Model 1: Single ownership 643
40.3.2 Model 2: Shared ownership 643
40.3.3 Model 3: No ownership in the interface (borrowed use) 644
40.4 How to express ownership in APIs (recommended patterns) 644
40.4.1 Passing a single object 644
40.4.2 Passing sequences (arrays, buffers) 645
40.4.3 Returning ownership 646
40.5 Ownership in class design 647
40.5.1 Rule of zero: let members own 647
40.5.2 When you manage a raw handle directly (move-only RAII) 647
40.6 Avoiding dangerous ownership patterns 649
40.6.1 Never transfer ownership by a raw pointer 649
40.6.2 Avoid owning raw pointers and owning references 649
40.6.3 Prefer factories to explicit `new` 649
40.6.4 Shared ownership pitfalls 650
40.7 Ownership annotations and the Guidelines Support Library (GSL) 650
40.7.1 Why annotations exist 650
40.7.2 Typical vocabulary 650
40.8 A practical checklist for resource ownership 651
40.9 Summary 652

41 `new` / `delete` in Modern C++ 653

41.1 Purpose of this chapter . . . 653
41.2 Core Guidelines viewpoint (Resource Management) . . . 653
41.3 What new really does . . . 654
41.3.1 Two operations: allocation + construction . . . 654
41.3.2 Failure behavior . . . 654
41.3.3 Array new . . . 655
41.4 What delete really does . . . 655
41.4.1 Two operations: destruction + deallocation . . . 655
41.4.2 delete on null . . . 655
41.4.3 Array delete must match array new . . . 656
41.4.4 Deleting through a base pointer . . . 656
41.5 Why explicit new/delete are discouraged . . . 656
41.6 Modern replacements (preferred patterns) . . . 657
41.6.1 Prefer owning containers . . . 657
41.6.2 Prefer std::make_unique for single objects . . . 657
41.6.3 Prefer std::make_shared when you truly need shared ownership . . . 658
41.6.4 Prefer std::unique_ptr<T[]> only when you really need a raw array 658
41.7 R.12 and R.13 in practice: exception safety and clarity . . . 658
41.7.1 R.12: Immediately bind to an owner . . . 658
41.7.2 R.13: One explicit allocation per statement . . . 659
41.8 When explicit new/delete can be justified . . . 659
41.9 Placement new: construction without allocation . . . 660
41.9.1 What it is . . . 660
41.9.2 Correct use cases . . . 660
41.9.3 Pitfall: forgetting explicit destruction . . . 660
41.10 Overloading allocation and deallocation functions . . . 660
41.10.1 Class-specific operator new/operator delete . . . 660

41.10.2 Sized deallocation and alignment-aware allocation (modern C++) . . 661
41.11 The `new/delete` taxonomy (quick reference) 661
41.11.1 `new` 661
41.11.2 `delete` 662
41.12 Common correctness rules (non-negotiable) 662
41.13 How to design code so you almost never write `delete` 662
41.13.1 Rule of zero 662
41.13.2 Return-by-value is ownership transfer 663
41.14 Performance note: allocation is expensive, but design comes first 663
41.15 Summary 664

42 Avoiding Leaks 665
42.1 What is a leak (precise definition) 665
42.1.1 Why leaks matter 666
42.2 Core Guidelines principles behind leak prevention 666
42.3 Leak classes (what actually goes wrong) 666
42.3.1 Class 1: missing cleanup on one control-flow path 666
42.3.2 Class 2: unclear ownership (two owners or no owner) 667
42.3.3 Class 3: mismatched acquire/release APIs 668
42.3.4 Class 4: cycles in shared ownership 668
42.3.5 Class 5: lifetime errors of views (not a leak, but often related) 668
42.4 Primary cure: RAII everywhere 669
42.4.1 Rule: every resource must have an owner object 669
42.4.2 Memory resources: use containers and smart pointers 669
42.4.3 Non-memory resources: create a small handle wrapper 669
42.5 Avoid explicit `new/delete` in application code 671
42.5.1 Prefer factories 671
42.5.2 R.12: if you must allocate explicitly, bind immediately 671

42.6 Make ownership explicit in interfaces . . . 671
42.6.1 Raw pointers are non-owning observers . . . 671
42.6.2 Transfer ownership with `unique_ptr` . . . 672
42.6.3 Use `std::span` for sequences instead of pointer+size . . . 672
42.7 Exception safety: preventing leaks under failure . . . 672
42.7.1 RAII makes basic exception safety automatic . . . 672
42.7.2 Do not let destructors throw . . . 673
42.7.3 One allocation per statement (clarity and robustness) . . . 673
42.8 Shared ownership: avoiding reference-count cycles . . . 673
42.8.1 The rule . . . 674
42.8.2 Example . . . 674
42.9 Thread and lock “leaks” . . . 674
42.9.1 Lock leaks . . . 674
42.9.2 Thread management leaks . . . 674
42.10 Detecting leaks: build configuration and tooling . . . 675
42.10.1 Design makes detection easier . . . 675
42.11 A practical “no leaks” checklist . . . 675
42.12 Summary . . . 676

43 Non-Memory Resources 677
43.1 Introduction . . . 677
43.2 Core Guidelines principles for non-memory resources . . . 678
43.3 Resource Acquisition Is Initialization (RAII) for non-memory resources . . . 678
43.3.1 RAII overview . . . 679
43.3.2 RAII guarantees . . . 679
43.4 Common non-memory resources in C++ systems . . . 679
43.4.1 File handles . . . 679
43.4.2 Network sockets . . . 679

43.4.3 Mutexes and locks . . . 680
43.4.4 Threads and thread handles . . . 680
43.4.5 Database connections and transactions . . . 680
43.5 Designing RAII wrappers for non-memory resources . . . 680
43.5.1 Essential wrapper properties . . . 681
43.6 RAII example: file handle . . . 681
43.6.1 POSIX file descriptor wrapper . . . 681
43.6.2 Usage pattern with RAII . . . 683
43.7 RAII example: network socket . . . 683
43.7.1 Socket wrapper . . . 683
43.7.2 Usage pattern . . . 685
43.8 RAII for synchronization primitives . . . 685
43.8.1 Standard library support . . . 685
43.8.2 Example: scoped lock . . . 685
43.9 Thread resources . . . 686
43.9.1 Thread as a resource . . . 686
43.9.2 RAII thread management . . . 686
43.10 Database connections and transactions . . . 686
43.10.1 Connection handle RAII . . . 687
43.10.2 Transaction scope guard . . . 687
43.11 Guidelines for non-memory resource APIs . . . 687
43.11.1 Make ownership explicit . . . 687
43.11.2 Prefer RAII handles in public interfaces . . . 688
43.11.3 Do not expose raw handles if not necessary . . . 688
43.12 Avoiding common pitfalls . . . 688
43.12.1 Pitfall: missing cleanup on error paths . . . 688
43.12.2 Pitfall: unintended copy of resources . . . 688

43.12.3 Pitfall: forgetting to join or detach threads . . . 688
43.12.4 Pitfall: inconsistent lock release . . . 688
43.13 Checking correctness and leaks . . . 689
43.13.1 Static analysis . . . 689
43.13.2 Dynamic leak detection . . . 689
43.14 Summary . . . 689

VII Pointers and References (P) 690

44 P.1 – P.12: Pointers and References (Guiding Principles) 692
44.1 Scope and intent . . . 692
44.2 P.1: Express ideas directly in code . . . 693
44.2.1 Rule . . . 693
44.2.2 What this means for pointers and references . . . 693
44.2.3 Examples . . . 693
44.3 P.2: Write in ISO Standard C++ . . . 694
44.3.1 Rule . . . 694
44.3.2 What this means for pointers and references . . . 694
44.3.3 Example: encapsulate a platform handle . . . 694
44.4 P.3: Express intent . . . 695
44.4.1 Rule . . . 695
44.4.2 Recommended intent patterns . . . 695
44.4.3 Example . . . 696
44.5 P.4: Ideally, a program should be statically type safe . . . 696
44.5.1 Rule . . . 696
44.5.2 What this means for pointers and references . . . 696
44.5.3 Example . . . 697

44.6 P.5: Prefer compile-time checking to run-time checking . . . 697
44.6.1 Rule . . . 697
44.6.2 What this means for pointers and references . . . 697
44.6.3 Examples . . . 698
44.7 P.6: What cannot be checked at compile time should be checkable at run time 698
44.7.1 Rule . . . 698
44.7.2 What this means for pointers and references . . . 698
44.7.3 Example: explicit checks for nullable pointers . . . 699
44.8 P.7: Catch run-time errors early . . . 699
44.8.1 Rule . . . 699
44.8.2 Pointer/reference guidance . . . 699
44.8.3 Example: fail fast on invalid ranges . . . 699
44.9 P.8: Don't leak any resources . . . 700
44.9.1 Rule . . . 700
44.9.2 Pointer/reference-specific guidance . . . 700
44.9.3 Example . . . 700
44.10 P.9: Don't waste time or space . . . 700
44.10.1 Rule . . . 700
44.10.2 Balanced guidance . . . 701
44.10.3 Example: avoid accidental shared ownership . . . 701
44.11 P.10: Prefer immutable data to mutable data . . . 701
44.11.1 Rule . . . 701
44.11.2 Guidance . . . 701
44.11.3 Examples . . . 702
44.12 P.11: Encapsulate messy constructs, rather than spreading through the code . 702
44.12.1 Rule . . . 702
44.12.2 What counts as "messy" . . . 702

44.12.3 How to apply . . . 703
44.12.4 Example: encapsulate pointer arithmetic behind a view . . . 703
44.13 P.12: Use supporting tools as appropriate . . . 703
44.13.1 Rule . . . 703
44.13.2 Tools that materially help pointer/reference safety . . . 703
44.13.3 Practical workflow . . . 704
44.14 Consolidated checklist (P.1–P.12 applied) . . . 704

45 Raw Pointers (`T*`) 705
45.1 What a raw pointer is (and is not) . . . 705
45.1.1 Definition . . . 705
45.1.2 The most important rule . . . 705
45.1.3 What raw pointers are good at . . . 706
45.1.4 What raw pointers are bad at . . . 706
45.2 Raw pointer meaning in interfaces . . . 706
45.2.1 Single object vs sequence . . . 707
45.2.2 Nullability contracts . . . 707
45.3 Ownership: raw pointers must not silently own . . . 708
45.3.1 The rule . . . 708
45.3.2 Bad and good patterns . . . 708
45.3.3 Owning raw pointers at boundaries . . . 708
45.4 Lifetime safety: avoiding dangling pointers . . . 709
45.4.1 How dangling happens . . . 709
45.4.2 Classic error: returning address of a local . . . 709
45.4.3 Safe alternatives . . . 710
45.5 Bounds safety: raw pointers are not ranges . . . 710
45.5.1 Why raw pointers are risky for sequences . . . 710
45.5.2 Prefer `std::span` for contiguous sequences . . . 710

45.5.3 If you must use pointer+size (interop boundary) . . . 711
45.6 Const correctness with pointers . . . 711
45.6.1 Two different meanings of const . . . 711
45.6.2 Guideline practice . . . 712
45.7 Polymorphism and deletion through base pointers . . . 712
45.7.1 The rule . . . 712
45.7.2 Guideline reminder . . . 712
45.8 Raw pointers and modern C++ safety tooling . . . 712
45.9 Recommended patterns (quick reference) . . . 713
45.9.1 Use raw pointers for . . . 713
45.9.2 Avoid raw pointers for . . . 713
45.10 Checklist: safe raw pointer usage . . . 713
45.11 Summary . . . 714
45.12 Introduction . . . 714
45.13 What a reference is . . . 714
45.13.1 Definition . . . 715
45.13.2 Key properties . . . 715
45.14 Why references are safer than pointers . . . 715
45.15 Lvalue references (`T&`) . . . 716
45.15.1 Definition and typical use . . . 716
45.15.2 Function parameters and return values . . . 716
45.15.3 Const lvalue references (`const T&`) . . . 716
45.16 Rvalue references (`T&&`) . . . 717
45.16.1 Definition and purpose . . . 717
45.16.2 Move semantics example . . . 717
45.16.3 Perfect forwarding with templates . . . 717
45.17 When to use references (Core Guidelines perspective) . . . 718

45.18 References vs pointers . . . 718
45.18.1 Nullability . . . 718
45.18.2 Ownership and lifetime . . . 718
45.18.3 Syntax differences . . . 719
45.19 Common reference pitfalls . . . 719
45.19.1 Dangling references . . . 719
45.19.2 Binding to temporary objects . . . 719
45.20 Reference collapsing rules . . . 719
45.21 Function overloading with references . . . 720
45.22 References and object lifetime . . . 720
45.22.1 Safe lifetime usage . . . 720
45.22.2 References to data in containers . . . 720
45.23 References in modern API design . . . 721
45.23.1 Required parameters . . . 721
45.23.2 Optional parameters . . . 721
45.24 Summary . . . 721

46 Pointers as Observers (Non-owning Use) 722
46.1 Core idea . . . 722
46.2 Observer vs owner (precise definitions) . . . 722
46.2.1 Observer . . . 722
46.2.2 Owner . . . 723
46.3 Core Guidelines rules behind “pointers as observers” . . . 723
46.4 Why treating pointers as observers matters . . . 724
46.4.1 Avoids ownership ambiguity . . . 724
46.4.2 Improves exception safety . . . 724
46.4.3 Enables robust tooling . . . 724
46.5 Observer pointers in interfaces (recommended patterns) . . . 725

46.5.1 Required vs optional single objects . . . 725
46.5.2 Sequences: use `std::span` as an observer . . . 725
46.5.3 Text: use `std::string_view` as an observer . . . 726
46.6 Owning objects should not be exposed as raw pointers . . . 726
46.6.1 Do not return owning raw pointers . . . 726
46.6.2 Return owners or values . . . 726
46.6.3 If you must accept a legacy owning raw pointer . . . 727
46.7 Lifetime rules for observer pointers . . . 727
46.7.1 The fundamental constraint . . . 727
46.7.2 Common ways observers dangle . . . 728
46.7.3 Classic example: dangling pointer . . . 728
46.7.4 Safe alternatives . . . 728
46.8 Nullability and observer pointers . . . 729
46.8.1 Use pointers when null is meaningful . . . 729
46.8.2 Use references when null must not occur . . . 729
46.8.3 Non-null pointer contracts (project vocabulary) . . . 729
46.9 Observer pointers and const correctness . . . 729
46.9.1 Default to read-only observation . . . 729
46.9.2 Make mutation intentional . . . 730
46.10 Observer pointers in class design . . . 730
46.10.1 Storing observer pointers requires a lifetime guarantee . . . 730
46.10.2 Prefer owning members when lifetime is complex . . . 730
46.11 A compact checklist: pointer-as-observer discipline . . . 730
46.12 Summary . . . 731

47 Avoiding Dangling Pointers 732
47.1 Introduction . . . 732
47.2 What is a dangling pointer . . . 732

47.2.1 Precise definition . . . 732
47.2.2 Common scenarios that cause dangling . . . 733
47.3 Why dangling pointers are dangerous . . . 733
47.4 Guidelines for preventing dangling pointers . . . 734
47.5 Lifetime rules for pointers and references . . . 734
47.5.1 Object lifetime vs pointer lifetime . . . 734
47.5.2 Local variables . . . 734
47.5.3 Safe alternatives . . . 735
47.6 Containers and iterator invalidation . . . 735
47.6.1 Dynamic array containers (vector, string) . . . 736
47.6.2 Associative containers (set, map) . . . 736
47.6.3 Guideline: prefer safe views or indices . . . 736
47.7 Temporary objects and views . . . 736
47.7.1 Pointers/references to temporaries . . . 737
47.7.2 Views to temporaries . . . 737
47.7.3 Safe alternatives . . . 737
47.8 Observer pointers and references in class design . . . 737
47.8.1 Storing observers as members . . . 738
47.8.2 Prefer owning members when lifetime is unclear . . . 738
47.9 Smart pointers and dangling avoidance . . . 738
47.9.1 Non-owning raw pointers vs owning smart pointers . . . 738
47.9.2 Using `std::weak_ptr` to avoid dangling . . . 738
47.10 Dangling and concurrency . . . 739
47.10.1 Data races and lifetime . . . 739
47.10.2 Guideline: synchronize or extend lifetime . . . 739
47.11 Compiler and tooling assistance . . . 740
47.11.1 Compile-time checks . . . 740

47.11.2 Static analysis and sanitizers . . . 740
47.12 Checklist for avoiding dangling pointers . . . 740
47.13 Summary . . . 741

48 Nullability 742
48.1 Introduction . . . 742
48.2 Nullability basics . . . 742
48.2.1 What nullability means . . . 743
48.2.2 Why nullability matters . . . 743
48.3 Nullability in function parameters . . . 743
48.3.1 Required object (non-null) . . . 743
48.3.2 Optional object (nullable) . . . 744
48.3.3 Why not use references for optional . . . 744
48.4 Nullability and ownership . . . 744
48.4.1 Observers vs owners . . . 744
48.4.2 Owning smart pointers and nullability . . . 744
48.5 Representing nullability explicitly . . . 745
48.5.1 Raw pointers . . . 745
48.5.2 Optional reference wrappers . . . 745
48.6 Nullability and return values . . . 746
48.6.1 Returning optional objects . . . 746
48.6.2 Avoid returning null raw pointers if alternatives exist . . . 746
48.7 Nullability in class members . . . 746
48.7.1 Nullable observers . . . 746
48.7.2 Prefer safe alternatives when possible . . . 747
48.8 Nullability and initialization . . . 747
48.8.1 Pointers must be initialized explicitly . . . 747
48.8.2 References must be bound at initialization . . . 747

48.9 Nullability and `nullptr` . . . 748
48.9.1 Use `nullptr` instead of 0 . . . 748
48.10 Nullability and comparisons . . . 748
48.10.1 Check for null safely . . . 748
48.10.2 Prefer direct tests . . . 748
48.11 Nullability and contracts . . . 749
48.11.1 Document requirements . . . 749
48.11.2 Fail early on null violations . . . 749
48.12 Special cases: arrays and nullability . . . 749
48.12.1 Pointers to arrays . . . 749
48.12.2 Bounds safety . . . 750
48.13 Checklist for nullability . . . 750
48.14 Summary . . . 751

VIII Expressions and Statements (ES) 752

49 ES.1 – ES.114: Expressions and Statements 754
49.1 Editorial accuracy and scope . . . 754
49.2 Purpose of the ES rules . . . 755
49.3 ES.1 – ES.34: Declarations, naming, initialization, macros . . . 755
49.4 ES.35 – ES.69: Expressions, conversions, pointers, lifetimes . . . 760
49.5 ES.70 – ES.87: Control flow and statements . . . 764
49.6 ES.88 – ES.107: Errors, arithmetic, and correctness . . . 766
49.7 ES.108 – ES.114: Non-existent rule numbers . . . 769
49.8 Summary . . . 770

50 Safe Expressions 771
50.1 Purpose and scope . . . 771

50.2 Design principle: one idea per expression 772
50.2.1 Keep expressions simple 772
50.3 Sequencing and order of evaluation 772
50.3.1 Avoid expressions with unspecified or undefined sequencing 772
50.3.2 Function arguments: do not rely on evaluation order 773
50.4 Operator precedence and readability 773
50.4.1 Parenthesize when there is any doubt 773
50.4.2 Avoid cleverness with ?: and comma 774
50.5 Conversions and narrowing 774
50.5.1 Avoid narrowing conversions 774
50.5.2 Avoid mixing signed and unsigned arithmetic 775
50.6 Pointer validity and lifetime in expressions 776
50.6.1 Never dereference an invalid pointer 776
50.6.2 Do not produce dangling references or pointers 776
50.6.3 Prefer safe views for ranges 777
50.7 Casts: keep them rare and explicit 777
50.7.1 Avoid casts 777
50.7.2 If you must cast, use a named cast 778
50.7.3 Avoid casting away const 778
50.8 Arithmetic safety 778
50.8.1 Signed overflow and division by zero 778
50.8.2 Avoid unsigned wraparound as “math” 779
50.9 Assignments, side effects, and clarity 779
50.9.1 Avoid assignment inside conditions unless it is unmistakable 779
50.9.2 Avoid hidden side effects in expressions 779
50.10 Switch and fallthrough safety 780
50.10.1 Avoid implicit fallthrough 780

50.11 A practical “safe expression” checklist . . . 780
50.12 Summary . . . 781

51 Control Flow 782
51.1 Purpose and scope . . . 782
51.2 General principle: prefer structured, readable flow . . . 783
51.2.1 One entry, one exit when practical . . . 783
51.2.2 Prefer RAII to control-flow-based cleanup . . . 783
51.3 Selection statements . . . 783
51.3.1 Prefer `switch` over long `if/else if` chains (ES.70) . . . 784
51.3.2 Avoid implicit fallthrough in `switch` (ES.78) . . . 784
51.3.3 Use `default` carefully (ES.79) . . . 785
51.3.4 `if` with initializer (modern C++) . . . 785
51.4 Iteration statements . . . 785
51.4.1 Prefer range-for for simple iteration (ES.71) . . . 785
51.4.2 Prefer `for` when there is an obvious loop variable (ES.72) . . . 786
51.4.3 Prefer `while` when there is no obvious loop variable (ES.73) . . . 786
51.4.4 Declare loop variables in the initializer (ES.74) . . . 786
51.4.5 Avoid `do/while` (ES.75) . . . 787
51.5 Exits, breaks, continues, and structure . . . 787
51.5.1 Avoid `goto` (ES.76) . . . 787
51.5.2 Minimize `break` and `continue` (ES.77) . . . 787
51.5.3 Avoid modifying the loop control variable in the body (ES.86) . . . 788
51.5.4 Prefer simple conditions; avoid redundant comparisons (ES.87) . . . 788
51.6 Empty statements and intentional emptiness (ES.85) . . . 788
51.7 Nesting and complexity control . . . 788
51.7.1 Prefer flattening over deep nesting . . . 788
51.8 Control flow and correctness hazards . . . 789

51.8.1 Fallthrough hazards . . . 789
51.8.2 Loop termination and off-by-one . . . 789
51.8.3 Control flow and resource safety . . . 790
51.9 Control flow patterns (recommended) . . . 790
51.9.1 Guard clauses (early return) . . . 790
51.9.2 Centralized decision with `switch` . . . 790
51.9.3 Loop with explicit termination condition . . . 791
51.10 Checklist for control-flow reviews . . . 791
51.11 Summary . . . 791

52 `if` and `switch` 793
52.1 Purpose and guideline focus . . . 793
52.2 `if`: precise meaning and modern forms . . . 794
52.2.1 Basic `if`/`else` . . . 794
52.2.2 Make conditions safe and obvious . . . 794
52.2.3 `if` with initializer (C++17 and later) . . . 795
52.2.4 Guard clauses: early returns for clarity . . . 795
52.3 When `if` is the right tool . . . 796
52.4 `switch`: precise meaning and good usage . . . 796
52.4.1 What `switch` does . . . 797
52.4.2 Prefer `switch` for multi-way selection (ES.70) . . . 797
52.4.3 Avoid implicit fallthrough (ES.78) . . . 797
52.4.4 Grouping cases safely . . . 798
52.4.5 Use `default` carefully (ES.79) . . . 798
52.4.6 `switch` and initialization safety . . . 799
52.5 `if` vs `switch`: choosing correctly . . . 800
52.5.1 Prefer `switch` when . . . 800
52.5.2 Prefer `if` when . . . 800

52.6 Common pitfalls and safe rewrites . . . 800
52.6.1 Pitfall: long if-else chains for discrete values . . . 801
52.6.2 Pitfall: missing break . . . 801
52.6.3 Pitfall: default hides missing enum cases . . . 801
52.7 Review checklist . . . 802
52.8 Summary . . . 802

53 Loops 803
53.1 Purpose and guideline focus . . . 803
53.2 Choose the right loop form . . . 804
53.2.1 Prefer range-for for iterating over a range (ES.71) . . . 804
53.2.2 Prefer `for` when there is an obvious loop variable (ES.72) . . . 804
53.2.3 Prefer `while` when there is no obvious loop variable (ES.73) . . . 805
53.2.4 Declare loop variables in the initializer (ES.74) . . . 805
53.2.5 Avoid `do/while` unless it is the clearest (ES.75) . . . 805
53.3 Prefer algorithms when possible (ES.1, ES.2) . . . 805
53.4 Bounds, indices, and signedness safety . . . 806
53.4.1 Prefer half-open ranges . . . 806
53.4.2 Avoid unsigned indices for subscripts (ES.107) . . . 806
53.4.3 Avoid mixing signed and unsigned arithmetic (ES.100) . . . 807
53.5 Pointer/iterator validity inside loops . . . 807
53.5.1 Do not dereference invalid pointers (ES.65) or invalid iterators . . . 807
53.5.2 Prefer range-for or indices that are re-derived when needed . . . 808
53.6 Loop exits and control-flow clarity . . . 808
53.6.1 Minimize `break` and `continue` (ES.77) . . . 808
53.6.2 Avoid `goto` (ES.76); use structured exits . . . 808
53.6.3 Avoid modifying the loop control variable inside the body (ES.86) . . . 809
53.7 Empty loop bodies (ES.85) . . . 809

53.8 Performance without sacrificing safety . . . 810
53.8.1 Avoid unnecessary work in the condition . . . 810
53.8.2 Prefer preincrement for iterators when appropriate . . . 810
53.9 Loop patterns that are safe by construction . . . 810
53.9.1 Counting loop . . . 810
53.9.2 Range loop . . . 810
53.9.3 Sentinel loop . . . 810
53.9.4 Search loop with early return . . . 811
53.10 Review checklist for loops . . . 811
53.11 Summary . . . 811

54 Implicit Conversions 813
54.1 Purpose and guideline focus . . . 813
54.2 What counts as an implicit conversion . . . 814
54.3 Why implicit conversions are risky . . . 814
54.3.1 They can silently change values . . . 814
54.3.2 They can change overload selection . . . 815
54.3.3 They can make code ambiguous or fragile . . . 816
54.4 Core Guidelines: controlling implicit conversions . . . 816
54.4.1 Avoid narrowing conversions (ES.46, ES.64) . . . 816
54.4.2 Prefer explicit conversions when the meaning matters . . . 816
54.4.3 Avoid mixing signed and unsigned arithmetic (ES.100–ES.107) . . . 816
54.4.4 Avoid implicit pointer conversions that hide ownership or lifetime . . . 817
54.5 Safe numeric conversion patterns . . . 817
54.5.1 Prefer widening conversions . . . 817
54.5.2 For potentially narrowing conversions, check . . . 818
54.5.3 Prefer `std::byte` for raw byte data . . . 818
54.6 Implicit conversions in boolean contexts . . . 818

54.6.1 Prefer clear boolean expressions . . . 818
54.7 User-defined implicit conversions (type design) . . . 819
54.7.1 Converting constructors . . . 819
54.7.2 Conversion operators . . . 820
54.7.3 Avoid implicit conversion to bool for “smart” types unless it is the established pattern . . . 820
54.8 Implicit conversions and overload sets (API design) . . . 820
54.8.1 Avoid overloads that differ only by signedness or narrow types . . . 820
54.8.2 Prefer strongly typed enums over integers . . . 821
54.9 When implicit conversions are acceptable . . . 821
54.10 Review checklist . . . 821
54.11 Summary . . . 822

55 Operator Precedence 823
55.1 Purpose and guideline focus . . . 823
55.2 Precedence vs associativity (precise meaning) . . . 824
55.2.1 Precedence . . . 824
55.2.2 Associativity . . . 824
55.2.3 Precedence is not evaluation order . . . 824
55.3 Core guideline rule (ES.41) . . . 824
55.3.1 ES.41: If in doubt about operator precedence, parenthesize . . . 824
55.4 A practical precedence map (most-used operators) . . . 825
55.5 The precedence mistakes that cause real bugs . . . 826
55.5.1 Bitwise vs equality (classic trap) . . . 826
55.5.2 Shift vs addition . . . 826
55.5.3 Logical vs bitwise operators . . . 827
55.5.4 Assignment inside conditions . . . 827
55.5.5 The conditional operator ?: . . . 828

55.5.6 Comma operator . . . 828
55.6 Safe patterns recommended by the guidelines . . . 828
55.6.1 Parenthesize bit tests and masks . . . 828
55.6.2 Name intermediate results . . . 828
55.6.3 Avoid clever expressions with side effects . . . 829
55.7 Precedence and overload resolution (brief note) . . . 829
55.8 Review checklist . . . 829
55.9 Summary . . . 830

IX Naming and Layout (NL) 831

56 NL.1 – NL.26: Naming and Layout 833
56.1 Scope and intent of the NL rules . . . 833
56.2 Rule index (current guideline revision) . . . 833
56.3 Comments: clarity without noise . . . 834
56.3.1 NL.1: Don't say in comments what can be clearly stated in code . . . 834
56.3.2 NL.2: State intent in comments . . . 834
56.3.3 NL.3: Keep comments crisp . . . 834
56.4 Indentation and visual structure . . . 835
56.4.1 NL.4: Maintain a consistent indentation style . . . 835
56.5 Naming: express meaning, not types . . . 835
56.5.1 NL.5: Avoid encoding type information in names . . . 835
56.5.2 NL.6: (Not present in the July 8, 2025 revision) . . . 836
56.5.3 NL.7: Make the length of a name roughly proportional to the length of its scope . . . 836
56.5.4 NL.8: Use a consistent naming style . . . 836
56.5.5 NL.9: Use ALL_CAPS for macro names only . . . 837

56.5.6 NL.10: Prefer `underscore_style` names . . . 837
56.5.7 NL.11: Make literals readable . . . 837
56.5.8 NL.12: (Not present in the July 8, 2025 revision) . . . 838
56.5.9 NL.13: (Not present in the July 8, 2025 revision) . . . 838
56.5.10 NL.14: (Not present in the July 8, 2025 revision) . . . 838
56.6 Whitespace, declarations, and layout defaults . . . 838
56.6.1 NL.15: Use spaces sparingly . . . 838
56.6.2 NL.16: Use a conventional class member declaration order . . . 839
56.6.3 NL.17: Use K&R-derived layout . . . 840
56.6.4 NL.18: Use C++-style declarator layout . . . 840
56.6.5 NL.19: Avoid names that are easily misread . . . 840
56.6.6 NL.20: Don't place two statements on the same line . . . 841
56.6.7 NL.21: Declare one name (only) per declaration . . . 841
56.6.8 NL.22: (Not present in the July 8, 2025 revision) . . . 842
56.6.9 NL.23: (Not present in the July 8, 2025 revision) . . . 842
56.6.10 NL.24: (Not present in the July 8, 2025 revision) . . . 842
56.7 Interface notation and compatibility reminders . . . 842
56.7.1 NL.25: Don't use `void` as an argument type . . . 842
56.7.2 NL.26: Use conventional `const` notation . . . 842
56.8 Practical adoption notes . . . 843
56.9 Summary . . . 843

57 Naming Rules 844
57.1 Scope and intent . . . 844
57.2 Naming principles . . . 845
57.2.1 Name meaning, not mechanics . . . 845
57.2.2 Consistency beats personal preference . . . 845
57.3 Core Guidelines naming rules (NL) . . . 845

57.3.1 NL.5: Avoid encoding type information in names 845
57.3.2 NL.7: Make the length of a name roughly proportional to the length of its scope 846
57.3.3 NL.8: Use a consistent naming style 846
57.3.4 NL.9: Use `ALL_CAPS` for macro names only 847
57.3.5 NL.10: Prefer `underscore_style` names 847
57.3.6 NL.11: Make literals readable 847
57.3.7 NL.19: Avoid names that are easily misread 848
57.4 Practical naming rules for modern C++ codebases 848
57.4.1 Types, functions, and variables 848
57.4.2 Avoid misleading abbreviations 849
57.4.3 Do not use reserved or dangerous identifier forms 850
57.4.4 Name ownership and lifetime explicitly in interfaces 850
57.5 Naming and overload sets (avoid confusion) 851
57.5.1 Prefer overload sets that preserve meaning 851
57.6 Checklist for naming reviews 851
57.7 Summary 852

58 Consistency 853
58.1 Purpose and guideline focus 853
58.2 What “consistency” means in practice 854
58.3 Core Guidelines rules that directly enforce consistency 854
58.3.1 NL.4: Maintain a consistent indentation style 854
58.3.2 NL.8: Use a consistent naming style 855
58.3.3 NL.10: Prefer underscore_style names (default suggestion) 855
58.3.4 NL.17: Use K&R-derived layout (default suggestion) 855
58.3.5 NL.26: Use conventional const notation (default suggestion) 856
58.4 Consistency and correctness: where it prevents bugs 856

58.4.1 Consistent bracing prevents “dangling else” confusion 856
58.4.2 Consistent naming reduces misuse 856
58.4.3 Consistent units in names prevent dangerous mistakes 857
58.5 Consistency at interfaces: the most important place 857
58.5.1 Consistent ownership vocabulary 857
58.5.2 Consistent error-handling strategy 858
58.6 Consistency and refactoring safety 858
58.7 How to achieve consistency in a real codebase 858
58.7.1 Step 1: Choose a small, enforceable house style 858
58.7.2 Step 2: Automate formatting 859
58.7.3 Step 3: Automate naming and rule checks where valuable 859
58.7.4 Step 4: Fix inconsistencies at boundaries 859
58.8 When consistency should yield 860
58.9 Checklist 860
58.10 Summary 861

59 Layout and Formatting 862
59.1 Purpose and guideline focus 862
59.2 Indentation and block structure 863
59.2.1 NL.4: Maintain a consistent indentation style 863
59.3 Whitespace and spacing 864
59.3.1 NL.15: Use spaces sparingly 864
59.3.2 Recommended spacing conventions 865
59.4 Brace style (layout of blocks) 865
59.4.1 NL.17: Use K&R-derived layout 865
59.5 Declarations and declarators 866
59.5.1 NL.21: Declare one name (only) per declaration 866
59.5.2 NL.18: Use C++-style declarator layout 866

59.5.3 Pointers and references: avoid confusing formatting . . . 866
59.6 Statements per line . . . 867
59.6.1 NL.20: Don’t place two statements on the same line . . . 867
59.7 Class layout and member ordering . . . 867
59.7.1 NL.16: Use a conventional class member declaration order . . . 867
59.8 Line length and wrapping . . . 868
59.9 Formatting as an automated decision . . . 869
59.9.1 Use tools to enforce formatting . . . 869
59.9.2 Format boundaries: do not reformat third-party code . . . 869
59.10 Checklist for layout and formatting reviews . . . 870
59.11 Summary . . . 870

60 Readability **871**
60.1 Purpose and scope . . . 871
60.2 Readability principle: make intent obvious . . . 872
60.3 Comments that improve readability . . . 872
60.3.1 NL.1: Don’t say in comments what can be clearly stated in code . . . 872
60.3.2 NL.2: State intent in comments . . . 873
60.3.3 NL.3: Keep comments crisp . . . 873
60.4 Structure and visual clarity . . . 873
60.4.1 NL.4: Maintain consistent indentation style . . . 873
60.4.2 NL.17: Use K&R-derived layout as a default . . . 874
60.4.3 NL.20: Don’t place two statements on the same line . . . 874
60.5 Naming for readable code . . . 874
60.5.1 NL.5: Avoid encoding type information in names . . . 874
60.5.2 NL.7: Name length proportional to scope . . . 875
60.5.3 NL.8: Use a consistent naming style . . . 875
60.5.4 NL.11: Make literals readable . . . 875

60.5.5 NL.19: Avoid names that are easily misread . . . 876
60.6 Declaration readability (avoid syntactic traps) . . . 876
60.6.1 NL.21: Declare one name per declaration . . . 876
60.6.2 NL.18: Use conventional declarator layout . . . 876
60.7 Readability patterns that scale . . . 877
60.7.1 Prefer small, named steps over clever expressions . . . 877
60.7.2 Guard clauses reduce nesting . . . 877
60.7.3 Prefer standard algorithms when they express intent . . . 878
60.8 Readability and safety: never trade one for the other . . . 878
60.9 Checklist for readability reviews . . . 878
60.10 Summary . . . 879

61 Avoiding Ambiguity 880
61.1 Purpose and guideline focus . . . 880
61.2 Kinds of ambiguity in C++ . . . 881
61.2.1 Lexical and visual ambiguity . . . 881
61.2.2 Syntactic ambiguity . . . 881
61.2.3 Semantic ambiguity . . . 882
61.3 Avoid ambiguous names (NL.19 and related guidance) . . . 882
61.3.1 NL.19: Avoid names that are easily misread . . . 882
61.3.2 Avoid ambiguous abbreviations . . . 883
61.3.3 Avoid type-encoded names (NL.5) to prevent semantic ambiguity . . . 883
61.4 Avoid ambiguous structure (indentation, braces, statements) . . . 883
61.4.1 NL.4: Maintain consistent indentation style . . . 883
61.4.2 Avoid the “dangling else” ambiguity . . . 884
61.4.3 NL.20: Don’t place two statements on the same line . . . 884
61.4.4 NL.15: Use spaces sparingly (avoid token ambiguity) . . . 885
61.5 Avoid ambiguous declarations (NL.21 and NL.18) . . . 885

61.5.1 NL.21: Declare one name per declaration . . . 885
61.5.2 NL.18: Use conventional declarator layout . . . 886
61.5.3 Prefer type aliases for complex declarators . . . 886
61.6 Avoid ambiguous expressions (related ES guidance) . . . 886
61.6.1 Operator precedence ambiguity (ES.41) . . . 886
61.6.2 Implicit conversion ambiguity (ES conversion rules) . . . 887
61.6.3 Switch fallthrough ambiguity (ES.78) . . . 887
61.7 Avoid ambiguous APIs (naming and overload sets) . . . 888
61.7.1 Use names that preserve meaning across overloads . . . 888
61.7.2 Avoid overload sets that invite surprising conversions . . . 888
61.8 Ambiguity in comments . . . 888
61.8.1 NL.2 and NL.3: State intent crisply . . . 889
61.9 Checklist for avoiding ambiguity . . . 889
61.10 Summary . . . 890

X Templates and Metaprogramming (T) 891

62 T.1–T.83: Templates and Metaprogramming 893
62.1 Purpose of this chapter . . . 893
62.2 T.1–T.9: Generic programming fundamentals . . . 894
62.3 T.10–T.19: Using concepts and constraints . . . 895
62.4 T.20–T.29: Defining concepts . . . 896
62.5 T.30–T.39: Advanced concept composition . . . 898
62.6 T.40–T.49: Template interfaces . . . 899
62.7 T.50–T.59: (Reserved range) . . . 900
62.8 T.60–T.69: Template definitions and implementation . . . 902
62.9 T.70–T.79: (Reserved range) . . . 903

62.10 T.80–T.83: Templates and class hierarchies 905
62.11 Summary 905

63 Template Design 906
63.1 Purpose and scope 906
63.2 Design goals for templates 907
63.2.1 Goal 1: Express requirements explicitly (concepts) 907
63.2.2 Goal 2: Require only what you need (avoid over-constraints) 907
63.2.3 Goal 3: Preserve meaning across instantiations 908
63.3 Designing template requirements (concepts) 908
63.3.1 Use standard concepts when possible 908
63.3.2 Define concepts by use patterns, not by member names 909
63.3.3 Document semantic laws (axioms) 909
63.4 Template interface design 910
63.4.1 Prefer clean interfaces: types communicate intent 910
63.4.2 Prefer function objects (operations as parameters) 911
63.4.3 Prefer aliases to simplify template-heavy types 911
63.5 Type deduction and usability 911
63.5.1 Support deduction via helpers or CTAD when it improves clarity . . . 911
63.5.2 Make parameter passing rules consistent 912
63.6 Customization points and ADL safety 913
63.6.1 Avoid accidental ADL (Argument-Dependent Lookup) in templates . 913
63.6.2 If you want customization, design it explicitly 913
63.7 Specialization, tag dispatch, and alternatives 914
63.7.1 Prefer overloads and constraints over specialization for functions . . . 914
63.7.2 Use class template specialization for true representation differences . 915
63.8 Avoiding common template design traps 915
63.8.1 Avoid over-parameterization (compile-time cost and fragility) 915

63.8.2 Minimize context dependencies . . . 915
63.8.3 Prefer {} initialization in templates to reduce ambiguity . . . 916
63.8.4 Keep error messages humane . . . 916
63.9 Templates and class hierarchies . . . 917
63.9.1 Do not naively templatize a hierarchy . . . 917
63.9.2 Do not attempt virtual member function templates . . . 917
63.10 Practical template design checklist . . . 917
63.11 Summary . . . 918

64 Specialization 919
64.1 Purpose and scope . . . 919
64.2 Specialization fundamentals (precise language rules) . . . 920
64.2.1 Primary template vs specialization . . . 920
64.2.2 Explicit vs partial specialization . . . 920
64.2.3 Function templates: no partial specialization . . . 921
64.3 Why and when to specialize (design criteria) . . . 922
64.4 Guidelines: specialization-related T rules . . . 922
64.4.1 T.64: Use specialization to provide alternative implementations of class templates . . . 922
64.4.2 T.67: Use specialization to provide alternative implementations for irregular types . . . 923
64.4.3 T.65: Use tag dispatch to provide alternative implementations of functions . . . 923
64.5 Specialization patterns (recommended and safe) . . . 923
64.5.1 Pattern 1: Traits-based specialization (classic and still useful) . . . 923
64.5.2 Pattern 2: Partial specialization for classification . . . 924
64.5.3 Pattern 3: “policy” specialization for representation differences . . . 924
64.5.4 Pattern 4: Tag dispatch as the function alternative (T.65) . . . 925

64.5.5 Pattern 5: Constrained overloads (modern default with concepts) . . . 926
64.6 Where specialization becomes dangerous . . . 926
64.6.1 Surprising behavior and hidden coupling . . . 926
64.6.2 Overlapping partial specializations . . . 927
64.6.3 Specializing in the wrong namespace . . . 927
64.6.4 Specializing standard library templates . . . 927
64.6.5 ABI and ODR risks . . . 928
64.7 Choosing among specialization, tag dispatch, and `if constexpr` . . . 928
64.8 Checklist for specialization reviews . . . 929
64.9 Summary . . . 929

65 SFINAE 931
65.1 Purpose and scope . . . 931
65.2 Definition: what SFINAE is . . . 932
65.2.1 Core idea . . . 932
65.3 Where SFINAE applies (and where it does not) . . . 932
65.3.1 Immediate context . . . 932
65.3.2 Not-SFINAE: errors outside the immediate context . . . 933
65.4 SFINAE and overload resolution . . . 933
65.4.1 A minimal working example (member detection) . . . 933
65.4.2 Conditional overload enablement with `std::enable_if` . . . 934
65.5 Expression SFINAE (the modern core technique) . . . 935
65.5.1 The comma operator trick and `decltype` . . . 935
65.6 The detection idiom and `std::void_t` . . . 935
65.6.1 The canonical detection pattern . . . 935
65.6.2 Why `void_t` matters . . . 936
65.7 SFINAE and partial specialization . . . 936
65.7.1 Class templates can be partially specialized . . . 936

65.7.2 Function templates cannot be partially specialized 937
65.8 Common SFINAE pitfalls 937
65.8.1 Pitfall 1: SFINAE that leaks into unreadable interfaces 937
65.8.2 Pitfall 2: Unexpected ambiguity between viable overloads 938
65.8.3 Pitfall 3: Negation-heavy SFINAE 938
65.8.4 Pitfall 4: ADL surprises and unqualified calls inside templates 938
65.8.5 Pitfall 5: Hard errors mistaken for SFINAE 938
65.9 Modern replacement: constraints and `requires` 938
65.9.1 Why concepts and requires are preferred 938
65.9.2 Rewrite example: SFINAE to `requires` 939
65.10 Choosing between SFINAE techniques 939
65.10.1 A practical decision guide 939
65.11 Recommended patterns (clean and maintainable) 940
65.11.1 Pattern 1: `if constexpr` with a trait 940
65.11.2 Pattern 2: Constrained overloads for interface clarity 940
65.12 Checklist 941
65.13 Summary 941

66 Constraints **943**
66.1 Purpose and scope 943
66.2 Terminology 944
66.2.1 Constraint, constraint-expression, constrained template 944
66.2.2 Concept 944
66.2.3 Requires-clause vs requires-expression 944
66.3 Why constraints matter (engineering benefits) 945
66.3.1 Early rejection: fail fast and locally 945
66.3.2 Better overload selection 945
66.3.3 Readable contracts 945

66.4 Forms of constraints in C++ . . . 945
66.4.1 Constrained template parameters . . . 945
66.4.2 Abbreviated function templates (constrained `auto`) . . . 946
66.4.3 Requires-clause on a template/function . . . 946
66.4.4 Requires-clause on a member function . . . 946
66.4.5 Requires-expression (checking operations) . . . 947
66.5 Designing good constraints (Core Guidelines aligned) . . . 947
66.5.1 State requirements for every template parameter (T.10) . . . 947
66.5.2 Use standard concepts whenever possible (T.11) . . . 948
66.5.3 Require only what you need (T.41) . . . 948
66.5.4 Prefer meaningful concepts with semantics (T.20–T.22) . . . 949
66.5.5 Avoid negation and disjunction-heavy constraints (T.30–T.31) . . . 949
66.6 Constraint satisfaction and overload resolution (practical model) . . . 950
66.6.1 Viability . . . 950
66.6.2 More constrained overloads are preferred (subsumption) . . . 950
66.7 Common constraint patterns . . . 951
66.7.1 Pattern 1: Constrain a function template parameter . . . 951
66.7.2 Pattern 2: Constrain a class template and its operations . . . 951
66.7.3 Pattern 3: Requires-expression for structural requirements . . . 952
66.7.4 Pattern 4: Internal selection with `if constexpr` . . . 952
66.8 Migrating from SFINAE to constraints (recommended approach) . . . 952
66.8.1 Example rewrite . . . 953
66.9 Pitfalls and how to avoid them . . . 953
66.9.1 Pitfall 1: Over-constraining public interfaces . . . 953
66.9.2 Pitfall 2: “syntactic” concepts with no semantic meaning . . . 953
66.9.3 Pitfall 3: Negation/disjunction producing fragile overload sets . . . 954
66.9.4 Pitfall 4: Accidental ADL customization points . . . 954

66.10 Checklist for constraint reviews . . . 954
66.11 Summary . . . 954

67 Concepts 956
67.1 Purpose and scope . . . 956
67.2 What a concept is . . . 957
67.2.1 Definition . . . 957
67.2.2 Concepts are part of the interface . . . 957
67.3 Core syntax and forms . . . 958
67.3.1 Constrained template parameters . . . 958
67.3.2 Abbreviated function templates . . . 958
67.3.3 Requires-clause using concepts . . . 958
67.3.4 Requires-expression (defining a concept by use patterns) . . . 959
67.4 Prefer standard concepts (T.11) . . . 959
67.5 Designing good concepts (T.20–T.26) . . . 960
67.5.1 Concepts should be meaningful (T.20) . . . 960
67.5.2 Require a complete set of operations (T.21) . . . 960
67.5.3 Specify axioms (semantic laws) when required (T.22) . . . 960
67.5.4 Refinement should add capability (T.23) . . . 961
67.5.5 Separate semantics when operations are identical (T.24) . . . 961
67.5.6 Avoid complementary concepts as primary design tools (T.25) . . . 962
67.5.7 Prefer use-pattern concepts (T.26) . . . 962
67.6 Using concepts in template interfaces (T.10, T.12, T.13, T.41) . . . 962
67.6.1 T.10: Specify concepts for all template parameters . . . 962
67.6.2 T.12: Prefer concept names over `auto` when requirements matter . . . 963
67.6.3 T.13: Prefer abbreviated templates when they are simplest . . . 963
67.6.4 T.41: Require only essential properties . . . 964
67.7 Concept composition (T.30–T.31) . . . 964

67.7.1 Use negation sparingly (T.30) . . . 964
67.7.2 Use disjunction sparingly (T.31) . . . 964
67.8 Overload resolution and subsumption (practical understanding) . . . 965
67.9 Concepts vs SFINAE (migration guidance) . . . 965
67.9.1 Concepts are the modern replacement . . . 965
67.9.2 Rewrite example . . . 966
67.10 Pitfalls and best practices . . . 966
67.10.1 Pitfall 1: Concepts that check syntax but not meaning . . . 966
67.10.2 Pitfall 2: Over-constraining breaks interoperability . . . 966
67.10.3 Pitfall 3: Unstable overload sets with negation/disjunction . . . 966
67.10.4 Pitfall 4: Accidental customization points . . . 967
67.11 Checklist for concept-based designs . . . 967
67.12 Summary . . . 967

68 Reducing Template Error Complexity 969
68.1 Purpose and scope . . . 969
68.2 Root causes of complex template errors . . . 970
68.2.1 Missing requirements (no contract) . . . 970
68.2.2 Overly generic overload sets . . . 970
68.2.3 Accidental substitution and ADL surprises . . . 971
68.2.4 Type-level metaprogramming without clear boundaries . . . 971
68.3 Technique 1: Constrain templates at the interface (T.10) . . . 971
68.3.1 Use concepts/requires to fail at the call site . . . 971
68.3.2 Prefer standard concepts (T.11) . . . 972
68.4 Technique 2: Name requirements (avoid “giant boolean” constraints) . . . 972
68.5 Technique 3: Require only what you need (T.41) . . . 973
68.6 Technique 4: Keep overload sets small and constrained (T.47) . . . 973
68.6.1 Do not publish unconstrained templates with common names . . . 973

68.6.2 Use “detail” namespaces for helper templates . . . 974
68.7 Technique 5: Prefer constraints over SFINAE in interfaces (T.48) . . . 974
68.7.1 If SFINAE is required, isolate it and name it . . . 974
68.8 Technique 6: Use `static_assert` with a clear message (localize failure) . . . 975
68.9 Technique 7: Prefer `requires`-expressions for diagnostics in concepts . . . 976
68.10 Technique 8: Control ADL and unqualified calls (T.69) . . . 976
68.11 Technique 9: Minimize template context dependencies (T.60) . . . 977
68.12 Technique 10: Reduce instantiation depth and type noise . . . 977
68.12.1 Use aliases to shorten types . . . 978
68.12.2 Prefer simple metaprogramming patterns . . . 978
68.13 Technique 11: Provide readable customization points . . . 978
68.14 A practical “error complexity” checklist . . . 979
68.15 Summary . . . 979

XI Error Handling and Exceptions (E) 981

69 Error Handling and Exceptions (E.1–E.30) 983
69.1 Overview: what error handling must achieve . . . 983
69.2 Rule summary: E.1–E.30 and numbering gaps . . . 983
69.3 E.1: Develop an error-handling strategy early in a design . . . 984
69.4 E.2: Throw an exception to signal that a function can’t perform its assigned task 985
69.5 E.3: Use exceptions for error handling only . . . 987
69.6 E.4: Design your error-handling strategy around invariants . . . 987
69.7 E.5: Let a constructor establish an invariant, and throw if it cannot . . . 988
69.8 E.6: Use RAII to prevent leaks . . . 989
69.9 E.7: State your preconditions . . . 990
69.10 E.8: State your postconditions . . . 991

69.11 E.9–E.11: (Not defined in the current guideline revision) 991
69.12 E.12: Use `noexcept` when exiting a function because of a throw is impossible or unacceptable 992
69.13 E.13: Never throw while being the direct owner of an object 993
69.14 E.14: Use purpose-designed user-defined types as exceptions (not built-in types) 993
69.15 E.15: Throw by value, catch exceptions from a hierarchy by reference 994
69.16 E.16: Destructors, deallocation, swap, and exception type copy/move construction must never fail 995
69.17 E.17: Don't try to catch every exception in every function 996
69.18 E.18: Minimize the use of explicit `try/catch` 996
69.19 E.19: Use a `final_action` object to express cleanup if no suitable resource handle is available 997
69.20 E.20–E.24: (Not defined in the current guideline revision) 999
69.21 E.25: If you can't throw exceptions, simulate RAII for resource management . 999
69.22 E.26: If you can't throw exceptions, consider failing fast 1001
69.23 E.27: If you can't throw exceptions, use error codes systematically 1001
69.24 E.28: Avoid error handling based on global state (e.g., `errno`) 1002
69.25 E.29: (Not defined in the current guideline revision) 1003
69.26 E.30: Don't use exception specifications 1003
69.27 Chapter checklist (engineering practice) 1004

70 When to Use Exceptions 1005

70.1 Purpose and scope 1005
70.2 Key definition: what an exception should mean 1006
70.3 Use exceptions when these conditions hold 1006
70.3.1 1) The failure is not part of normal expected control flow (E.2, E.3) . 1006
70.3.2 2) The caller cannot reasonably continue without handling the failure 1006
70.3.3 3) You need to propagate failure across multiple layers cleanly 1006

70.3.4 4) A constructor cannot establish its invariant (E.5) 1006
70.3.5 5) Resource acquisition fails, but RAII ensures cleanup (E.6) 1007
70.4 Do not use exceptions in these cases 1008
70.4.1 1) To signal ordinary outcomes (E.3) 1008
70.4.2 2) In low-level code where exceptions are forbidden by policy 1008
70.4.3 3) In destructors and during stack unwinding (E.16) 1009
70.4.4 4) To replace precondition checking strategy 1009
70.4.5 5) In performance-critical inner loops when failures are frequent . . . 1010
70.5 A clear practical rule: exceptions vs normal returns 1010
70.5.1 Use exceptions when 1010
70.5.2 Prefer return-based results when 1010
70.6 How to write exception-based APIs correctly 1011
70.6.1 Throw types that communicate meaning (E.14) 1011
70.6.2 Throw by value, catch by reference (E.15) 1011
70.6.3 Mark functions `noexcept` when throwing is impossible or unacceptable (E.12) 1011
70.7 Where to catch exceptions (E.17, E.18) 1012
70.8 Exception safety and resource safety (E.4–E.6) 1013
70.9 No-exception builds: what to do instead (E.25–E.27) 1013
70.10 Checklist: deciding “exceptions or not” for an API 1013
70.11 Summary 1014

71 Exception Guarantees 1015
71.1 Purpose and scope 1015
71.2 Why guarantees matter: invariants and recovery 1016
71.2.1 Invariants (E.4) 1016
71.2.2 RAII (E.6) 1016
71.3 The four commonly used exception safety guarantees 1016

71.3.1 1) No-throw guarantee . . . 1016
71.3.2 2) Strong guarantee (commit-or-rollback) . . . 1017
71.3.3 3) Basic guarantee . . . 1017
71.3.4 4) No guarantee . . . 1018
71.4 How `noexcept` affects guarantees . . . 1018
71.4.1 E.12: Use `noexcept` when throwing is impossible or unacceptable . . . 1018
71.4.2 E.16: Operations that must never fail . . . 1018
71.5 Design patterns to achieve strong or basic guarantees . . . 1018
71.5.1 Pattern 1: Copy-and-swap (strong guarantee) . . . 1018
71.5.2 Pattern 2: Build-then-commit (strong guarantee) . . . 1019
71.5.3 Pattern 3: Maintain invariants and provide at least the basic guarantee 1020
71.6 Exception guarantees in standard library components (practical view) . . . 1021
71.7 Guarantees for constructors and destructors . . . 1021
71.7.1 Constructors . . . 1021
71.7.2 Destructors . . . 1022
71.8 Specifying exception guarantees in API design . . . 1022
71.8.1 Document guarantees explicitly . . . 1022
71.8.2 Use types to support guarantees . . . 1022
71.8.3 Prefer narrow, composable operations . . . 1023
71.9 A worked example: strong guarantee for a mutating operation . . . 1023
71.10 Common pitfalls that break guarantees . . . 1024
71.10.1 Throwing with “direct ownership” (E.13) . . . 1024
71.10.2 Catching too early and continuing in an unknown state . . . 1024
71.10.3 Throwing from destructors . . . 1025
71.10.4 Marking `noexcept` incorrectly . . . 1025
71.11 Checklist: selecting a guarantee . . . 1025
71.12 Summary . . . 1025

72 No Exceptions from Destructors 1027

72.1 Purpose and scope . . . 1027
72.2 The rule . . . 1027
72.3 Why throwing from a destructor is dangerous . . . 1028
72.3.1 1) Termination during stack unwinding . . . 1028
72.3.2 2) Destructors run in many critical contexts . . . 1028
72.3.3 3) Cleanup must be reliable . . . 1028
72.4 What the language guarantees about destructors . . . 1029
72.4.1 Destructors are implicitly `noexcept` . . . 1029
72.5 Practical rule: destructor code must be "failure-free" . . . 1029
72.5.1 Allowed destructor actions . . . 1029
72.5.2 Forbidden destructor actions . . . 1029
72.6 How to design with "cleanup can fail" . . . 1030
72.6.1 Pattern A: Provide an explicit operation that can report failure . . . 1030
72.6.2 Pattern B: Catch and handle exceptions inside the destructor . . . 1031
72.6.3 Pattern C: "Two-phase" completion (explicit finalize) . . . 1032
72.7 Connection to RAII and invariants . . . 1033
72.7.1 E.6: RAII provides cleanup during exceptions . . . 1033
72.7.2 E.5: constructors establish invariants or throw . . . 1033
72.8 Interaction with `noexcept` (E.12) . . . 1033
72.9 Common mistakes and correct alternatives . . . 1034
72.9.1 Mistake 1: throwing on destructor failure . . . 1034
72.9.2 Mistake 2: calling a throwing function from destructor without handling 1034
72.9.3 Mistake 3: hiding important failures . . . 1034
72.10 Checklist . . . 1035
72.11 Summary . . . 1035

73 Alternatives to Exceptions 1037

73.1 Purpose and scope . . . 1037
73.2 First principle: alternatives must still be safe . . . 1038
73.3 Alternative 1: Return status (`bool` / error enums) and out-parameters . . . 1038
73.3.1 When it fits . . . 1038
73.3.2 Example: status + out-parameter . . . 1039
73.3.3 Pros and cons . . . 1039
73.3.4 Guideline advice (E.27) . . . 1039
73.4 Alternative 2: `std::optional` for "absence is not an error" . . . 1040
73.4.1 When it fits . . . 1040
73.4.2 Important limitation . . . 1040
73.5 Alternative 3: Return a "result" type (value-or-error) . . . 1041
73.5.1 When it fits . . . 1041
73.5.2 Minimal result type example . . . 1041
73.5.3 Discipline requirement . . . 1042
73.6 Alternative 4: Fail fast (terminate) when recovery is unsafe or impossible . . . 1043
73.6.1 When it fits (E.26) . . . 1043
73.6.2 Example: fail fast on contract violation . . . 1043
73.7 Alternative 5: Error propagation boundaries (translation layers) . . . 1044
73.8 Alternative 6: Using `errno` and global state (generally avoid) (E.28) . . . 1044
73.8.1 Guideline position . . . 1044
73.8.2 If you must use C APIs that set `errno` . . . 1044
73.9 Resource safety without exceptions (E.25) . . . 1045
73.9.1 RAII still works . . . 1045
73.9.2 Example: RAII + error codes . . . 1046
73.10 How to choose an alternative: decision guide . . . 1046
73.10.1 If failure is normal and expected . . . 1047
73.10.2 If you need value *or* a reason . . . 1047

73.10.3 If recovery is not safe . . . 1047
73.10.4 If you are wrapping C APIs . . . 1047
73.11 How to keep alternative approaches maintainable . . . 1047
73.11.1 Enforce checking . . . 1047
73.11.2 Centralize error representation . . . 1048
73.11.3 Avoid mixing styles casually . . . 1048
73.12 Checklist . . . 1048
73.13 Summary . . . 1049

XII Concurrency and Parallelism (CP) 1050

74 CP.1 – CP.61: Concurrency and Parallelism (CP) 1052
74.1 Scope and intent . . . 1052
74.1.1 What this chapter covers (and what it does not) . . . 1052
74.2 Essential definitions (precision matters) . . . 1053
74.2.1 Concurrency vs. parallelism . . . 1053
74.2.2 Data race (the one rule you cannot negotiate) . . . 1053
74.2.3 Happens-before (why locks and atomics exist) . . . 1054
74.3 Rule map (CP.1–CP.61) . . . 1054
74.3.1 Defined rules in this numeric range . . . 1054
74.3.2 CP numbers not defined by the Core Guidelines (in CP.1–CP.61) . . . 1054
74.4 CP.1 – CP.9: Baseline rules for all C++ code . . . 1055
74.4.1 CP.1: Assume that your code will run as part of a multi-threaded program 1055
74.4.2 CP.2: Avoid data races . . . 1056
74.4.3 CP.3: Minimize explicit sharing of writable data . . . 1057
74.4.4 CP.4: Think in terms of tasks, rather than threads . . . 1058
74.4.5 CP.8: Don't try to use `volatile` for synchronization . . . 1058

74.4.6 CP.9: Whenever feasible use tools to validate your concurrent code . 1059
74.5 CP.20 – CP.50: Shared-state concurrency rules (CP.con) 1059
74.5.1 CP.20: Use RAII, never plain `lock()`/`unlock()` 1059
74.5.2 CP.21: Use `std::lock()` or `std::scoped_lock` to acquire multiple mutexes 1060
74.5.3 CP.22: Never call unknown code while holding a lock (e.g., a callback) 1061
74.5.4 CP.23: Think of a joining thread as a scoped container 1062
74.5.5 CP.24: Think of a thread as a global container 1063
74.5.6 CP.25: Prefer a joining thread over `std::thread` 1063
74.5.7 CP.26: Don't `detach()` a thread 1064
74.5.8 CP.31: Pass small amounts of data between threads by value 1064
74.5.9 CP.32: To share ownership between unrelated threads use `shared_ptr` 1065
74.5.10 CP.40: Minimize context switching 1066
74.5.11 CP.41: Minimize thread creation and destruction 1066
74.5.12 CP.42: Don't `wait` without a condition 1067
74.5.13 CP.43: Minimize time spent in a critical section 1068
74.5.14 CP.44: Remember to name your `lock_guard`s and `unique_lock`s . . 1069
74.5.15 CP.50: Define a mutex together with the data it guards. Use `synchronized_value<T>` where possible 1069
74.6 CP.51 – CP.53: Coroutines (CP.coro) 1071
74.6.1 CP.51: Do not use capturing lambdas that are coroutines 1071
74.6.2 CP.52: Do not hold locks or other synchronization primitives across suspension points 1071
74.6.3 CP.53: Parameters to coroutines should not be passed by reference . . 1072
74.7 CP.60 – CP.61: Message passing and task results (CP.mess) 1072
74.7.1 CP.60: Use a `future` to return a value from a concurrent task 1072
74.7.2 CP.61: Use `async()` to spawn concurrent tasks 1073

74.8 Practical enforcement checklist (what to review in code) . . . 1074
74.9 Common misconceptions corrected . . . 1075
74.9.1 “If it works in tests, it is thread-safe” . . . 1075
74.9.2 “Using `shared_ptr` makes my object thread-safe” . . . 1075
74.9.3 “`volatile` fixes reordering and visibility” . . . 1075

75 Data Races **1076**
75.1 Purpose . . . 1076
75.2 Precise definition (C++ memory model) . . . 1076
75.2.1 Conflicting actions and potential concurrency . . . 1076
75.2.2 What counts as “the same memory location” . . . 1077
75.2.3 Why “undefined behavior” is stronger than “unpredictable” . . . 1077
75.3 Data race vs. race condition . . . 1078
75.3.1 Race condition (general) . . . 1078
75.3.2 Data race (C++ formal meaning) . . . 1078
75.4 The Core Guidelines position . . . 1078
75.4.1 CP.2: Avoid data races . . . 1078
75.5 How data races happen in real code . . . 1078
75.5.1 Shared mutable state without a lock or atomic . . . 1078
75.5.2 Publishing an object without synchronization (unsafe publication) . . . 1079
75.5.3 Lifetime bugs masquerading as data races . . . 1079
75.6 Canonical examples . . . 1079
75.6.1 Example 1: Non-atomic increment . . . 1079
75.6.2 Example 2: “Read-only” that is not actually read-only . . . 1080
75.6.3 Example 3: Volatile does not fix it . . . 1081
75.7 How to prevent data races . . . 1082
75.7.1 Strategy A: Eliminate sharing (preferred) . . . 1082
75.7.2 Strategy B: Protect shared state with mutexes . . . 1082

75.7.3 Strategy C: Use atomics for atomic objects and simple coordination . 1083
75.7.4 Strategy D: Use futures and tasks to avoid shared writable state . . . 1084
75.8 Common misconceptions and their corrections 1085
75.8.1 `const` makes it thread-safe 1085
75.8.2 “It is just one machine instruction” 1085
75.8.3 “Data races only cause wrong values” 1085
75.9 Detecting data races (practice) 1085
75.9.1 Use dynamic analysis tools 1085
75.9.2 Write concurrency-focused tests 1085
75.10 Summary 1086

76 Mutexes **1087**
76.1 Purpose 1087
76.2 Core rules from the ISO C++ Core Guidelines (CP) 1087
76.2.1 The minimum set you must follow 1087
76.3 When to use a mutex 1088
76.3.1 The main use case 1088
76.3.2 Prefer not sharing instead 1088
76.4 Standard mutex types in C++ 1089
76.4.1 `std::mutex`: the default choice 1089
76.4.2 `std::recursive_mutex`: avoid unless you must 1089
76.4.3 `std::timed_mutex` and `std::recursive_timed_mutex` 1089
76.4.4 `std::shared_mutex`: many readers, few writers 1090
76.5 RAII locking (CP.20) 1090
76.5.1 Never manually `lock()` and `unlock()` 1090
76.5.2 `std::lock_guard`: simplest and safest 1090
76.5.3 `std::unique_lock`: for unlock/relock and condition variables 1091
76.5.4 `std::scoped_lock`: one or many mutexes (C++17+) 1092

76.6 Naming lock objects (CP.44) . . . 1092
76.6.1 Avoid accidental immediate unlock . . . 1092
76.7 Critical sections and invariants (CP.43) . . . 1093
76.7.1 Keep the locked region small . . . 1093
76.8 Deadlocks and how to avoid them . . . 1093
76.8.1 What is a deadlock . . . 1094
76.8.2 Lock ordering rule . . . 1094
76.8.3 Use `std::scoped_lock` or `std::lock` (CP.21) . . . 1094
76.8.4 Never call unknown code while holding a lock (CP.22) . . . 1095
76.9 Mutex + condition variable: correct waiting . . . 1096
76.9.1 The rule: always wait with a predicate . . . 1096
76.9.2 Correct producer/consumer pattern . . . 1097
76.9.3 Why the predicate must be protected by the same mutex . . . 1098
76.10 Encapsulate the mutex with the data (CP.50) . . . 1098
76.10.1 The strongest practical rule for mutex correctness . . . 1098
76.10.2 Example: invariant-preserving class . . . 1098
76.11 Try-lock and timeouts (careful use) . . . 1100
76.11.1 `try_lock()` is not a deadlock solution . . . 1100
76.11.2 Timed locks for bounded waiting . . . 1100
76.12 Performance notes (what matters in practice) . . . 1100
76.12.1 Contention is your real cost . . . 1100
76.12.2 Avoid “lock around everything” designs . . . 1101
76.13 Common mistakes and how to prevent them . . . 1101
76.13.1 Mistake: guarding data with multiple mutexes . . . 1101
76.13.2 Mistake: returning references into protected data . . . 1101
76.13.3 Mistake: locking and then calling user code . . . 1102
76.13.4 Mistake: using `volatile` for synchronization . . . 1103

76.14 Summary 1103

77 Atomics **1104**

77.1 Purpose 1104

77.2 Core Guidelines alignment (Part 12, CP) 1105

77.3 What `std::atomic` guarantees 1105

77.3.1 No data race on the atomic object itself 1105

77.3.2 Visibility and ordering (when you ask for it) 1105

77.3.3 What `std::atomic` does *not* guarantee 1105

77.4 When to use atomics vs. mutexes 1106

77.4.1 Use atomics when 1106

77.4.2 Use mutexes when 1106

77.4.3 A practical engineering rule 1106

77.5 Atomic operations: the essential set 1106

77.5.1 Load and store 1106

77.5.2 Read-modify-write (RMW) operations 1107

77.5.3 Compare-and-exchange (CAS) 1108

77.6 The memory order model (what it means, simply) 1108

77.6.1 Why memory ordering exists 1108

77.6.2 The six standard orders 1108

77.6.3 Relaxed: atomicity without synchronization 1109

77.6.4 Release and acquire: publishing and consuming 1109

77.6.5 Sequential consistency: simplest mental model 1111

77.7 Fences (advanced, use sparingly) 1111

77.7.1 What a fence does 1111

77.7.2 Engineering advice 1111

77.8 Atomic waiting and notification (C++20) 1111

77.8.1 Why it exists 1111

77.8.2 `wait/notify_one/notify_all` . . . 1112
77.9 `std::atomic_ref` (C++20) . . . 1113
77.9.1 What it is . . . 1113
77.9.2 The strict rule you must follow . . . 1113
77.9.3 Example: atomic updates to a field that you cannot change to `std::atomic` . . . 1113
77.10 Lock-free does not mean faster . . . 1114
77.10.1 Lock-free and wait-free (terminology) . . . 1114
77.10.2 Performance reality . . . 1114
77.11 Common mistakes (and correct replacements) . . . 1114
77.11.1 Mistake: using `volatile` instead of atomics . . . 1114
77.11.2 Mistake: mixing atomic and non-atomic access to the same object . . . 1114
77.11.3 Mistake: using relaxed when you need visibility . . . 1115
77.11.4 Mistake: building “clever” lock-free structures without a proof . . . 1115
77.12 Recommended patterns (safe and reviewable) . . . 1115
77.12.1 Pattern 1: relaxed counters . . . 1115
77.12.2 Pattern 2: one-way publication . . . 1115
77.12.3 Pattern 3: atomic state machine . . . 1115
77.12.4 Pattern 4: wait/notify for efficient blocking (C++20) . . . 1116
77.13 Summary . . . 1116

78 Deadlocks 1117
78.1 Purpose . . . 1117
78.2 Definition and the classic necessary conditions . . . 1117
78.2.1 Operational definition . . . 1117
78.2.2 The four necessary conditions (Coffman conditions) . . . 1118
78.3 How deadlocks appear in C++ . . . 1118
78.3.1 The most common causes . . . 1118

78.4 Core Guidelines rules that directly prevent deadlocks 1119
78.5 Deadlock pattern 1: inconsistent lock ordering 1119
78.5.1 Example: two mutexes locked in opposite order 1119
78.5.2 Fix A: use `std::scoped_lock` (C++17+) (CP.21) 1120
78.5.3 Fix B: use `std::lock` + `adopt_lock` (C++11/14) (CP.21) 1121
78.5.4 Fix C: impose and document a strict lock hierarchy 1121
78.6 Deadlock pattern 2: calling unknown code while holding a lock (CP.22) . . . 1122
78.6.1 Why this is dangerous 1122
78.6.2 Bad example: callback under lock 1122
78.6.3 Good example: snapshot under lock, call outside 1123
78.7 Deadlock pattern 3: condition variables used incorrectly 1124
78.7.1 The correct mental model 1124
78.7.2 Correct pattern: predicate wait 1124
78.7.3 Typical deadlock: waiting while holding another lock 1126
78.8 Deadlock pattern 4: joining threads while holding locks 1127
78.8.1 The problem 1127
78.8.2 Bad example 1127
78.8.3 Safe rule 1128
78.8.4 Better design with C++20 `std::jthread` and `stop_token` 1128
78.9 Engineering strategies for deadlock prevention 1129
78.9.1 Strategy 1: avoid shared mutable state 1129
78.9.2 Strategy 2: keep mutexes with the data they guard (CP.50) 1130
78.9.3 Strategy 3: lock multiple mutexes safely (CP.21) 1130
78.9.4 Strategy 4: keep critical sections short (CP.43) 1130
78.9.5 Strategy 5: do not call unknown code under lock (CP.22) 1130
78.9.6 Strategy 6: prefer one lock per invariant 1130
78.10 Detecting and diagnosing deadlocks 1131

78.10.1 Symptoms . . . 1131
78.10.2 Practical diagnostics . . . 1131
78.11 Deadlock vs. livelock vs. starvation . . . 1131
78.11.1 Deadlock . . . 1131
78.11.2 Livelock . . . 1131
78.11.3 Starvation . . . 1132
78.12 Summary . . . 1132

79 Concurrent System Design 1133
79.1 Purpose . . . 1133
79.2 Guiding principles from the ISO C++ Core Guidelines (CP) . . . 1133
79.3 Concurrency vs. parallelism (design consequences) . . . 1134
79.3.1 Concurrency . . . 1134
79.3.2 Parallelism . . . 1134
79.4 The three design axes you must decide early . . . 1135
79.4.1 Axis 1: Shared-state vs. message-passing . . . 1135
79.4.2 Axis 2: Unstructured vs. structured concurrency . . . 1135
79.4.3 Axis 3: Blocking vs. non-blocking coordination . . . 1135
79.5 Concurrency safety contract for every component . . . 1136
79.6 Model A: Shared-state design with encapsulated locking . . . 1136
79.6.1 The rule: protect invariants, not lines of code . . . 1136
79.6.2 Example: a thread-safe bounded queue . . . 1137
79.6.3 Avoiding deadlocks in shared-state systems . . . 1139
79.7 Model B: Message-passing design (queues, actors, and values) . . . 1139
79.7.1 Why message passing scales better . . . 1139
79.7.2 Single-owner event loop (actor-style) skeleton . . . 1140
79.8 Structured shutdown and cancellation (C++20) . . . 1142
79.8.1 Why shutdown is a first-class design problem . . . 1142

79.8.2 Use `std::jthread` and `std::stop_token` for cooperative cancellation . . . 1142
79.8.3 Rule: never join while holding locks the worker might need 1143
79.9 Selecting the right synchronization primitive 1143
79.9.1 Mutexes and condition variables 1144
79.9.2 Atomics 1144
79.9.3 Atomic waiting (`wait/notify`) (C++20) 1144
79.9.4 Latches and barriers (C++20) 1145
79.9.5 Semaphores (C++20) 1146
79.10 Task-based parallelism and standard algorithms 1146
79.10.1 Parallel algorithms 1146
79.10.2 Practical guidance 1146
79.11 Thread pools and execution resources 1146
79.11.1 Why you usually want a pool 1146
79.11.2 A minimal pool interface (conceptual) 1147
79.12 Modern direction: `std::execution` and senders/receivers (C++26 working draft) 1149
79.13 Performance and scalability rules for architects 1150
79.13.1 Rule 1: reduce contention first 1150
79.13.2 Rule 2: avoid false sharing 1150
79.13.3 Rule 3: control parallelism, do not assume “more threads is faster” . . 1150
79.13.4 Rule 4: prefer clear correctness over cleverness 1150
79.14 Design review checklist (practical) 1151
79.15 Summary 1151

80 Message Passing **1152**
80.1 Purpose 1152
80.2 Core Guidelines view: CP.mess (Message passing) 1153
80.3 What message passing means in practice 1153

80.3.1 Key idea: ownership stays local 1153
80.3.2 Benefits over shared-state designs 1154
80.3.3 Costs and trade-offs 1154
80.4 Standard C++ building blocks for message passing 1154
80.4.1 `std::future` and `std::promise` (CP.60) 1154
80.4.2 `std::async` (CP.61) 1156
80.4.3 `std::packaged_task` 1157
80.5 Message passing with queues (channels) 1158
80.5.1 The producer/consumer queue pattern 1158
80.5.2 A correct blocking queue (mutex + condition variable) 1158
80.5.3 Design rule: messages should be values 1160
80.6 Actor-style message passing (single-owner state) 1160
80.6.1 Why actors simplify correctness 1160
80.6.2 Minimal actor skeleton with `std::jthread` (C++20) 1161
80.7 Atomic wait/notify as a message-passing aid (C++20) 1163
80.7.1 What it is 1163
80.7.2 A simple “event” message 1164
80.8 Error propagation and failure semantics 1164
80.8.1 Prefer “value or exception” channels 1164
80.8.2 Queues need an explicit error strategy 1165
80.9 Shutdown and cancellation in message-passing systems 1165
80.9.1 Why shutdown must be designed 1165
80.9.2 Cooperative stop with `std::stop_token` (C++20) 1165
80.9.3 Queue close + stop request: a robust combination 1166
80.10 Modern direction: composable asynchronous message passing (`std::execution`) 1166
80.11 Design checklist for message-passing components 1167

80.12 Summary 1167

XIII Performance (Per) 1169

81 Per.1 – Per.14 1171

81.1 Scope and intent of the Performance rules 1171
81.2 Per.1: Don't optimize without reason 1171
81.2.1 Rule 1172
81.2.2 Reason 1172
81.2.3 Practical guidance 1172
81.3 Per.2: Don't optimize prematurely 1172
81.3.1 Rule 1172
81.3.2 Reason 1172
81.3.3 Practical guidance 1173
81.4 Per.3: Don't optimize something that's not performance critical 1173
81.4.1 Rule 1173
81.4.2 Reason 1173
81.4.3 Practical guidance 1173
81.5 Per.4: Don't assume that complicated code is necessarily faster than simple code 1174
81.5.1 Rule 1174
81.5.2 Reason 1174
81.5.3 Example 1174
81.5.4 Guidance 1174
81.6 Per.5: Don't assume that low-level code is necessarily faster than high-level code 1175
81.6.1 Rule 1175
81.6.2 Reason 1175
81.6.3 Practical examples 1175

81.7 Per.6: Don't make claims about performance without measurements 1175
81.7.1 Rule 1175
81.7.2 Reason 1176
81.7.3 Measurement essentials 1176
81.7.4 A minimal timing harness 1176
81.8 Per.7: Design to enable optimization 1177
81.8.1 Rule 1177
81.8.2 Reason 1177
81.8.3 Guidance 1177
81.8.4 Example: avoid type-erasing C interfaces 1178
81.9 Per.10: Rely on the static type system 1178
81.9.1 Rule 1178
81.9.2 Reason 1178
81.9.3 Guidance 1179
81.10 Per.11: Move computation from run time to compile time 1179
81.10.1 Rule 1179
81.10.2 Reason 1179
81.10.3 Example: prefer compile-time initialization 1179
81.10.4 Important trade-offs 1180
81.11 Per.12: Eliminate redundant aliases 1180
81.11.1 Rule 1180
81.11.2 Why it matters 1180
81.11.3 Guidance (safe, portable) 1180
81.11.4 Example: clarify which range is modified 1181
81.11.5 Note 1181
81.12 Per.13: Eliminate redundant indirections 1181
81.12.1 Rule 1181

81.12.2 Why it matters . . . 1182
81.12.3 Guidance . . . 1182
81.12.4 Example: replace pointer chasing with contiguous storage when possible . . . 1182
81.13 Per.14: Minimize the number of allocations and deallocations . . . 1183
81.13.1 Rule . . . 1183
81.13.2 Why it matters . . . 1183
81.13.3 Guidance . . . 1183
81.13.4 Example: avoid repeated reallocation . . . 1184
81.13.5 Example: `std::pmr` for controlled allocation . . . 1184
81.14 Putting Per.1–Per.14 into a single workflow . . . 1185

82 When to Optimize 1186
82.1 Purpose . . . 1186
82.2 Core message from the Performance rules (Per) . . . 1186
82.3 When optimization is *actually* needed . . . 1187
82.3.1 Trigger 1: a defined performance requirement exists . . . 1187
82.3.2 Trigger 2: measurements show a real problem . . . 1187
82.3.3 Trigger 3: a scalability bottleneck is proven . . . 1188
82.4 When optimization is *not* needed . . . 1188
82.4.1 No measurable bottleneck . . . 1188
82.4.2 The system is I/O bound or externally limited . . . 1188
82.4.3 You cannot test or reproduce performance . . . 1188
82.5 A practical decision workflow . . . 1189
82.5.1 Step 1: State the performance goal . . . 1189
82.5.2 Step 2: Establish a baseline . . . 1189
82.5.3 Step 3: Find the bottleneck (Per.3) . . . 1189
82.5.4 Step 4: Optimize with minimal complexity (Per.4, Per.5) . . . 1190
82.5.5 Step 5: Re-measure and keep only wins (Per.6) . . . 1190

82.6 Measurement fundamentals you must respect (Per.6) . . . 1191
82.6.1 Use a monotonic clock for timing . . . 1191
82.6.2 A minimal, reproducible timing harness . . . 1191
82.6.3 Avoid common benchmarking mistakes . . . 1192
82.7 What to optimize first (highest return on effort) . . . 1192
82.7.1 1) Algorithmic complexity . . . 1192
82.7.2 2) Data structures and memory layout . . . 1192
82.7.3 3) Allocation rate and object lifetime . . . 1193
82.7.4 4) Concurrency contention . . . 1193
82.8 How to decide between readability and performance . . . 1193
82.8.1 Default rule . . . 1193
82.8.2 When a faster version is acceptable . . . 1194
82.8.3 Keep optimized code maintainable . . . 1194
82.9 Checklist: “Should we optimize now?” . . . 1194
82.10 Summary . . . 1195

83 Measure Before Optimizing 1196
83.1 Purpose . . . 1196
83.2 Why Measurement Matters . . . 1196
83.2.1 Avoid wasted effort . . . 1196
83.2.2 Prevent interfering effects . . . 1197
83.3 The Guiding Rule: Per.6 . . . 1197
83.3.1 Statement of the rule . . . 1197
83.3.2 What this means in practice . . . 1197
83.4 Types of Measurement . . . 1198
83.4.1 Profiling . . . 1198
83.4.2 Benchmarking . . . 1198
83.5 Principles of Good Measurement . . . 1198

83.5.1 Define the metric clearly . . . 1198
83.5.2 Use a stable clock source . . . 1199
83.5.3 Isolate the code under test . . . 1199
83.5.4 Warm up caches and environment . . . 1199
83.5.5 Run multiple iterations . . . 1200
83.5.6 Control for measurement noise . . . 1200
83.6 A Simple Measurement Framework in C++ . . . 1200
83.6.1 Analyzing the results . . . 1201
83.7 Microbenchmarks vs. System Benchmarks . . . 1201
83.7.1 Microbenchmarks . . . 1202
83.7.2 System Benchmarks . . . 1202
83.8 Common Measurement Pitfalls . . . 1202
83.8.1 Measuring debug builds . . . 1202
83.8.2 Measuring initialization rather than steady state . . . 1202
83.8.3 Changing code between runs . . . 1203
83.8.4 Misinterpreting variability . . . 1203
83.9 Profile Before You Optimize . . . 1203
83.9.1 Why profiling matters . . . 1203
83.9.2 Types of profiling useful for C++ . . . 1204
83.10 Using Profilers to Guide Optimization . . . 1204
83.10.1 Interpret results carefully . . . 1204
83.10.2 Iterative approach . . . 1204
83.11 Validating the Impact of Optimization . . . 1205
83.11.1 Re-measure after each change . . . 1205
83.11.2 Check for unintended regressions . . . 1205
83.11.3 Maintain correctness under optimization . . . 1205
83.12 When Optimized Code Should Be Reviewed Differently . . . 1205

83.13 Summary 1206

84 Cost Models **1207**

84.1 Purpose 1207
84.2 Cost model foundations for modern C++ 1208
84.2.1 Two levels of cost 1208
84.2.2 The “trust but verify” rule (Per) 1208
84.3 Algorithmic cost models you must use 1208
84.3.1 Big-O as a design filter 1208
84.3.2 Standard library complexity guarantees are part of the model 1209
84.3.3 Amortized complexity is real cost, not a trick 1209
84.4 Machine-level cost models that dominate real performance 1209
84.4.1 Memory hierarchy and locality 1209
84.4.2 Allocation cost and variability 1210
84.4.3 Branching and predictability 1210
84.4.4 Synchronization and contention 1210
84.5 API-level cost models (what your interfaces promise) 1211
84.5.1 Cost transparency: cheap operations should look cheap 1211
84.5.2 Example: `std::string` to `std::string_view` is a cheap view . . . 1211
84.5.3 Value vs. reference: the cost model depends on size, ownership, and lifetime 1212
84.6 Container and algorithm cost model examples 1212
84.6.1 Example 1: reserve to control reallocation cost 1212
84.6.2 Example 2: beware of reference/iterator invalidation on growth . . . 1213
84.6.3 Example 3: choosing between `unordered_map` and sorting 1213
84.7 Building a cost model checklist for performance work 1214
84.8 How cost models interact with measurement (the correct workflow) 1214
84.9 Summary 1215

85 Cache-Friendly Code 1216

85.1 Purpose . . . 1216

85.2 What caches are and why they dominate performance . . . 1216

85.2.1 The key idea: the CPU is much faster than memory . . . 1216

85.2.2 Two kinds of locality . . . 1217

85.2.3 The Core Guidelines performance message . . . 1217

85.3 The main rules of cache-friendly design . . . 1217

85.3.1 Rule 1: Prefer contiguous storage for hot data . . . 1217

85.3.2 Rule 2: Minimize pointer chasing on critical paths . . . 1218

85.3.3 Rule 3: Iterate in the order the data is laid out . . . 1219

85.3.4 Rule 4: Use blocking (tiling) for large working sets . . . 1220

85.3.5 Rule 5: Reduce allocations and shrink the working set . . . 1222

85.3.6 Rule 6: Arrange hot data together and cold data apart . . . 1222

85.4 Data layout choices: AoS vs SoA . . . 1223

85.4.1 Array of Structures (AoS) . . . 1223

85.4.2 Structure of Arrays (SoA) . . . 1223

85.5 Cache friendliness in concurrent code: avoid false sharing . . . 1224

85.5.1 What false sharing is . . . 1224

85.5.2 A common pattern to avoid . . . 1224

85.6 Practical do's and don'ts . . . 1225

85.6.1 Do . . . 1225

85.6.2 Don't . . . 1225

85.7 How to validate cache-friendly changes . . . 1226

85.7.1 Measure the right thing . . . 1226

85.7.2 Use evidence-based workflow (Per) . . . 1226

85.8 Summary . . . 1226

XIV The Standard Library (SL) 1228

86 SL.1 – SL.16 1230

86.1 Scope and mapping note . . . 1230
86.2 SL.1 (SL.1): Use libraries wherever possible . . . 1231
86.2.1 Rule . . . 1231
86.2.2 Reason . . . 1231
86.2.3 Guidance . . . 1231
86.2.4 Example . . . 1231
86.2.5 Common mistakes . . . 1232
86.3 SL.2 (SL.2): Prefer the standard library to other libraries . . . 1232
86.3.1 Rule . . . 1232
86.3.2 Reason . . . 1232
86.3.3 Guidance . . . 1232
86.3.4 Exception . . . 1232
86.4 SL.3 (SL.3): Do not add non-standard entities to namespace `std` . . . 1233
86.4.1 Rule . . . 1233
86.4.2 Reason . . . 1233
86.4.3 Guidance . . . 1233
86.4.4 Example . . . 1233
86.5 SL.4 (SL.4): Use the standard library in a type-safe manner . . . 1234
86.5.1 Rule . . . 1234
86.5.2 Reason . . . 1234
86.5.3 Guidance . . . 1234
86.5.4 Example: avoid raw-memory tricks on non-trivial objects . . . 1234
86.6 SL.5 (SL.con.1): Prefer `std::array` or `std::vector` over C arrays . . . 1235
86.6.1 Rule . . . 1235
86.6.2 Reason . . . 1235

86.6.3 Guidance . . . 1235
86.6.4 Example . . . 1235
86.7 SL.6 (SL.con.2): Prefer std::vector by default . . . 1236
86.7.1 Rule . . . 1236
86.7.2 Reason . . . 1236
86.7.3 Guidance . . . 1236
86.7.4 Example: common initialization pitfall . . . 1236
86.8 SL.7 (SL.con.3): Avoid bounds errors . . . 1236
86.8.1 Rule . . . 1237
86.8.2 Reason . . . 1237
86.8.3 Guidance . . . 1237
86.8.4 Example: safe loop . . . 1237
86.9 SL.8 (SL.con.4): Do not use memset/memcpy on non-trivially-copyable objects 1238
86.9.1 Rule . . . 1238
86.9.2 Reason . . . 1238
86.9.3 Guidance . . . 1238
86.9.4 Example . . . 1238
86.10 SL.9 (SL.str.1): Use std::string to own character sequences . . . 1239
86.10.1 Rule . . . 1239
86.10.2 Reason . . . 1239
86.10.3 Example . . . 1239
86.11 SL.10 (SL.str.2): Use std::string_view or span<char> to refer to character sequences . . . 1240
86.11.1 Rule . . . 1240
86.11.2 Reason . . . 1240
86.11.3 Guidance . . . 1240
86.11.4 Example . . . 1240

86.12 SL.11 (SL.str.3): Use `zstring`/`czstring` for C-style zero-terminated strings . 1241
86.12.1 Rule . . . 1241
86.12.2 Reason . . . 1241
86.12.3 Guidance . . . 1241
86.13 SL.12 (SL.str.4): Use `char*` to refer to a single character . . . 1241
86.13.1 Rule . . . 1242
86.13.2 Reason . . . 1242
86.13.3 Example: dangerous misuse . . . 1242
86.14 SL.13 (SL.str.5): Use `std::byte` for non-character byte values . . . 1242
86.14.1 Rule . . . 1242
86.14.2 Reason . . . 1243
86.14.3 Example . . . 1243
86.15 SL.14 (SL.io.1): Use character-level input only when you have to . . . 1243
86.15.1 Rule . . . 1243
86.15.2 Reason . . . 1243
86.15.3 Example . . . 1244
86.16 SL.15 (SL.io.2): When reading, always consider ill-formed input . . . 1244
86.16.1 Rule . . . 1244
86.16.2 Reason . . . 1244
86.16.3 Guidance . . . 1244
86.16.4 Example . . . 1245
86.17 SL.16 (SL.io.3): Prefer iostreams for I/O . . . 1245
86.17.1 Rule . . . 1245
86.17.2 Reason . . . 1245
86.17.3 Guidance . . . 1245
86.17.4 Example . . . 1246
86.18 Consolidated checklist . . . 1246

87 Proper Use of the STL 1248

87.1 Purpose . . . 1248

87.2 The STL mental model . . . 1249

87.2.1 Three parts that must match . . . 1249

87.2.2 Rule of thumb . . . 1249

87.3 Core Guidelines alignment (SL) . . . 1249

87.3.1 High-level rules you should enforce . . . 1249

87.4 Prefer algorithms over raw loops . . . 1250

87.4.1 Why . . . 1250

87.4.2 Example: `find` + `erase` with correct semantics . . . 1250

87.4.3 Example: transform instead of manual mapping . . . 1250

87.5 Use the right container (the default is `std::vector`) . . . 1251

87.5.1 `std::vector` as the default . . . 1251

87.5.2 When `std::array` is better . . . 1251

87.5.3 When `std::deque` is useful . . . 1251

87.5.4 When node-based containers are appropriate . . . 1252

87.6 Know iterator invalidation rules . . . 1252

87.6.1 Key facts (high-impact) . . . 1252

87.6.2 Example: avoid keeping references across vector growth . . . 1252

87.6.3 Practical rule . . . 1253

87.7 Avoid bounds errors by design . . . 1253

87.7.1 Prefer range-based loops . . . 1253

87.7.2 Use `.at()` at trust boundaries . . . 1254

87.8 Use views to avoid unnecessary allocations and copies . . . 1254

87.8.1 `std::string_view` for read-only string parameters . . . 1254

87.8.2 `std::span` for non-owning array views . . . 1254

87.9 Prefer value semantics and RAII . . . 1255

87.9.1 Use standard owning types 1255
87.9.2 Avoid manual new/delete 1255
87.10 Use the correct algorithm for the job 1256
87.10.1 Do not misuse std::sort vs std::stable_sort 1256
87.10.2 Prefer partitioning and selection algorithms when you do not need full sorting 1256
87.11 Comparators and strict weak ordering 1256
87.11.1 Safe comparator example 1257
87.12 Avoid raw memory functions on non-trivial types 1257
87.12.1 Rule 1257
87.12.2 Correct alternatives 1257
87.13 Understand complexity and avoid accidental quadratic behavior 1258
87.13.1 Common trap: repeated growth without reserving 1258
87.13.2 Better: reserve capacity 1258
87.14 Use ranges when they improve clarity (C++20 and later) 1259
87.14.1 Example: filter then transform 1259
87.14.2 Important rule 1260
87.15 Use polymorphic allocators (std::pmr) when allocation strategy matters . . . 1260
87.16 Common STL anti-patterns (and fixes) 1260
87.16.1 Anti-pattern 1: using std::vector<bool> as a normal container . . 1260
87.16.2 Anti-pattern 2: storing raw pointers as ownership 1261
87.16.3 Anti-pattern 3: ignoring invalidation and lifetime 1261
87.16.4 Anti-pattern 4: mixing unrelated responsibilities 1261
87.17 Practical checklist: proper STL usage 1261
87.18 Summary 1262

88 Containers **1263**
88.1 Purpose 1263

88.2 Container taxonomy . . . 1263
88.2.1 Sequence containers . . . 1264
88.2.2 Associative containers (ordered) . . . 1264
88.2.3 Unordered associative containers (hash-based) . . . 1264
88.2.4 Container adaptors . . . 1264
88.3 The Core Guidelines rules for containers (SL.con) . . . 1265
88.3.1 SL.con.1: Prefer using STL `array` or `vector` instead of a C array . . . 1265
88.3.2 SL.con.2: Prefer using `vector` by default unless you have a reason to use a different container . . . 1265
88.3.3 SL.con.3: Avoid bounds errors . . . 1266
88.3.4 SL.con.4: Do not use `memset`/`memcpy` for arguments that are not trivially-copyable . . . 1266
88.4 How to choose the right container . . . 1267
88.4.1 Start with the access pattern . . . 1267
88.4.2 Default recommendations . . . 1267
88.5 Contiguous storage and locality . . . 1268
88.5.1 `std::vector` and contiguous elements . . . 1268
88.5.2 Avoid `std::vector<bool>` when you need normal element semantics 1268
88.6 Iterator and reference invalidation . . . 1268
88.6.1 A safe rule for `std::vector` . . . 1269
88.6.2 Erasing while iterating . . . 1269
88.7 Performance guidance for containers . . . 1270
88.7.1 Minimize allocations . . . 1270
88.7.2 Prefer locality over pointer-chasing on hot paths . . . 1271
88.8 Ownership and lifetime with containers . . . 1271
88.8.1 Store values by default . . . 1271
88.8.2 If you need dynamic lifetime, prefer owning smart pointers . . . 1271

88.9 Non-owning views: `std::span` . . . 1272
88.9.1 Why `span` helps . . . 1272
88.9.2 Example . . . 1272
88.10 Common container mistakes and fixes . . . 1273
88.10.1 Mistake: choosing `list` for "fast insert/erase" without measuring . . . 1273
88.10.2 Mistake: using `unordered_map` without reserving in growth-heavy code . . . 1273
88.10.3 Mistake: mixing representation with interface . . . 1273
88.11 Summary . . . 1273

89 Algorithms 1275
89.1 Purpose . . . 1275
89.2 The algorithm model: "operate on a range" . . . 1276
89.2.1 Classic iterator-pair form . . . 1276
89.2.2 C++20 ranges algorithms (recommended when available) . . . 1276
89.3 Core principles for proper algorithm use . . . 1277
89.3.1 Principle 1: Prefer algorithms to handwritten loops . . . 1277
89.3.2 Principle 2: Make preconditions explicit . . . 1277
89.3.3 Principle 3: Understand iterator invalidation . . . 1278
89.4 Algorithm categories (what to reach for) . . . 1278
89.4.1 Non-modifying sequence algorithms . . . 1278
89.4.2 Modifying sequence algorithms . . . 1278
89.4.3 Sorting and ordering . . . 1279
89.4.4 Binary search on sorted data . . . 1280
89.5 Correctness contracts you must respect . . . 1280
89.5.1 Comparators: strict weak ordering . . . 1280
89.5.2 Projections (C++20 ranges): prefer them to custom comparators . . . 1281
89.5.3 Overlapping ranges: do not assume it is safe . . . 1281
89.6 Erase correctly: the erase-remove idiom . . . 1281

89.6.1 Why `remove` does not erase . . . 1281
89.6.2 The correct pattern for sequence containers . . . 1282
89.7 Numeric algorithms and reductions . . . 1282
89.7.1 `std::accumulate` vs `std::reduce` . . . 1282
89.8 Parallel algorithms (C++17 execution policies): power with sharp edges . . . 1283
89.8.1 What execution policies do . . . 1283
89.8.2 Critical safety rule: avoid data races and disallowed synchronization . 1283
89.8.3 Exception behavior with parallel algorithms . . . 1283
89.8.4 Example: safe parallel transformation into an independent output . . 1284
89.9 Ranges pipelines (C++20): expressive composition . . . 1284
89.9.1 Views are lazy; algorithms are eager . . . 1284
89.9.2 Example: filter + transform pipeline, then materialize . . . 1284
89.9.3 Lifetime rule . . . 1285
89.10 Practical selection guide . . . 1285
89.10.1 Common tasks and the right algorithms . . . 1285
89.11 Common mistakes and how to avoid them . . . 1286
89.11.1 Mistake 1: writing loops that re-implement algorithms . . . 1286
89.11.2 Mistake 2: using `remove` without `erase` . . . 1286
89.11.3 Mistake 3: invalidating iterators and then using them . . . 1286
89.11.4 Mistake 4: using parallel policies with shared mutable state . . . 1286
89.11.5 Mistake 5: assuming floating-point reductions are identical . . . 1286
89.12 Checklist: proper algorithm usage . . . 1286

90 Ranges 1288
90.1 Purpose . . . 1288
90.2 Core ideas and terminology . . . 1288
90.2.1 Range . . . 1289
90.2.2 View . . . 1289

90.2.3 Iterator and sentinel . . . 1289
90.2.4 Constrained algorithms . . . 1289
90.3 Why ranges improve STL usage . . . 1290
90.3.1 Less error-prone . . . 1290
90.3.2 Better composability . . . 1290
90.3.3 Performance-friendly by default . . . 1290
90.4 Basic usage patterns . . . 1290
90.4.1 Ranges algorithms (direct range argument) . . . 1290
90.4.2 Projections (C++20) to reduce comparator boilerplate . . . 1291
90.4.3 Finding elements with projections . . . 1291
90.5 Views and pipelines . . . 1292
90.5.1 Filter and transform . . . 1292
90.5.2 Key rule: views are usually non-owning . . . 1292
90.5.3 When to materialize . . . 1292
90.6 Returning iterators safely: borrowed ranges and `dangling` . . . 1293
90.6.1 The problem . . . 1293
90.6.2 The solution in C++20 . . . 1293
90.6.3 What this means in practice . . . 1293
90.7 Subranges: a first-class “slice” type . . . 1294
90.7.1 `std::ranges::subrange` . . . 1294
90.7.2 Example: remove returns a subrange . . . 1294
90.8 Contracts and correctness rules . . . 1294
90.8.1 Predicates and projections should be pure . . . 1294
90.8.2 Comparators must satisfy strict weak ordering . . . 1295
90.8.3 Iterator category matters . . . 1295
90.9 Interoperability with existing STL types . . . 1295
90.9.1 Containers as ranges . . . 1295

90.9.2 Non-owning views you should combine with ranges 1296
90.10 Performance guidance for ranges 1296
90.10.1 Costs to remember 1296
90.10.2 Rule: measure before micro-optimizing 1296
90.11 Common mistakes and how to avoid them 1296
90.11.1 Mistake 1: dangling views 1296
90.11.2 Mistake 2: assuming a view is multi-pass 1297
90.11.3 Mistake 3: side effects in predicates 1297
90.11.4 Mistake 4: forgetting that algorithms may return `dangling` 1298
90.12 Practical checklist 1298

91 Avoiding Reinvention 1299
91.1 Purpose 1299
91.2 Core Guidelines position 1299
91.2.1 SL.1: Use libraries wherever possible 1299
91.2.2 SL.2: Prefer the standard library to other libraries 1300
91.2.3 What “avoid reinvention” means 1300
91.3 Why reinvention is harmful 1300
91.3.1 Correctness risk 1300
91.3.2 Security risk 1301
91.3.3 Performance risk 1301
91.3.4 Maintenance and portability cost 1301
91.4 A practical “anti-reinvention” checklist 1302
91.5 Common reinventions and the correct STL replacements 1302
91.5.1 Reinventing dynamic arrays 1302
91.5.2 Reinventing fixed-size arrays 1303
91.5.3 Reinventing ownership and lifetime management 1303
91.5.4 Reinventing search, counting, and loops 1304

91.5.5 Reinventing “remove elements” . . . 1305
91.5.6 Reinventing sorting and ordering . . . 1306
91.5.7 Reinventing string handling and string parameters . . . 1306
91.5.8 Reinventing “pointer + length” array parameters . . . 1307
91.5.9 Reinventing byte buffers . . . 1307
91.5.10 Reinventing timing and durations . . . 1307
91.5.11 Reinventing concurrency primitives . . . 1307
91.6 When reinvention *is* justified . . . 1308
91.7 A disciplined replacement workflow . . . 1308
91.8 Summary . . . 1309

XV Source Files and Build Structure (SF) 1310

92 SF.1 – SF.11: Source Files and Build Structure 1312
92.1 Scope and intent . . . 1312
92.2 Background: what these rules protect . . . 1312
92.3 Recommended project layout (practical baseline) . . . 1313
92.4 SF.1: Use a `.cpp` suffix for code files and `.h` for interface files (unless your project already follows another convention) . . . 1313
92.5 SF.2: A `.h` file must not contain object definitions or non-inline function definitions . . . 1314
92.6 SF.3: Use `.h` files for all declarations used in multiple source files . . . 1316
92.7 SF.4: Include `.h` files before other declarations in a file . . . 1317
92.8 SF.5: A `.cpp` file must include the `.h` file(s) that defines its interface . . . 1318
92.9 SF.6: Use `using namespace` directives for transition, for foundation libraries (such as `std`), or within a local scope only . . . 1319
92.10 SF.7: Don’t write `using namespace` at global scope in a header file . . . 1320

92.11 SF.8: Use #include guards for all .h files . . . 1322
92.12 SF.9: Avoid cyclic dependencies among source files . . . 1323
92.13 SF.10: Avoid dependencies on implicitly #included names . . . 1324
92.14 SF.11: Header files should be self-contained . . . 1326
92.15 Build-structure checklist (practical summary) . . . 1327
92.16 Tool support (adoption at scale) . . . 1328

93 File organization 1329
93.1 Purpose . . . 1329
93.2 Core principle: separate interface from implementation . . . 1329
93.2.1 Interface lives in headers . . . 1330
93.2.2 Implementation lives in source files . . . 1330
93.2.3 Why this matters (ODR and link-time failures) . . . 1330
93.3 A scalable directory layout . . . 1330
93.3.1 Public vs private headers . . . 1331
93.4 Header content rules (what belongs in a header) . . . 1331
93.4.1 Avoid definitions that cause ODR problems . . . 1332
93.4.2 Safe header constants: modern patterns . . . 1332
93.5 Make headers self-contained . . . 1332
93.5.1 Rule . . . 1332
93.5.2 Practical checklist . . . 1332
93.5.3 A robust pattern: include what you use . . . 1333
93.6 Include guards and portability . . . 1333
93.6.1 Rule . . . 1334
93.6.2 Guard naming . . . 1334
93.6.3 About #pragma once . . . 1334
93.7 Include order and consistency . . . 1334
93.7.1 Place includes first . . . 1334

93.7.2 In `.cpp`, include the corresponding header first . . . 1334
93.8 Avoid transitive-include dependencies . . . 1335
93.8.1 The problem . . . 1335
93.8.2 Rule of thumb . . . 1335
93.9 Avoid cyclic header dependencies . . . 1336
93.9.1 Why cycles are harmful . . . 1336
93.9.2 How to break cycles safely . . . 1336
93.10 Forward declarations vs includes . . . 1336
93.10.1 When forward declarations are appropriate . . . 1336
93.10.2 Example: forward declaration with PImpl . . . 1337
93.10.3 When forward declarations are *not* enough . . . 1338
93.11 Namespaces and headers . . . 1339
93.11.1 No `using namespace` in headers . . . 1339
93.11.2 Preferred style in headers . . . 1339
93.11.3 Local allowances in `.cpp` . . . 1339
93.12 Configuration headers and build-generated headers . . . 1339
93.12.1 Configuration header policy . . . 1339
93.12.2 Do not leak build-system details into public interfaces . . . 1340
93.13 Precompiled headers and unity builds (build-time techniques) . . . 1340
93.13.1 Precompiled headers (PCH) . . . 1340
93.13.2 Unity builds . . . 1340
93.14 Modules note (C++20 and later) . . . 1340
93.15 Automated enforcement . . . 1341
93.15.1 High-value CI checks . . . 1341
93.15.2 Static analysis . . . 1341

94 Header files 1342
94.1 Purpose of header files . . . 1342

94.2 What belongs in a header . . . 1342
94.2.1 Header files express interfaces . . . 1343
94.2.2 SF.2: Do not put non-inline definitions in headers . . . 1343
94.2.3 Safe shared constants in headers (modern patterns) . . . 1345
94.3 Headers must be self-contained . . . 1345
94.3.1 SF.11: A header should compile on its own . . . 1345
94.3.2 SF.10: Do not depend on transitive includes . . . 1346
94.3.3 Example: self-contained public header . . . 1346
94.3.4 Recommended CI check: compile each header alone . . . 1347
94.4 Include guards and portability . . . 1347
94.4.1 SF.8: Use include guards for all headers . . . 1347
94.4.2 Why include guards are the baseline . . . 1347
94.4.3 Guard naming . . . 1347
94.4.4 About `#pragma once` . . . 1348
94.5 Include placement and include order . . . 1348
94.5.1 SF.4: Put includes first . . . 1348
94.5.2 SF.5: In a `.cpp`, include the matching header . . . 1348
94.6 Namespace rules for headers . . . 1349
94.6.1 SF.7: Never use `using namespace` at global scope in a header . . . 1349
94.6.2 Why this is dangerous . . . 1349
94.6.3 Bad example . . . 1349
94.6.4 Acceptable alternatives . . . 1349
94.6.5 Narrow exception mentioned by the guidelines . . . 1350
94.7 Forward declarations, incomplete types, and dependency control . . . 1350
94.7.1 Why forward declarations matter . . . 1350
94.7.2 When forward declarations are safe . . . 1350
94.7.3 PImpl pattern to keep heavy includes out of headers . . . 1350

94.7.4 When you must include the full definition in the header 1352
94.8 Macro hygiene in headers 1352
94.8.1 Why macros are risky in interfaces 1353
94.8.2 If you must use macros 1353
94.9 Cyclic include dependencies (SF.9) 1353
94.9.1 What to avoid 1353
94.9.2 How to break cycles 1354
94.10 Templates and inline functions in headers 1354
94.10.1 Why templates are normally defined in headers 1354
94.10.2 Inline functions: what “inline” really means here 1354
94.10.3 Guideline for header-only utilities 1354
94.11 Interaction with C++20 modules and header units 1354
94.11.1 Modules reduce many header problems, but do not remove the need for discipline 1355
94.11.2 Important practical note: macros and standard library modules 1355
94.12 Tooling and enforcement 1355
94.12.1 Clang-Tidy 1355
94.12.2 MSVC code analysis 1355
94.12.3 Recommended enforcement policy for teams 1355
94.13 Checklist: what a professional header should guarantee 1356

95 Include guards 1357
95.1 What an include guard is and why it exists 1357
95.2 SF.8: Use `#include` guards for all `.h` files 1357
95.2.1 Rule 1357
95.2.2 Reason 1358
95.3 How include guards work (mechanics) 1358
95.3.1 Canonical pattern 1358

95.3.2 Execution model 1358
95.4 Guard macro naming: make collisions practically impossible 1359
95.4.1 Why uniqueness matters 1359
95.4.2 Recommended naming strategy 1359
95.4.3 Example: public header 1359
95.4.4 Example: internal/detail header 1360
95.5 Guard placement and structure rules 1360
95.5.1 Place the guard at the very top 1360
95.5.2 Do not wrap includes unnecessarily 1360
95.5.3 Avoid multiple guards in a single header 1361
95.6 Include guards and self-contained headers (SF.10 and SF.11) 1361
95.7 Common include-guard pitfalls (and how to avoid them) 1362
95.7.1 Pitfall 1: guard name collision 1362
95.7.2 Pitfall 2: inconsistent include paths 1362
95.7.3 Pitfall 3: macro-dependent headers 1362
95.8 `#pragma once` and include guards 1363
95.8.1 What `#pragma once` is 1363
95.8.2 Core Guidelines position 1363
95.8.3 MSVC practical note 1363
95.8.4 Engineering guidance 1363
95.9 Performance considerations (what matters and what does not) 1364
95.9.1 Guard form 1364
95.9.2 Build-time strategies do not replace guards 1364
95.10 Testing and enforcement in real projects 1364
95.10.1 Header self-containment test (high value) 1365
95.10.2 Static analysis support 1365
95.11 Practical checklist 1365

96 Modules (where applicable) 1366

96.1 Why modules matter to file and build structure 1366

96.2 What the C++ standard calls “modules” 1367

96.3 Named modules: the core building block 1367

96.3.1 Module interface unit 1367

96.3.2 Module implementation unit 1367

96.3.3 Importing a named module 1368

96.3.4 Module partitions (optional) 1368

96.4 Header units: importing headers 1369

96.4.1 What a header unit is 1369

96.4.2 Example 1369

96.4.3 Important practical note 1369

96.5 The global module fragment and mixing with legacy headers 1369

96.5.1 Why it exists 1370

96.5.2 Example 1370

96.6 Standard library modules (where supported) 1370

96.6.1 What is standardized vs what is provided 1370

96.6.2 MSVC named STL modules: `std` and `std.compat` 1371

96.6.3 Example (MSVC documentation model) 1371

96.6.4 Engineering guidance 1371

96.7 Build system reality: modules require dependency scanning 1371

96.7.1 Why modules change the build graph 1372

96.7.2 CMake support 1372

96.7.3 Practical implication 1372

96.8 Toolchain status (high-level, standards-based guidance) 1372

96.8.1 Do not assume uniform support 1373

96.8.2 Authoritative reference points 1373

96.9 File organization patterns for modular projects . . . 1373
96.9.1 Recommended directory layout . . . 1373
96.9.2 Naming guidance . . . 1374
96.10 Rules of thumb: when to use modules vs headers . . . 1374
96.10.1 Prefer named modules when . . . 1374
96.10.2 Prefer headers when . . . 1374
96.10.3 A pragmatic migration strategy . . . 1375
96.11 Common pitfalls and how to avoid them . . . 1375
96.11.1 Pitfall: macros in public interfaces . . . 1375
96.11.2 Pitfall: assuming `import` behaves like `#include` . . . 1375
96.11.3 Pitfall: build graph not module-aware . . . 1376
96.12 How modules relate back to SF rules . . . 1376

XVI Const Correctness (Con) 1377

97 Con.1 – Con.4: Const Correctness (Constants and Immutability) 1379
97.1 Overview: what “const correctness” buys you . . . 1379
97.2 Const rule summary (Con.1–Con.4) . . . 1380
97.3 Con.1: By default, make objects immutable . . . 1380
97.3.1 Rule . . . 1380
97.3.2 Reason . . . 1380
97.3.3 Example . . . 1380
97.3.4 Important exceptions . . . 1381
97.3.5 Enforcement guidance . . . 1381
97.4 Con.2: By default, make member functions `const` . . . 1381
97.4.1 Rule . . . 1381
97.4.2 Reason . . . 1382

97.4.3 Example: a missing `const` breaks legitimate use 1382
97.4.4 Logical constness, caches, and `mutable` 1382
97.4.5 Dealing with legacy APIs that are not const-correct 1383
97.4.6 Enforcement guidance 1384
97.4.7 Tool support note 1384
97.5 Con.3: By default, pass pointers and references to `const`s 1384
97.5.1 Rule 1384
97.5.2 Reason 1384
97.5.3 Example: the signature communicates intent 1385
97.5.4 Critical note: do not cast away const 1385
97.5.5 Enforcement guidance 1385
97.6 Con.4: Use `const` to define objects with values that do not change after construction 1385
97.6.1 Rule 1385
97.6.2 Reason 1386
97.6.3 Example: remove unnecessary uncertainty 1386
97.6.4 Where Con.4 is most valuable 1386
97.6.5 Enforcement guidance 1386
97.7 Practical patterns and pitfalls (for Con.1–Con.4) 1387
97.7.1 Prefer const-correct interfaces over const-correct call sites 1387
97.7.2 Do not sprinkle const on pass-by-value parameters 1387
97.7.3 Const correctness is about semantics, not just syntax 1387
97.7.4 Do not use `const_cast` as a design tool 1387
97.8 Checklist for code reviews 1387

98 `const` objects 1389
98.1 Scope of this chapter 1389
98.2 Definition: what is a `const` object? 1389

98.2.1 The language rule (observable effect) . . . 1389
98.2.2 Shallow immutability . . . 1390
98.3 Con.1 and Con.4: when to declare objects const . . . 1390
98.3.1 Make objects immutable by default . . . 1390
98.3.2 What const promises . . . 1391
98.4 Initialization: when can a const object be assigned? . . . 1391
98.4.1 Initialization is allowed, later modification is not . . . 1391
98.4.2 const data members must be initialized in constructors . . . 1391
98.4.3 Use constexpr when you need compile-time constants . . . 1392
98.4.4 Use constinit to prevent dynamic initialization . . . 1392
98.5 Const and member functions: what a const object can call . . . 1392
98.5.1 A const object can call only const member functions . . . 1392
98.5.2 Logical constness and mutable . . . 1393
98.6 Const and pointers: reading the type correctly . . . 1393
98.6.1 Top-level vs low-level const . . . 1394
98.7 Const at namespace scope: linkage and headers . . . 1394
98.7.1 A common surprise: const affects linkage . . . 1394
98.7.2 Prefer C++17 inline variables for shared header-defined constants . . . 1395
98.7.3 ODR reminder . . . 1395
98.8 Const does not guarantee “compile-time” . . . 1395
98.8.1 Const vs constant expression . . . 1396
98.9 Undefined behavior: “casting away const” and writing anyway . . . 1396
98.9.1 Why it is dangerous . . . 1396
98.9.2 Acceptable use: fixing a non-const legacy API via a wrapper . . . 1397
98.10 Const and concurrency . . . 1397
98.11 Practical guidance: when and how to use const objects . . . 1397
98.11.1 Prefer const for true invariants . . . 1397

98.11.2 Avoid meaningless const on pass-by-value parameters 1397
98.11.3 Be careful with returned locals and move 1398
98.12 Checklist for reviews and refactoring 1398

99 const member functions 1399
99.1 Purpose and definition 1399
99.2 Core Guideline Con.2: default to const 1400
99.2.1 Rule 1400
99.2.2 Reason 1400
99.2.3 Example: missing const breaks normal usage 1400
99.3 How the language models const member functions 1401
99.3.1 The implicit object parameter (this) 1401
99.3.2 Overloading on constness 1401
99.4 Ref-qualifiers with const member functions 1402
99.4.1 Why this matters 1402
99.4.2 Example: safe accessors 1402
99.5 What const allows and what it does not 1403
99.5.1 Const is not deep immutability 1403
99.5.2 Const should match semantics (observable state) 1404
99.6 Logical constness and mutable 1404
99.6.1 Motivation 1404
99.6.2 Example: caching in a const operation 1404
99.6.3 Guidance 1405
99.7 Const member functions and thread safety 1406
99.7.1 Const does not imply thread-safe 1406
99.7.2 Example: cache with synchronization 1406
99.8 Designing APIs with const and non-const overloads 1407
99.8.1 Pattern 1: read-only vs mutable accessors 1407

99.8.2 Pattern 2: avoid duplication with one implementation 1408
99.8.3 Better alternative (when possible) 1408
99.9 Interaction with `constexpr`, `noexcept`, and `[[nodiscard]]` 1408
99.9.1 `const` composes with other function qualifiers 1408
99.10 Modern note: C++23 "deducing `this`" (explicit object parameter) 1409
99.10.1 Example: one definition handles multiple cv/ref cases 1409
99.11 Common mistakes and how to avoid them 1410
99.11.1 Mistake 1: marking a logically mutating function as const 1410
99.11.2 Mistake 2: forgetting const on observers 1410
99.11.3 Mistake 3: returning non-const references from const member functions 1411
99.11.4 Mistake 4: using `const_cast` as a design tool 1411
99.12 Review checklist 1411

100 Logical vs. bitwise const 1413
100.1 Why this distinction exists 1413
100.2 Definitions 1413
100.2.1 Bitwise (physical) constness 1414
100.2.2 Logical constness 1414
100.3 What the language enforces vs. what you promise 1414
100.3.1 What the compiler enforces inside a `const` member function 1414
100.3.2 What `const` should mean in design 1415
100.4 Examples: bitwise const vs. logical const 1415
100.4.1 Example 1: bitwise-const observer 1415
100.4.2 Example 2: logically const cache 1415
100.5 The role of `mutable` 1417
100.5.1 What `mutable` means 1417
100.5.2 What `mutable` is for (logical state separation) 1417
100.6 Logical constness and concurrency 1417

100.6.1 Critical rule: logically-const writes must be safe under concurrency . 1417
100.6.2 Example: logically const with synchronization 1418
100.7 When logical constness becomes a bug 1419
100.7.1 Bug pattern 1: changing semantics via a `mutable` member 1419
100.7.2 Bug pattern 2: transitive writes through pointers 1419
100.8 Practical guidance for robust const-correct design 1420
100.8.1 Prefer bitwise constness for simple observers 1420
100.8.2 Use logical constness deliberately (and document it) 1420
100.8.3 Checklist for code reviews 1421

XVII Type Safety (Type) 1422

101 Type.1 – Type.7: Type Safety 1424
101.1 Overview: what "type safety" means in the Core Guidelines 1424
101.1.1 Rule summary (Type.1–Type.7) 1424
101.2 Type.1: Avoid casts 1425
101.2.1 Rule 1425
101.2.2 Rationale 1425
101.2.3 Prefer construction (especially `T{e}`) for value conversion 1425
101.2.4 When narrowing is intended: use GSL narrowing helpers 1426
101.2.5 Type.1 strict sub-rules (what the profile bans) 1426
101.2.6 Safer alternatives to common cast motives 1427
101.3 Type.2: Don't use `static_cast` to downcast; use `dynamic_cast` 1427
101.3.1 Rule 1427
101.3.2 Rationale 1427
101.3.3 Example: unsafe `static_cast` vs checked `dynamic_cast` 1428
101.3.4 Design note: prefer avoiding downcasts entirely 1428

101.4 Type.3: Don't use `const_cast` to cast away `const` . . . 1428
101.4.1 Rule . . . 1428
101.4.2 Rationale . . . 1429
101.4.3 Example: undefined behavior risk . . . 1429
101.4.4 Practical guidance . . . 1429
101.5 Type.4: Don't use C-style or functional casts . . . 1429
101.5.1 Rule . . . 1429
101.5.2 Rationale . . . 1430
101.5.3 Example . . . 1430
101.6 Type.5: Don't use a variable before it has been initialized . . . 1430
101.6.1 Rule . . . 1430
101.6.2 Rationale . . . 1431
101.6.3 Example . . . 1431
101.6.4 Recommended initialization style . . . 1431
101.6.5 Tooling . . . 1431
101.7 Type.6: Always initialize a data member . . . 1432
101.7.1 Rule . . . 1432
101.7.2 Rationale . . . 1432
101.7.3 Preferred pattern: default member initializers . . . 1432
101.7.4 When constructor initialization is better . . . 1433
101.8 Type.7: Avoid naked `union`; use `std::variant` instead . . . 1433
101.8.1 Rule . . . 1433
101.8.2 Rationale . . . 1433
101.8.3 Example: from union to variant . . . 1434
101.8.4 When a union can still be appropriate . . . 1434
101.9 Putting it together: a practical Type Safety checklist . . . 1435

102 Type Safety **1436**

102.1 What “type safety” means in modern C++ 1436
102.2 Why type safety matters (engineering, not style) 1437
102.3 The Core Guidelines view: main sources of type-unsafe code 1437
102.4 Casts: the primary escape hatch (Type.1–Type.4) 1437
102.4.1 Principle: avoid casts when a clearer alternative exists 1437
102.4.2 Numeric conversions and narrowing 1438
102.4.3 Downcasting in class hierarchies (Type.2) 1439
102.4.4 Casting away const (Type.3) 1440
102.4.5 C-style and functional casts (Type.4) 1441
102.5 Initialization: prevent “use before set” (Type.5–Type.6) 1441
102.5.1 Type.5: never read an uninitialized object 1441
102.5.2 Prefer value initialization and invariants 1442
102.5.3 Type.6: always initialize every data member 1442
102.5.4 Tooling support 1443
102.6 Avoid raw unions; use `std::variant` (Type.7) 1443
102.6.1 Why a naked union is unsafe for sum types 1443
102.6.2 Use `std::variant` for type-safe alternatives 1443
102.6.3 Low-level note 1444
102.7 Representation reinterpretation: prefer defined behavior 1444
102.8 Type-safety profiles and modern enforcement 1445
102.9 Tooling: how to make type safety real in a codebase 1445
102.9.1 Static analysis (MSVC) 1445
102.9.2 Static analysis (Clang-Tidy) 1445
102.10 Practical checklist 1446

103 Casts **1447**
103.1 Why casts matter to type safety 1447
103.2 Core Guidelines focus: Type.1–Type.4 1448

103.3 First principle: prefer “no cast” solutions 1448
103.3.1 Prefer correct interfaces 1448
103.3.2 Prefer construction for value conversions 1449
103.3.3 Prefer library facilities for representation work 1449
103.4 A map of cast forms and what they mean 1449
103.4.1 `static_cast` 1449
103.4.2 `dynamic_cast` 1450
103.4.3 `const_cast` 1450
103.4.4 `reinterpret_cast` 1450
103.4.5 C-style and functional casts 1450
103.5 Type.4: Avoid C-style casts 1450
103.5.1 Why they are dangerous 1450
103.5.2 Bad and good 1450
103.5.3 Functional casts vs construction 1451
103.6 Type.1: Avoid casts (practical guidance) 1451
103.6.1 Numeric conversions and narrowing 1451
103.6.2 Pointer conversions 1452
103.7 Type.2: Downcasting safely (`dynamic_cast` vs `static_cast`) 1452
103.7.1 The problem with `static_cast` downcasts 1452
103.7.2 Checked downcast with `dynamic_cast` 1452
103.7.3 Design note 1453
103.8 Type.3: `const_cast` and casting away const 1453
103.8.1 The rule 1453
103.8.2 What is actually safe 1453
103.8.3 What is not safe 1454
103.9 `reinterpret_cast`: the most dangerous named cast 1454
103.9.1 Why it is risky 1454

103.9.2 Prefer defined alternatives for bit reinterpretation 1455
103.9.3 If you truly need `reinterpret_cast` 1455
103.10 Casts and object lifetime 1456
103.10.1 Example: “pretending” bytes are an object 1456
103.11 Tooling and enforcement (practical) 1456
103.12 Review checklist 1457

104 `static_cast` 1458
104.1 Role of `static_cast` in type safety 1458
104.2 Core Guidelines position (Type Safety) 1459
104.3 What `static_cast` can do (language-approved conversions) 1459
104.3.1 Numeric conversions 1459
104.3.2 Enumeration conversions 1459
104.3.3 Pointer and reference conversions within a class hierarchy 1459
104.3.4 Converting `void*` to a typed pointer 1460
104.3.5 Reference casts and value categories 1460
104.4 What `static_cast` cannot do 1460
104.4.1 It cannot cast away `const`/`volatile` 1460
104.4.2 It is not a general-purpose “bit reinterpretation” tool 1460
104.5 Best practices: use `static_cast` only when it makes code safer 1461
104.5.1 Prefer construction and brace initialization for numeric conversions . 1461
104.5.2 Use `static_cast` to document intentional, safe conversions 1461
104.5.3 Upcasting: prefer implicit conversions, but `static_cast` can clarify . 1462
104.6 The biggest pitfall: downcasting with `static_cast` (Type.2) 1462
104.6.1 Why it is dangerous 1462
104.6.2 Bad example 1462
104.6.3 Preferred alternative: `dynamic_cast` 1463
104.6.4 Design note: frequent downcasts are a design smell 1463

104.7 Working with void*: what static_cast does and does not guarantee 1464
104.7.1 Converting back from void* 1464
104.8 Tooling: making static_cast usage enforceable 1465
104.8.1 Clang-Tidy 1465
104.8.2 MSVC Core Guidelines Checkers 1465
104.9 Practical checklist for reviews 1465

105 dynamic_cast **1466**
105.1 Role of dynamic_cast in type safety 1466
105.2 When dynamic_cast is allowed 1466
105.2.1 Polymorphic requirement 1467
105.2.2 Allowed target types 1467
105.3 How failure is reported (pointers vs references) 1467
105.3.1 Pointer form: returns null 1467
105.3.2 Reference form: throws std::bad_cast 1468
105.3.3 Guideline: prefer pointer dynamic_cast in general code 1468
105.4 What dynamic_cast can do 1469
105.4.1 Safe downcast (Base → Derived) 1469
105.4.2 Cross-cast and side-cast (between siblings via a shared base) 1469
105.4.3 Cast to void*: pointer to the most-derived object 1470
105.5 What dynamic_cast does not do 1470
105.5.1 It does not replace good design 1470
105.5.2 It cannot be used for non-polymorphic base types 1471
105.5.3 It is not for unrelated types or representation reinterpretation 1471
105.6 RTTI and build configuration considerations 1471
105.6.1 RTTI must be enabled for the runtime check 1471
105.6.2 Interoperability note (exceptions on some implementations) 1471
105.7 Performance and usage guidance 1472

105.7.1 Cost model (high-level) . . . 1472
105.7.2 Preferred patterns . . . 1472
105.8 Common mistakes and how to avoid them . . . 1472
105.8.1 Mistake 1: using `static_cast` for downcasts . . . 1472
105.8.2 Mistake 2: assuming failure cannot happen . . . 1473
105.8.3 Mistake 3: using `dynamic_cast` as a design substitute . . . 1473
105.9 Review checklist . . . 1473

106 Avoiding `reinterpret_cast` 1474
106.1 Why `reinterpret_cast` is a type-safety hazard . . . 1474
106.2 Core Guidelines perspective (Type Safety) . . . 1474
106.2.1 Type.1 and the type-safety profile . . . 1475
106.2.2 Reasoning . . . 1475
106.3 What `reinterpret_cast` actually does (and does not do) . . . 1475
106.3.1 It changes the *view*, not the object . . . 1475
106.3.2 Pointer round-trip is not permission to dereference . . . 1475
106.4 The three big failure modes . . . 1475
106.4.1 Failure mode 1: violating type-based aliasing rules . . . 1476
106.4.2 Failure mode 2: alignment violations . . . 1477
106.4.3 Failure mode 3: object lifetime violations . . . 1478
106.5 Common misuse patterns and safe replacements . . . 1479
106.5.1 Pattern: C-style “type punning” for performance . . . 1479
106.5.2 Pattern: “view a struct header at the start of a byte buffer” . . . 1479
106.5.3 Pattern: downcasting in polymorphic hierarchies . . . 1480
106.6 What is allowed: byte-wise access and object representations . . . 1480
106.6.1 Accessing raw bytes of an object . . . 1481
106.6.2 Recommended practice . . . 1481
106.7 When `reinterpret_cast` might be justified (rare) . . . 1481

106.8 Tooling: enforce “no `reinterpret_cast`” 1482
106.9 Review checklist 1482

XVIII Object Lifetime Safety (Lifetime) 1483

107 Lifetime.1 – Lifetime.6 1485
107.1 Overview: what the Lifetime safety profile is trying to guarantee 1485
107.1.1 Important note about numbering 1485
107.2 Lifetime.1: Don’t dereference a possibly invalid pointer (detect or avoid) . . . 1486
107.2.1 Rule 1486
107.2.2 Typical sources of invalid pointers 1486
107.2.3 Bad example: pointer to a local that goes out of scope 1486
107.2.4 Good patterns 1487
107.3 Lifetime.2: Don’t create dangling pointers or references 1487
107.3.1 Rule 1487
107.3.2 Bad example: returning a reference to a local 1487
107.3.3 Bad example: binding a view to a temporary 1488
107.3.4 Good alternatives 1488
107.4 Lifetime.3: Don’t pass a potentially dangling pointer/reference to another function 1489
107.4.1 Rule 1489
107.4.2 Bad example: passing a pointer to a local 1489
107.4.3 Preferred interface patterns 1489
107.5 Lifetime.4: Don’t return a potentially dangling pointer/reference/view 1490
107.5.1 Rule 1490
107.5.2 Bad example: returning `std::string_view` to a local string 1490
107.5.3 Good alternatives 1490

107.6 Lifetime.5: Avoid invalidation by mutation (iterators, pointers into containers, views) . . . 1491
107.6.1 Rule . . . 1491
107.6.2 Bad example: iterator invalidation . . . 1491
107.6.3 Safer patterns . . . 1492
107.7 Lifetime.6: Make ownership and lifetime relationships explicit . . . 1492
107.7.1 Rule . . . 1492
107.7.2 Practical guidance . . . 1492
107.7.3 Example: explicit ownership via `unique_ptr` . . . 1493
107.8 Enforcement model (what tools typically flag) . . . 1493
107.9 Review checklist . . . 1494

108 Dangling References 1495
108.1 Definition and core risk . . . 1495
108.2 Where dangling references come from . . . 1495
108.2.1 Returning a reference to a local object . . . 1495
108.2.2 Binding references to temporaries . . . 1496
108.2.3 Lifetime of temporaries and reference lifetime-extension . . . 1496
108.2.4 Dangling references via `std::string_view` and other non-owning views . . . 1498
108.2.5 Dangling references after container mutation (invalidation) . . . 1499
108.3 Consequences: why dangling references are severe . . . 1500
108.4 How to prevent dangling references (engineering rules) . . . 1500
108.4.1 Prefer returning values . . . 1500
108.4.2 Make ownership explicit . . . 1500
108.4.3 Treat references as non-owning and short-lived . . . 1501
108.4.4 Be strict with non-owning views . . . 1501
108.4.5 Use tooling based on the Lifetime profile . . . 1501

108.5 Practical review checklist 1502

109 Lifetime Extension 1503

109.1 What “lifetime extension” means 1503
109.2 Why lifetime extension is important for lifetime safety 1503
109.3 The default rule: temporaries die at end of the full-expression 1504
109.4 The main lifetime extension rule: binding to a reference 1504
109.4.1 Const lvalue reference extends the temporary 1504
109.4.2 Rvalue reference also extends the temporary (since C++11) 1505
109.4.3 Key constraint: the binding must be direct and in the right form . . . 1505
109.5 The most important traps and exceptions 1505
109.5.1 Trap 1: binding through a reference parameter does *not* extend beyond the call 1505
109.5.2 Trap 2: lifetime generally cannot be “extended again” by returning a reference 1506
109.5.3 Trap 3: `std::string_view` and other non-owning views do not extend lifetimes 1506
109.5.4 Trap 4: reference members and `new`-initializers (not “as long as the object”) 1507
109.5.5 Trap 5: aggregate initialization subtlety (braces vs parentheses, since C++20) 1508
109.6 When lifetime extension *does* happen reliably 1508
109.6.1 Local reference bound directly to a temporary 1509
109.6.2 Rvalue reference bound directly to a temporary 1509
109.7 Engineering guidelines: using lifetime extension safely 1509
109.7.1 Prefer owning results over reference tricks 1509
109.7.2 Avoid returning references to objects that might be temporaries . . . 1510
109.7.3 Treat non-owning views as “borrowed” and short-lived 1510

109.8 Review checklist (quick) 1510

110 Temporaries **1511**

110.1 What a temporary is 1511
110.2 Why temporaries matter for lifetime safety 1511
110.3 When temporaries are created 1512
110.3.1 Temporary materialization (since C++17 wording model) 1512
110.3.2 Temporary results of conversions 1512
110.3.3 Braced initialization and `std::initializer_list` 1513
110.4 Default destruction rule: end of the full-expression 1513
110.4.1 Example: temporary dies at end of statement 1513
110.5 Lifetime extension: when a temporary can live longer 1513
110.5.1 Direct binding extends lifetime 1514
110.5.2 Rvalue references 1514
110.6 The most important non-extension cases (common traps) 1515
110.6.1 Trap 1: binding through a function parameter does not extend past the full-expression 1515
110.6.2 Trap 2: you generally cannot "extend again" by returning a reference . 1515
110.6.3 Trap 3: non-owning views do not extend lifetime 1516
110.6.4 Trap 4: `new`-initializers and reference members 1517
110.6.5 Trap 5: temporaries in `std::initializer_list` 1517
110.7 Temporaries, return values, and guaranteed copy elision 1517
110.8 Temporaries and sequencing: destruction order notes 1518
110.9 Guidelines for writing lifetime-safe code with temporaries 1518
110.9.1 Prefer owning results to borrowed views 1518
110.9.2 Avoid returning references/views unless the referent lifetime is guaranteed 1519
110.9.3 Use lifetime extension deliberately, not accidentally 1519

110.9.4 Treat view types as “borrowed” 1519
110.10 Review checklist 1519

111 Ownership Clarity 1520
111.1 Purpose of this chapter 1520
111.2 Definitions: ownership, borrowing, and lifetime 1520
111.2.1 Ownership 1521
111.2.2 Borrowing (non-owning access) 1521
111.2.3 Lifetime safety goal 1521
111.3 Core Guidelines principle: encode ownership in types 1522
111.3.1 Rule of thumb 1522
111.4 Owning types: the standard options 1522
111.4.1 Prefer value ownership when practical 1522
111.4.2 `std::unique_ptr`: single owner 1523
111.4.3 `std::shared_ptr`: shared ownership 1523
111.4.4 Avoid owning raw pointers 1524
111.5 Non-owning types: make borrowing explicit 1524
111.5.1 References for mandatory non-null parameters 1524
111.5.2 Raw pointers as observers for optional objects 1524
111.5.3 `std::span` for non-owning ranges 1525
111.5.4 `std::string_view` for string-like input 1525
111.6 Ownership clarity in interfaces 1526
111.6.1 Rule: express ownership transfer in the type 1526
111.6.2 Rule: do not store borrowed pointers/views without a lifetime mechanism 1527
111.7 Annotating legacy code: ownership markers and non-null contracts 1527
111.8 Common ownership anti-patterns and safe replacements 1528
111.8.1 Anti-pattern: function returns a borrowed pointer/reference to a local 1528

111.8.2 Anti-pattern: storing raw pointers without a lifetime policy 1528
111.8.3 Anti-pattern: returning or storing `string_view` to a temporary 1529
111.9 Design patterns that improve ownership clarity 1529
111.9.1 Factory functions 1529
111.9.2 Pimpl (ownership of implementation) 1529
111.9.3 Observer pointers vs. owning pointers (naming and conventions) . . . 1530
111.10 Ownership clarity checklist for reviews 1530

XIX Bounds Safety (Bounds) 1532

112 Bounds Safety (Bounds.1 – Bounds.4) 1534
112.1 Scope and intent of the Bounds profile 1534
112.1.1 Why this matters 1535
112.2 Core idea: represent ranges explicitly 1535
112.2.1 Key vocabulary and types 1535
112.3 Rule Bounds.1: Don't use pointer arithmetic. Use `span` instead 1536
112.3.1 Rule statement 1536
112.3.2 Rationale 1536
112.3.3 Bad examples 1537
112.3.4 Preferred approach: take a `span` (range-as-a-value) 1537
112.3.5 Practical guidance 1538
112.3.6 Enforcement and tool support 1538
112.4 Rule Bounds.2: Only index into arrays using constant expressions 1538
112.4.1 Rule statement 1538
112.4.2 Rationale 1538
112.4.3 Bad examples 1539
112.4.4 Better options 1539

112.4.5 Enforcement and tool support 1540
112.5 Rule Bounds.3: No array-to-pointer decay 1540
112.5.1 Rule statement 1540
112.5.2 Rationale 1540
112.5.3 Bad examples 1541
112.5.4 Preferred approach: preserve bounds in the type 1541
112.5.5 Enforcement and tool support 1542
112.6 Rule Bounds.4: Don't use standard-library functions and types that are not bounds-checked 1542
112.6.1 Rule statement 1542
112.6.2 Rationale 1542
112.6.3 Common examples in practice 1543
112.6.4 Performance reality: checking can be optimized 1544
112.6.5 Enforcement and tool support 1544
112.7 Profile-wide enforcement, suppression, and engineering workflow 1544
112.7.1 Mechanical enforcement ideas (as described by the Guidelines) . . . 1544
112.7.2 Suppression: use sparingly and document why 1545
112.8 Summary: applying Bounds.1–Bounds.4 in modern C++ 1545

113 Out-of-bounds access 1547
113.1 What is out-of-bounds access? 1547
113.1.1 Definition 1547
113.1.2 A crucial point: C++ does not automatically protect you 1548
113.2 How out-of-bounds bugs are created 1548
113.2.1 Unchecked indexing 1548
113.2.2 Pointer arithmetic used as implicit range traversal 1549
113.2.3 Array-to-pointer decay loses the size 1549
113.2.4 Signed/unsigned and integer narrowing bugs 1550

113.3 Why it is dangerous even when it “seems to work” 1550
113.3.1 Undefined behavior is optimization-visible 1550
113.3.2 Out-of-bounds reads can still break invariants 1551
113.4 Bounds profile strategy against out-of-bounds access 1551
113.4.1 Principle 1: represent ranges as a single object 1551
113.4.2 Principle 2: avoid the unsafe building blocks 1552
113.5 Checked access: what “checked” means in practice 1552
113.5.1 `at()` as a standard checked access 1552
113.5.2 Prefer expressing the loop so no per-iteration check is needed 1553
113.6 Bridging to legacy C and low-level APIs safely 1553
113.6.1 Wrap at the boundary 1554
113.6.2 When pointer arithmetic is unavoidable 1554
113.7 Tooling and enforcement 1555
113.7.1 Static analysis 1555
113.7.2 Runtime sanitizers and hardened builds 1555
113.8 Practical checklist for preventing out-of-bounds access 1555
113.9 Summary 1556

114 `span` **1557**
114.1 Purpose and role in the Bounds profile 1557
114.1.1 What `span` is 1557
114.1.2 Why the Core Guidelines emphasize `span` 1557
114.1.3 When to use `span` 1558
114.2 Basic model and key properties 1558
114.2.1 Type parameters: element type and extent 1558
114.2.2 Contiguity requirement 1558
114.2.3 Non-owning and lifetime 1559
114.3 Construction and conversions 1559

114.3.1 Default construction . . . 1559
114.3.2 From pointer and count . . . 1559
114.3.3 From C arrays and `std::array` . . . 1560
114.3.4 From contiguous containers . . . 1560
114.3.5 Static extent size requirements . . . 1561
114.3.6 Initializer-list construction (C++26) . . . 1561
114.3.7 Deduction guides and C++26 improvement . . . 1562
114.4 Access, iteration, and slicing . . . 1562
114.4.1 Core observers . . . 1562
114.4.2 Element access: `operator[]` . . . 1562
114.4.3 `span::at()` (C++26) . . . 1563
114.4.4 Subspans and view-based slicing . . . 1563
114.5 Byte views and object representation . . . 1564
114.5.1 `as_bytes` and `as_writable_bytes` . . . 1564
114.6 Using `span` to design safe interfaces . . . 1565
114.6.1 Replacing pointer+length parameters . . . 1565
114.6.2 Const-correctness patterns . . . 1565
114.6.3 Returning spans: when it is safe . . . 1566
114.7 Safety pitfalls and how to avoid them . . . 1566
114.7.1 Dangling spans . . . 1566
114.7.2 Reallocation invalidation . . . 1567
114.7.3 Bounds checking expectations: `std::span` vs GSL . . . 1567
114.8 Tooling support for the Bounds profile . . . 1568
114.8.1 Clang-tidy (Core Guidelines checks) . . . 1568
114.8.2 MSVC C++ Core Guidelines checker . . . 1568
114.9 Design guidance: choosing between `span`, containers, and other views . . . 1568
114.9.1 Use `span` when you do not own . . . 1568

114.9.2 Use an owning container when you own 1568
114.9.3 Use `std::string_view` for text 1569
114.10 Summary 1569

115 Safe indexing **1570**
115.1 What "safe indexing" means 1570
115.1.1 Definition 1570
115.1.2 Why unchecked indexing is risky 1570
115.2 Bounds profile rules that define safe indexing expectations 1571
115.2.1 Bounds.2: Constant-expression indexing for arrays 1571
115.2.2 Bounds.4: Avoid unchecked standard access 1571
115.3 The safest pattern: avoid indexing entirely when possible 1572
115.3.1 Prefer range-for and algorithms 1572
115.4 When you must index: safe indexing strategies 1572
115.4.1 Strategy A: Use checked access (`at()`) when available 1572
115.4.2 Strategy B: Use a checked helper (`gsl::at`) under the Bounds profile 1573
115.4.3 Strategy C: Validate once, then index safely 1573
115.5 Index type safety: avoid signed/unsigned traps 1574
115.5.1 Use `size_t` for sizes, but be careful with negative indices 1574
115.5.2 Use `std::ssize` for signed loop indices when needed 1575
115.6 Safe indexing for arrays and `std::array` 1575
115.6.1 Follow Bounds.2 strictly for raw arrays 1576
115.6.2 Prefer `std::array` + algorithms, or `span` 1576
115.7 Safe indexing for containers under Bounds.4 1576
115.7.1 Avoid unchecked container access when tools are enforcing Bounds.4 1576
115.7.2 Practical rule of thumb 1576
115.8 Designing interfaces that make safe indexing the default 1577
115.8.1 Take `span` instead of pointer+length 1577

115.8.2 Prefer slicing over pointer arithmetic . . . 1577
115.9 Checklist: safe indexing in real code . . . 1577

XX GSL (Guidelines Support Library) 1579

116 GSL.1–GSL.6 (Guidelines Support Library) 1581
116.1 Purpose and scope of the GSL . . . 1581
116.1.1 What "GSL.1–GSL.6" means in this manual . . . 1581
116.2 GSL.1: Use GSL vocabulary types to make safety assumptions explicit . . . 1582
116.2.1 Rationale . . . 1582
116.2.2 Key idea . . . 1582
116.3 GSL.2: Use gsl::not_null to rule out null pointers . . . 1583
116.3.1 Rationale . . . 1583
116.3.2 Example: safer interface . . . 1583
116.3.3 Notes . . . 1583
116.4 GSL.3: Use gsl::span for contiguous sequences (avoid pointer+length) . . . 1584
116.4.1 Rationale . . . 1584
116.4.2 Important compatibility note: gsl::span vs std::span . . . 1584
116.4.3 Example: replace pointer+count . . . 1584
116.4.4 Example: subspans instead of pointer arithmetic . . . 1585
116.5 GSL.4: Mark ownership explicitly (prefer RAII; use owner<T*> only for legacy) 1585
116.5.1 Rationale . . . 1585
116.5.2 Example: legacy boundary with explicit ownership . . . 1586
116.5.3 Rule of thumb . . . 1586
116.6 GSL.5: Use Expects and Ensures to document and enforce contracts . . . 1587
116.6.1 Rationale . . . 1587
116.6.2 Example: checked preconditions and postconditions . . . 1587

116.6.3 Guidance 1587
116.7 GSL.6: Prefer GSL utilities for safe cleanup and safe narrowing; integrate tool support thoughtfully 1588
116.7.1 Safe cleanup: `finally` 1588
116.7.2 Safe narrowing: `narrow` and `narrow_cast` 1588
116.7.3 Indexing consistency: `gsl::index` 1589
116.7.4 Tool integration: checked access and suppression with justification . . 1590
116.7.5 Minimal policy for suppressions 1590
116.8 Summary checklist (practical adoption) 1591

117 span (GSL) 1592
117.1 What `gsl::span` is 1592
117.1.1 Definition 1592
117.1.2 Why the C++ Core Guidelines want it 1592
117.2 Relationship to `std::span` 1593
117.2.1 Two spans, one idea 1593
117.2.2 A key practical difference: bounds checking 1593
117.2.3 Choosing between them in real projects 1593
117.3 Construction: how to obtain a `gsl::span` 1594
117.3.1 From pointer + count 1594
117.3.2 From C arrays and `std::array` 1594
117.3.3 From contiguous containers 1595
117.4 Element access and bounds safety 1595
117.4.1 The problem: unchecked indexing is a common bug source 1595
117.4.2 GSL approach: contract-checked access 1596
117.4.3 Recommended checked access: `gsl::at` 1596
117.4.4 Safe indexing pattern: validate once, then work 1596
117.5 Slicing and subranges 1597

117.5.1 Why slicing matters . . . 1597
117.5.2 `first`, `last`, and `subspan` . . . 1597
117.6 Byte views and object representation . . . 1597
117.6.1 Viewing memory as bytes safely . . . 1598
117.7 Designing interfaces with `gsl::span` . . . 1598
117.7.1 Replace pointer+length parameters . . . 1598
117.7.2 Const-correctness . . . 1599
117.8 Lifetime rules: bounds safety is not lifetime safety . . . 1599
117.8.1 A span never extends lifetime . . . 1600
117.9 Tooling: why `gsl::span` fits the Guidelines ecosystem . . . 1600
117.9.1 Bounds profile enforcement . . . 1600
117.9.2 Practical consequence . . . 1601
117.10 Performance notes . . . 1601
117.10.1 Span is lightweight . . . 1601
117.10.2 Checks vs structure . . . 1601
117.11 Summary . . . 1602

118 `not_null` (GSL) 1603
118.1 What `gsl::not_null` is . . . 1603
118.1.1 Definition . . . 1603
118.1.2 What `not_null` does and does not do . . . 1603
118.2 Why `not_null` exists . . . 1604
118.2.1 The real problem: “nullable by default” . . . 1604
118.2.2 Tooling benefit (Core Guidelines checkers) . . . 1605
118.3 Core semantics and operations . . . 1605
118.3.1 Construction and enforcement . . . 1605
118.3.2 Access . . . 1605
118.3.3 Assignment and rebinding . . . 1606

118.4 When to use `not_null` . . . 1606
118.4.1 Best use cases . . . 1606
118.4.2 When not to use it . . . 1606
118.5 Comparing `not_null` with alternatives . . . 1607
118.5.1 Why not just use references? . . . 1607
118.5.2 Why not just use assertions? . . . 1607
118.5.3 What about smart pointers? . . . 1607
118.6 Practical usage patterns . . . 1608
118.6.1 Non-null parameter . . . 1608
118.6.2 Non-null return value . . . 1608
118.6.3 Data member that must never be null . . . 1608
118.6.4 Rebinding control with `const` . . . 1609
118.7 Common pitfalls and how to avoid them . . . 1609
118.7.1 Mistaking non-null for valid . . . 1610
118.7.2 Leaking nullable semantics into non-null APIs . . . 1610
118.7.3 Overusing `not_null` in low-level glue . . . 1610
118.8 Integration with Core Guidelines tools (MSVC examples) . . . 1610
118.9 Performance and build-mode considerations . . . 1611
118.9.1 Cost model . . . 1611
118.9.2 Engineering guidance . . . 1611
118.10 Summary . . . 1612

119 `owner` (GSL) 1613
119.1 What `gsl::owner` is . . . 1613
119.1.1 Definition . . . 1613
119.1.2 Key point: `owner` is not RAII . . . 1614
119.1.3 Nullability . . . 1614
119.2 Why `owner` exists . . . 1614

119.2.1 The Guidelines default: raw pointers are non-owning 1614
119.2.2 The preferred alternative: use RAII owners 1614
119.2.3 When owner<T*> is acceptable 1615
119.3 Core semantics: what owner<T*> means 1615
119.3.1 Ownership responsibility 1615
119.3.2 Transfer, not sharing 1615
119.3.3 What owner does not provide 1616
119.4 Recommended usage patterns 1616
119.4.1 Pattern 1: Mark ownership transfer at an interface boundary 1616
119.4.2 Pattern 2: Use a real owner internally; expose owner<T*> only when forced 1617
119.4.3 Pattern 3: Owning pointers inside resource-handle implementations . 1617
119.5 Tooling: how analyzers interpret owner 1618
119.5.1 MSVC C++ Core Guidelines checker: ownership diagnostics 1618
119.5.2 clang-tidy: ownership semantics check 1619
119.6 Common mistakes and how to avoid them 1619
119.6.1 Mistake 1: Thinking owner makes a raw pointer safe 1619
119.6.2 Mistake 2: Assigning raw pointers into owners 1619
119.6.3 Mistake 3: Deleting non-owners 1620
119.6.4 Mistake 4: Using owner instead of upgrading to RAII 1620
119.7 Practical guidance for real projects 1620
119.7.1 Adoption strategy 1620
119.7.2 Interoperability note 1621
119.8 Summary 1621

120 index (GSL) 1622
120.1 What gsl::index is 1622
120.1.1 Definition 1622

120.1.2 What it is not 1623
120.2 Why the Guidelines use a signed index type 1623
120.2.1 The core problem: signed/unsigned traps 1623
120.2.2 Why ptrdiff_t? 1623
120.3 When to use gsl::index 1624
120.3.1 Use it for indexing loops 1624
120.3.2 Use it for "not found" and similar sentinel results 1624
120.3.3 Use it for reverse indexing loops 1625
120.4 How to use gsl::index safely with standard containers 1626
120.4.1 Dealing with size() returning size_t 1626
120.4.2 Prefer expressing the algorithm without indexing 1626
120.4.3 When a checked access fits better 1627
120.5 Common mistakes and how to avoid them 1627
120.5.1 Mistake 1: converting a negative index to size_t 1627
120.5.2 Mistake 2: using gsl::index for sizes that cannot be negative 1627
120.5.3 Mistake 3: assuming gsl::index is always wide enough 1628
120.6 How gsl::index fits with modern standard C++ 1628
120.6.1 std::ssize as a standard alternative for signed sizes 1628
120.7 Summary 1628

121 gsl::Expects / gsl::Ensures 1630
121.1 Purpose and intent 1630
121.1.1 What these facilities are 1630
121.1.2 What they are not 1631
121.2 Semantics and failure behavior 1631
121.2.1 Fail-fast contract checking 1631
121.2.2 Microsoft GSL implementation: termination is the standard behavior 1631
121.2.3 Why termination (and not exceptions) 1632

121.3 How to use `Expects` correctly 1632
121.3.1 Place preconditions at the top of the function body 1632
121.3.2 Prefer expressing contracts using safer types 1633
121.3.3 Keep contracts simple, cheap, and side-effect free 1633
121.4 How to use `Ensures` correctly 1633
121.4.1 Use postconditions to document and verify guarantees 1633
121.4.2 Avoid postconditions that are expensive or redundant 1634
121.5 Relationship to standard C++ contracts 1634
121.5.1 Why `Expects/Ensures` are macros today 1634
121.5.2 Practical takeaway 1635
121.6 Debugging and diagnostics strategies 1635
121.6.1 Use a terminate handler for last-chance logging 1635
121.6.2 Do not use `Expects/Ensures` to validate user input 1636
121.7 Common patterns 1637
121.7.1 Indexing and bounds assumptions 1637
121.7.2 Non-null assumptions 1637
121.8 Common mistakes and how to avoid them 1638
121.8.1 Mistake: using contracts for recoverable conditions 1638
121.8.2 Mistake: writing contracts with side effects 1638
121.8.3 Mistake: relying on contracts for program logic 1638
121.9 Summary 1638

XXI Profiles **1640**

122 Type Safety Profile (Pro.safety) **1642**
122.1 Overview 1642
122.1.1 What a *profile* is 1642

122.1.2 Goal of the Type Safety Profile . . . 1642
122.1.3 Completeness requires Bounds and Lifetime profiles too . . . 1643
122.2 Profile summary (Type.1 – Type.8) . . . 1643
122.2.1 Type.1: Avoid casts . . . 1643
122.2.2 Type.2: Don't use `static_cast` to downcast . . . 1644
122.2.3 Type.3: Don't use `const_cast` to cast away `const` . . . 1645
122.2.4 Type.4: Don't use C-style or functional casts . . . 1646
122.2.5 Type.5: Don't use a variable before it has been initialized . . . 1647
122.2.6 Type.6: Always initialize a data member . . . 1648
122.2.7 Type.7: Avoid naked `union` . . . 1649
122.2.8 Type.8: Avoid varargs . . . 1650
122.3 Adoption strategy . . . 1651
122.3.1 Start with vocabulary and interfaces . . . 1651
122.3.2 Use tools to enforce the profile . . . 1651
122.3.3 Know what the profile does *not* guarantee . . . 1651
122.4 Practical notes and exceptions . . . 1652
122.4.1 Interoperability with C APIs and low-level code . . . 1652
122.4.2 About future profiles (safe arithmetic) . . . 1652
122.5 Summary . . . 1652

123 Bounds Safety Profile (Pro.bounds) 1654
123.1 Overview . . . 1654
123.1.1 What a profile is . . . 1654
123.1.2 Goal of the Bounds Safety Profile . . . 1654
123.1.3 The two main enemies . . . 1654
123.1.4 Completeness requires Type and Lifetime profiles too . . . 1655
123.2 Profile summary (Bounds.1 – Bounds.4) . . . 1655
123.2.1 Bounds.1: Don't use pointer arithmetic. Use `span` instead . . . 1655

123.2.2 Bounds.2: Only index into arrays using constant expressions 1656
123.2.3 Bounds.3: No array-to-pointer decay 1657
123.2.4 Bounds.4: Don't use standard-library types and functions that are not bounds-checked 1658
123.3 Design guidance: making bounds safety the default 1659
123.3.1 Prefer span at interfaces 1659
123.3.2 Validate once, then operate under an invariant 1660
123.3.3 Choose the right error model 1661
123.4 Tool enforcement (practical adoption) 1661
123.4.1 Clang-Tidy (cppcoreguidelines Pro.Bounds checks) 1661
123.4.2 MSVC C++ Core Guidelines checkers (CppCoreCheck) 1661
123.5 Common exceptions and how to handle them safely 1662
123.5.1 Low-level code and ABI boundaries 1662
123.5.2 Performance notes 1662
123.6 Summary 1663

124 Lifetime Safety Profile (Pro.lifetime) 1664
124.1 Overview 1664
124.1.1 What a profile is 1664
124.1.2 Goal of the Lifetime Safety Profile 1664
124.1.3 Profile summary 1665
124.1.4 Impact (why this matters) 1665
124.2 Lifetime.1 in practice: what counts as a "possibly invalid pointer" 1665
124.2.1 Common sources of invalid pointers and references 1665
124.2.2 A key guideline: the profile is about dereference 1666
124.3 Typical lifetime bugs (with safe rewrites) 1666
124.3.1 Dangling pointer to a local object 1666
124.3.2 Use-after-delete 1667

124.3.3 Iterator/reference invalidation 1668
124.3.4 `std::string_view` bound to a temporary 1669
124.4 How the profile is intended to be enforced 1670
124.4.1 What tools should diagnose 1670
124.4.2 Microsoft lifetime analysis (practical status) 1670
124.4.3 Relationship to other profiles 1670
124.5 Design techniques that make Lifetime.1 easier to satisfy 1671
124.5.1 Prefer RAII and owning smart pointers 1671
124.5.2 Avoid returning references/pointers to locals 1671
124.5.3 Be careful with non-owning "view" types 1671
124.5.4 Avoid storing references/iterators across invalidating operations . . . 1672
124.6 Summary 1672

125 Enforcement and Tooling (Profiles) 1673
125.1 Why enforcement matters 1673
125.1.1 Guidelines become real only when enforced 1673
125.1.2 Profiles are designed for tool enforcement 1674
125.2 Enforcement layers 1674
125.2.1 Layer 1: Compile-time diagnostics (warnings as errors) 1674
125.2.2 Layer 2: Static analysis (semantic and dataflow checking) 1675
125.2.3 Layer 3: Dynamic analysis (sanitizers and runtime checks) 1675
125.2.4 Layer 4: Code review with rule-based checklists 1676
125.3 Core Guidelines tool ecosystem 1676
125.3.1 Clang-Tidy (cross-platform, CI-friendly) 1676
125.3.2 Microsoft Visual Studio: C++ Code Analysis and CppCoreCheck . . 1677
125.3.3 Sanitizers (runtime defect detection) 1678
125.3.4 Debug-mode checked interfaces 1679
125.4 Mapping tools to profiles 1679

125.4.1 Type Safety Profile (Pro.safety) . . . 1679
125.4.2 Bounds Safety Profile (Pro.bounds) . . . 1679
125.4.3 Lifetime Safety Profile (Pro.lifetime) . . . 1680
125.5 A workable adoption plan for real code bases . . . 1680
125.5.1 Step 1: Establish a baseline . . . 1680
125.5.2 Step 2: Prevent regressions . . . 1680
125.5.3 Step 3: Refactor toward profile-friendly code . . . 1681
125.5.4 Step 4: Isolate unavoidable unsafe code . . . 1681
125.6 Suppressions and exceptions . . . 1682
125.6.1 Prefer local, documented suppressions . . . 1682
125.6.2 Avoid blanket disabling of profile checks . . . 1682
125.7 Measuring success . . . 1682
125.8 Summary . . . 1683

XXII Enforcement and Tooling 1684

126 `clang-tidy` 1686
126.1 What `clang-tidy` is . . . 1686
126.2 Where `clang-tidy` fits in Core Guidelines enforcement . . . 1686
126.2.1 Core Guidelines and profiles . . . 1686
126.2.2 Check naming and families . . . 1687
126.3 Fundamental requirement: compile commands . . . 1687
126.3.1 Why a compilation database is important . . . 1687
126.3.2 Generating `compile_commands.json` with CMake . . . 1687
126.3.3 If you do not have a compilation database . . . 1687
126.4 Selecting checks . . . 1688
126.4.1 Enabling and disabling checks with `-checks=` . . . 1688

126.4.2 Listing checks 1688
126.5 Configuration files: `.clang-tidy` 1689
126.5.1 How configuration is discovered 1689
126.5.2 Key configuration fields 1689
126.5.3 Inspecting the effective configuration 1689
126.5.4 A practical Core-Guidelines oriented `.clang-tidy` 1690
126.6 Fixes and refactoring support 1690
126.6.1 Applying fixes directly 1690
126.6.2 Exporting fixes for controlled application 1691
126.6.3 Formatting around fixes 1691
126.7 Automation scripts shipped with `clang-tidy` 1691
126.7.1 `run-clang-tidy.py`: run the whole database in parallel 1691
126.7.2 `clang-tidy-diff.py`: focus on changed lines 1692
126.8 Suppressing diagnostics safely 1692
126.8.1 Prefer code fixes over suppressions 1692
126.8.2 Generic suppression comments 1692
126.8.3 Suppression policy (recommended) 1693
126.9 Clang diagnostics inside `clang-tidy` 1694
126.9.1 Filtering compiler warnings and errors 1694
126.9.2 Warnings-as-errors 1694
126.10 Using `clang-tidy` effectively in CI 1694
126.10.1 Baseline first, then prevent regressions 1694
126.10.2 Recommended CI modes 1695
126.11 How `clang-tidy` relates to other LLVM tooling 1695
126.11.1 `clang-format` and formatting 1695
126.11.2 IDE integration 1695
126.12 Limitations and what `clang-tidy` cannot guarantee 1695

126.12.1 It depends on correct build flags . . . 1696
126.12.2 It is not a proof system . . . 1696
126.13 Summary . . . 1696

127 Static analysis 1698
127.1 Definition and scope . . . 1698
127.1.1 What static analysis means in C++ . . . 1698
127.1.2 What static analysis can and cannot guarantee . . . 1698
127.2 Main categories of static analysis . . . 1699
127.2.1 Compiler diagnostics (warnings) . . . 1699
127.2.2 Rule-based linters . . . 1699
127.2.3 Path-sensitive / dataflow static analyzers . . . 1700
127.2.4 Semantic security scanning (query-based analysis) . . . 1700
127.3 Why static analysis is essential for the Core Guidelines . . . 1700
127.3.1 Profiles are designed to be checkable . . . 1700
127.3.2 Where static analysis is strongest . . . 1701
127.4 Core tools you will meet in modern C++ workflows . . . 1701
127.4.1 `clang-tidy` (rule-based lint + fix-its) . . . 1701
127.4.2 Clang Static Analyzer (path-sensitive symbolic execution) . . . 1702
127.4.3 MSVC code analysis and Core Guidelines checkers . . . 1702
127.4.4 GCC static analyzer (`-fanalyzer`) . . . 1703
127.4.5 CodeQL (query-based static analysis for security) . . . 1704
127.5 Static analysis workflow patterns . . . 1704
127.5.1 Baseline then block regressions . . . 1704
127.5.2 Two CI modes: diff-focused and full . . . 1704
127.5.3 Triage discipline . . . 1704
127.6 How static analysis supports the Profiles . . . 1705
127.6.1 Type Safety Profile . . . 1705

127.6.2 Bounds Safety Profile . . . 1705
127.6.3 Lifetime Safety Profile . . . 1705
127.7 Limitations and best practices . . . 1706
127.7.1 Static analysis needs correct build information . . . 1706
127.7.2 Do not confuse “no warnings” with “no bugs” . . . 1706
127.7.3 Keep unsafe code small and audited . . . 1706
127.8 Summary . . . 1707

128 CI integration 1708
128.1 What CI integration means for the Core Guidelines . . . 1708
128.1.1 Goal . . . 1708
128.1.2 Why CI matters more than local checking . . . 1709
128.2 Core CI building blocks . . . 1709
128.2.1 Build metadata: `compile_commands.json` . . . 1709
128.2.2 A layered verification model . . . 1710
128.2.3 Two run modes: PR gate vs. full gate . . . 1710
128.3 Turning Core Guidelines Profiles into CI policy . . . 1710
128.3.1 Pick profiles as the enforcement baseline . . . 1710
128.3.2 Policy: “no new warnings” . . . 1711
128.4 Practical CI patterns for `clang-tidy` . . . 1711
128.4.1 Recommended `.clang-tidy` configuration . . . 1711
128.4.2 Run `clang-tidy` using the compilation database . . . 1712
128.4.3 PR-mode: check only changed lines . . . 1712
128.4.4 Nightly-mode: full-project run . . . 1712
128.4.5 Fail the CI job on selected checks . . . 1712
128.5 Sanitizers in CI (runtime enforcement) . . . 1713
128.5.1 What sanitizers contribute . . . 1713
128.5.2 CMake example: sanitizer build preset . . . 1713

128.6 Microsoft tooling in CI (Windows/MSVC) . . . 1714
128.6.1 MSVC Code Analysis and Core Guidelines checks . . . 1714
128.6.2 MSBuild property example . . . 1714
128.7 Security scanning in CI (CodeQL example) . . . 1714
128.7.1 What CodeQL adds . . . 1714
128.7.2 CI placement . . . 1715
128.8 Caching, artifacts, and reproducibility . . . 1715
128.8.1 Caching . . . 1715
128.8.2 Artifacts . . . 1715
128.8.3 Reproducibility rules . . . 1716
128.9 A reference CI layout . . . 1716
128.9.1 Recommended job matrix . . . 1716
128.9.2 Principle: treat safety checks as merge gates . . . 1716
128.10 Common failure modes (and fixes) . . . 1717
128.10.1 Wrong flags lead to noisy or missing diagnostics . . . 1717
128.10.2 Too many warnings cause developers to ignore CI . . . 1717
128.10.3 Third-party headers dominate findings . . . 1717
128.11 Summary . . . 1717

129 Gradual rule enforcement 1719
129.1 Why gradual enforcement is necessary . . . 1719
129.1.1 Large code bases cannot be converted all at once . . . 1719
129.1.2 Profiles enable incremental enforcement . . . 1719
129.2 Principles of gradual enforcement . . . 1719
129.2.1 Principle 1: Establish a baseline, then stop regressions . . . 1720
129.2.2 Principle 2: Enforce the highest-risk rules first . . . 1720
129.2.3 Principle 3: Prefer structural refactors over endless suppressions . . . 1720
129.2.4 Principle 4: Keep enforcement deterministic and reviewable . . . 1721

129.3 A staged adoption model 1721
129.3.1 Stage 0: Measure (no gates yet) 1721
129.3.2 Stage 1: Stop new violations in touched code 1721
129.3.3 Stage 2: Make a small rule set non-negotiable 1722
129.3.4 Stage 3: Expand coverage gradually 1722
129.3.5 Stage 4: Enforce full profiles (release gate) 1723
129.4 Baseline management strategies 1723
129.4.1 Strategy A: “No new warnings” 1723
129.4.2 Strategy B: “Budgeted debt reduction” 1723
129.4.3 Strategy C: “Module-by-module enforcement” 1724
129.5 Choosing what to enforce first 1724
129.5.1 A practical severity ladder 1724
129.5.2 Profiles-first recommendation 1724
129.6 Handling warnings: suppressions, exclusions, and waiver policy 1725
129.6.1 Prefer precise suppressions with justification 1725
129.6.2 Exclude third-party code deliberately 1725
129.6.3 Waivers should be time-bound 1725
129.7 CI mechanics for gradual enforcement 1726
129.7.1 Diff-focused enforcement for pull requests 1726
129.7.2 Full runs for scheduled pipelines 1726
129.7.3 Publishing results as CI artifacts (SARIF where supported) 1726
129.8 When gradual enforcement fails (and how to fix it) 1727
129.8.1 Failure mode: too many warnings, developers ignore CI 1727
129.8.2 Failure mode: blanket suppressions 1727
129.8.3 Failure mode: inconsistent tool versions 1727
129.9 Summary 1728

Appendices **1729**

129.10 Appendix B — Rule Classification by Risk 1744
129.10.1 What “Risk” Means in This Manual 1744
129.10.2 Risk Levels 1744
129.10.3 Fast Classification Heuristics 1745
129.10.4 Classification by Rule Family 1745
129.10.5 Critical Risk Rules (Must-fix First) 1747
129.10.6 High Risk Rules (Fix Early) 1748
129.10.7 Examples: Why These Risks Are Classified This Way 1749
129.10.8 Recommended Adoption Order (Risk-Driven) 1750
129.10.9 Practical Enforcement Notes 1751
129.11 Appendix C — Comparison with MISRA and AUTOSAR 1751
129.11.1 Scope and Purpose 1751
129.11.2 High-Level Comparison 1752
129.11.3 How to Read Differences Correctly 1753
129.11.4 Key Common Ground (Broad Overlap Areas) 1754
129.11.5 Typical Differences You Will See in Practice 1754
129.11.6 Practical Mapping: When to Prefer Which Guidance 1755
129.11.7 Illustrative Examples 1756
129.11.8 Guidance for Using the Three Together 1758
129.12 Appendix D — When and Why to Break a Rule 1758
129.12.1 Purpose of This Appendix 1758
129.12.2 Key Principles 1759
129.12.3 A Decision Checklist (Use Before Any Rule Exception) 1759
129.12.4 Legitimate Reasons to Break a Rule 1760
129.12.5 How to Break a Rule Safely 1761
129.12.6 A Concrete Example: Breaking a Concurrency Rule 1763
129.12.7 Documenting a Rule Exception (Template) 1764

129.12.8 When You Should *Not* Break a Rule 1764
129.12.9 Recommended Policy for This Book 1764
129.13 Appendix E — Real-World Industrial Examples 1765
129.13.1 What This Appendix Covers 1765
129.13.2 Reading Guide 1765
129.13.3 Example 1 — Buffer Parsing in Embedded / Devices (Bounds and Type Safety) 1766
129.13.4 Example 2 — Preventing Hidden Data Races in a Library (Concurrency) 1767
129.13.5 Example 3 — Ownership at ABI Boundaries (Libraries and Plugins) . 1768
129.13.6 Example 4 — Smart Pointer Parameters That Communicate Intent (API Design) 1771
129.13.7 Example 5 — Preventing Undefined Behavior in Class Hierarchies (Virtual Destructors) 1772
129.13.8 Example 6 — Non-Null Contracts at Call Sites (Defensive Interfaces) 1773
129.13.9 Example 7 — Deliberate Low-Level Escape Hatch (Rare Suppression) 1774
129.13.10 Industrial Adoption Pattern: A Practical Playbook 1775
129.14 How This Reference List Is Organized 1776
129.15 Primary Rule Source 1776

Bibliography **1777**
129.16 ISO/IEC C++ Language Standards (Normative Baselines) 1777
129.17 MISRA and AUTOSAR (Safety-Critical Coding Standards) 1778
129.18 Guidelines Support Library (GSL) (Support Types and Concepts) 1778
129.19 Tooling Documentation (Core Guidelines Enforcement) 1779
129.20 Standards Status and Publication Context (Official Committee/Community Sources) 1779

Author's Introduction

C++ is not a language that forgives assumptions.
It is a language that exposes intent, rewards discipline, and amplifies both skill and negligence. For more than four decades, C++ has remained at the core of systems where correctness, performance, and longevity are non-negotiable: operating systems, compilers, databases, financial engines, embedded systems, scientific platforms, and large-scale infrastructure.
The price of this power is responsibility.
The ISO C++ Core Guidelines exist because experience has shown—repeatedly and consistently—that many failures in C++ software do not originate from obscure language rules or compiler bugs. They originate from human decisions: inconsistent ownership models, unsafe interfaces, ambiguous lifetimes, unchecked assumptions, and architectural shortcuts that quietly accumulate risk over time.
This book is written from a single, uncompromising premise:
Professional C++ engineering is not about knowing the language syntax. It is about consistently applying proven rules that prevent entire classes of errors before they occur.
The ISO C++ Core Guidelines are not a style guide. They are not a collection of personal preferences. They are a distilled body of engineering knowledge derived from decades of real-world failures, postmortems, large-scale codebases, and language design experience. They encode answers to questions every serious C++ developer eventually faces:

- Who owns this object, and for how long?

- What assumptions does this interface impose on its users?
- Which errors are prevented by construction, and 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?

What makes the Core Guidelines unique is not that they introduce new language features. Rather, they provide a coherent framework for using existing C++ features—especially modern C++—in a way that is explicit, verifiable, and reviewable.
They align language rules, library design, ownership semantics, concurrency models, and error handling into a single engineering philosophy:

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

This book exists because the Guidelines themselves, while authoritative, are dense, cross-referenced, and intentionally concise. They assume a reader who already understands C++, but who seeks deeper clarity on *why* each rule exists, *what problem it prevents*, and *how it should be applied in real systems*.
Inside these pages, each guideline is treated not as an isolated rule, but as part of a larger system of constraints:

- Language rules are connected to object lifetime and the memory model
- API design rules are tied directly to ownership and error propagation
- Concurrency rules are grounded in data races, visibility, and correctness
- Performance rules are evaluated against real architectural trade-offs

This book is intentionally large because professional C++ is inherently large. There are no shortcuts that replace understanding. There are no abstractions that remove responsibility. There is only disciplined engineering.
Whether you are:

- maintaining a legacy codebase,
- designing new APIs,
- reviewing critical infrastructure code,
- teaching advanced C++,
- or building systems that must remain correct for decades,

the ISO C++ Core Guidelines provide a shared technical language—a common ground between developers, reviewers, and architects.
This book is not meant to be read once. It is meant to be consulted, revisited, argued with, and applied.
Its ultimate goal is simple:
To help C++ developers write code that is not merely clever or fast, but correct, maintainable, and worthy of long-term trust.

`Ayman Alheraki`

Book’s Introduction

What Are the ISO C++ Core Guidelines?

Purpose and Definition

The **C++ Core Guidelines** are a curated set of practical rules, recommendations, and best practices aimed at helping developers **use C++ well**, especially **modern ISO C++**. They focus on writing code that is safer, simpler to maintain, easier to review, and still aligned with C++ performance and zero-overhead principles. The Guidelines are published as a **living document** and are continuously updated by the community under the stewardship of recognized C++ experts.

Why They Exist

C++ is large and multi-paradigm. The same feature set that enables high performance and low-level control also enables accidental complexity, undefined behavior, brittle ownership, and subtle concurrency bugs. The Guidelines exist to:

- Establish **consistent, modern defaults** for writing and reviewing C++.
- Reduce common classes of defects (notably around **lifetime**, **bounds**, and **type** mistakes).
- Encourage designs that scale (clear interfaces, explicit ownership, predictable resource management).

- Make code **amenable to automated checking** where possible.

Status: What They Are (and What They Are Not)

Not the ISO Standard, but Strongly Aligned with It

The Core Guidelines are **not** the ISO C++ Standard. They do not define language rules; instead, they recommend **how to use the standard effectively**. Their content is written to track modern ISO C++ practice and is updated as the language and ecosystem evolve.

A Living, Community-Improved Document

The Guidelines explicitly describe themselves as a living document under continuous improvement, with identified editors and an open contribution model. This matters: unlike a printed book, the content can be refined as new guidance becomes clearer and as tool support improves.

Editors, Origins, and Community Input

The Guidelines were originally launched publicly in 2015 and have been led and edited by major figures in the C++ community. The current Core Guidelines page identifies **Bjarne Stroustrup** and **Herb Sutter** as editors, and the ISO C++ site announcement describes the work as a collaborative effort with contributions from multiple organizations.

How the Guidelines Are Organized

Rule Categories

The document is organized into thematic sections that typically include:

- **Philosophy and design** guidance (how to think about modern C++).
- **Interfaces** (how to design APIs that are safe and clear).
- **Resource management and ownership** (RAII, avoiding leaks, explicit lifetime).
- **Type safety and conversions** (preventing narrowing and misuse of casts).
- **Bounds safety** (preventing out-of-range access).
- **Concurrency and synchronization** (avoiding data races and fragile patterns).
- **Error handling** (expressing errors clearly and safely).
- **Performance and low-level optimization** (doing the right thing without premature harm).

Rules, Recommendations, and Rationale

A characteristic strength of the Guidelines is that they usually do more than state a rule. Many rules come with:

- A clear statement (what to do / avoid).
- A rationale (why).
- Examples of good and bad code.
- Notes on enforceability (whether tools can check the rule).

This format helps transform the Guidelines from a checklist into a **teachable engineering reference**.

The Three Core Safety Profiles

A major practical concept in the Guidelines is the idea of **safety profiles**—sets of rules that collectively target common categories of serious defects. Tooling and checkers frequently refer to these profiles:

- **Type safety** (avoid unsafe casts and narrowing, keep invariants explicit).
- **Bounds safety** (avoid out-of-range memory access).
- **Lifetime safety** (avoid dangling references/pointers and use-after-free patterns).

These profiles are widely referenced in tooling ecosystems and documentation.

GSL: The Guidelines Support Library

Why a Support Library Exists

Some guideline recommendations become dramatically easier to apply when code uses small, expressive helper types. That is the motivation for the **Guidelines Support Library (GSL)**: a set of lightweight types and utilities designed to help express intent (especially around ownership and range views) and make misuse harder.

Key Examples from GSL

Commonly used GSL components include:

- `gsl::not_null<T>` to express non-null pointer intent.
- `gsl::span<T>` as a view over contiguous sequences, making size explicit and enabling safer access patterns.

Enforcement in Practice: Static Analysis and Linters

Tool Support Is a Design Goal

Many guidelines are designed to be **tool-checkable**. In production environments, the Guidelines are often adopted alongside automated enforcement to reduce review burden and improve consistency.

Visual Studio C++ Code Analysis (Core Check)

Microsoft documents multiple rule sets that directly reference and enforce subsets of the Core Guidelines, including rules aligned with the **Bounds**, **Type**, and **Lifetime** profiles.

Clang-Tidy Checks

`clang-tidy` is a widely used C++ linter and static-analysis tool with an extensible architecture for diagnosing typical programming errors and style issues. It includes checks in the `cppcoreguidelines-*` family, which map to subsets of Core Guidelines recommendations.

A Small Demonstration: From "Permissive" to "Intent-Expressive"

The following short examples illustrate the *style* of transformation encouraged by the Guidelines: make ownership and preconditions explicit, and prefer safer interfaces.

Express Non-Null Intent

```
#include <gsl/gsl> // from the Guidelines Support Library (GSL)
```

```
int length_of(const gsl::not_null<const char*> p) {
    // p cannot be null by construction
    int n = 0;
    while (p.get()[n] != '\0') ++n;
    return n;
}
```

Prefer Range Views Over Pointer+Length Pairs

```
#include <gsl/span>

int sum(gsl::span<const int> xs) {
    int s = 0;
    for (int v : xs) s += v;
    return s;
}
```

How to Read and Use the Guidelines (Recommended Approach)

For teams and long-lived codebases, the Guidelines are most effective when adopted deliberately:

- Use them as **review language**: shared terminology for discussing safety and design.
- Start with **high-impact areas**: ownership/lifetime, bounds, and narrowing conversions.
- Add **automated checks** where feasible (tooling support is substantial but not complete).
- Prefer incremental adoption: new code follows the rules first; legacy code is modernized as it is touched.

This approach aligns with how major tool vendors and the community position Guidelines usage: as a practical, enforceable path to better C++ code over time.

What This Book Means by "Complete Reference Manual"

In this book, "complete reference" means:

- Covering the Guidelines as a coherent system (principles, rules, profiles, and patterns).
- Presenting rules in an engineer-friendly structure: **intent, rationale, correct usage, violations, and fixes**.
- Showing how to translate recommendations into code that is:
  - Idiomatic modern ISO C++
  - Compatible with real-world toolchains and checkers
  - Practical for large and long-lived systems

Objectives of This Reference

Why This Book Exists

The *C++ Core Guidelines* are a living, evolving set of rules and best practices for using modern ISO C++ effectively. They are practical by design: they aim to make real codebases safer, clearer, and easier to maintain, while still respecting C++ performance and zero-overhead principles.
Because the Guidelines are broad, frequently updated, and sometimes interpreted differently across teams, this reference manual exists to provide a single, consistent, engineering-focused way to **read**, **apply**, and **enforce** them in production C++ code.

Primary Objectives

Objective 1: Provide a Complete, Rule-Oriented Reference

This book is structured to be used as a day-to-day engineering reference:

- It follows the Guidelines as published in the canonical document, including their rule categories and intent.
- It preserves each rule's identity and meaning, so readers can map what they see in this book back to the original guideline text and its rationale.

Objective 2: Translate Rules into Repeatable Engineering Practice

Guidelines are most valuable when they change how code is written and reviewed. Therefore, for each major guideline group, this book aims to provide:

- The engineering motivation (what problem the rule prevents).
- The safe modern alternative (what to write instead).
- The typical failure patterns seen in real code.
- A practical migration strategy for legacy code.

Objective 3: Emphasize the Most Defect-Reducing Areas First

The Guidelines explicitly organize many rules into **profiles**—sets of related rules that target specific classes of defects and can be adopted incrementally.
This manual focuses heavily on the profiles that address the most common and most expensive bug families in C++:

- **Lifetime safety** (avoid dangling pointers/references, use-after-free patterns).
- **Bounds safety** (avoid out-of-range access).
- **Type safety** (avoid unsafe casts, narrowing, and type confusion).

Objective 4: Make Guidelines Enforceable, Not Just Advisable

A distinctive feature of the C++ Core Guidelines is their strong interest in **enforcement**. Tool support is not an afterthought; it is explicitly discussed in the Guidelines and implemented in major toolchains.
This reference maps guideline intent to practical enforcement paths, especially:

- **clang-tidy** checks, including the `cppcoreguidelines-*` families.
- **Microsoft C++ Code Analysis (C++ Core Check)** rule sets designed to enforce subsets of the Core Guidelines, including profiles such as Lifetime and Type rules.

Scope and Non-Goals

What This Reference Covers

This book covers the Core Guidelines as an engineering system:

- The purpose, structure, and adoption model (including the use of profiles).
- The most important rule families for large-scale and long-lived C++ systems.
- Practical examples that show how code changes when rules are applied.
- Tool-assisted enforcement approaches using mainstream compilers and analyzers.

What This Reference Does Not Claim

To remain correct and defensible, this book does not treat the Guidelines as:

- A replacement for the ISO C++ Standard.
- A guarantee that code is bug-free.
- A single rigid style guide that ignores project constraints.

Instead, it treats the Guidelines as a well-motivated body of best practices, designed to be adopted gradually and supported by tooling.

How This Book Is Organized

Rule-Centered Chapters with a Standard Template

Each guideline group in later parts of the book will follow a consistent template:

- **Intent**: what the guideline tries to achieve.
- **Risk Addressed**: what failures it prevents (UB, leaks, races, etc.).
- **Preferred Modern Form**: the modern C++ approach encouraged by the rule.
- **Bad vs. Good Examples**: minimal but realistic code transformations.
- **Enforcement Notes**: what tools can and cannot check reliably.
- **Migration Advice**: how to apply the rule safely in existing code.

A Practical Emphasis on Interfaces and Intent

Modern C++ safety is often achieved by making intent explicit in interfaces:

- Prefer expressing preconditions (e.g., non-null requirements).
- Prefer range-aware interfaces over raw pointer arithmetic.
- Prefer RAII and deterministic lifetime management.

The Guidelines Support Library (GSL) exists specifically to help express such intent with lightweight types such as `not_null` and `span`.

A Small Example of the Book's Approach

The examples below are not meant to "introduce GSL" in full; they demonstrate the style of transformation this manual will repeatedly apply: make intent explicit and reduce error surface.

Example 1: Make Non-Null a Type-Level Requirement

```
#include <gsl/gsl>

// Express intent: the parameter must not be null.
void process(gsl::not_null<int*> p) {
    *p = 42;
}
```

The GSL provides `not_null` to restrict a pointer-like value to non-null, making a common precondition explicit at the call boundary.

Example 2: Prefer Range Views Over Pointer+Length Pairs

```
#include <gsl/span>

int sum(gsl::span<const int> xs) {
    int s = 0;
    for (int v : xs) s += v;
    return s;
}
```

The GSL provides `span` as a view over a contiguous sequence, enabling clearer interfaces and reducing the chance of mismatched pointer/length usage.

Alignment with Modern Tooling and Current Direction

The Core Guidelines are supported by major toolchains:

- The Guidelines explicitly note tool support, including `clang-tidy` and Microsoft's Core Checkers.
- Microsoft documents how Visual Studio supports a subset of the Core Guidelines via its C++ code analysis tools.
- LLVM documents `clang-tidy` as an extensible framework for diagnosing and fixing typical programming errors, and its check list includes many `cppcoreguidelines-*` checks.

In addition, C++ safety profiles are also being actively discussed in WG21 as proposals for standard-enabled enforcement mechanisms (notably around bounds, type, and lifetime categories). Such proposals are not the Core Guidelines themselves, but they show that the broader C++ standards community continues to prioritize practical safety improvements.

How to Use This Book as a Reference

What This Chapter Helps You Do

This manual is designed to be used like an engineering reference, not as a book you must read strictly from page one to the end. The C++ Core Guidelines themselves are a living, continuously improved document, edited by recognized C++ experts, and intended for practical use in real codebases. Accordingly, this book is organized to help you:

- Find the relevant guideline quickly.
- Understand the intent and engineering rationale.
- Apply the rule correctly in production code.
- Enforce the rule with mainstream tooling where feasible (static analysis and linters).

How the Book Is Structured

Two Reading Modes: Study Mode vs. Lookup Mode

You can use this manual in two complementary ways:
Study mode (building a mental model):

- Read the early chapters in Part 0 to understand what the Guidelines are, why they exist, and how enforcement works.

- Then study the major guideline families in a deliberate order (interfaces, resource management, type safety, bounds, lifetime, concurrency).

Lookup mode (solving an immediate problem):

- Start from the symptom (a bug class, a review comment, or a static-analysis warning).

- Jump directly to the matching guideline family and rule discussion.

- Use the **Rule Template** sections (intent, rationale, violations, fixes, and enforceability notes) to apply a correct change quickly.

Rule-Centered Chapters

Later chapters are written around rule families and recurring defect patterns. This mirrors how the Core Guidelines present rules and how tools often implement checks.

The Standard Rule Template Used Throughout This Book

Whenever you open a rule-focused section, expect a consistent structure that is designed for rapid engineering decisions:

Intent

A one-sentence statement of what the guideline is trying to achieve.

Why It Matters

A practical explanation of the defect class it addresses (undefined behavior, lifetime errors, bounds violations, data races, brittle interfaces, etc.). The Core Guidelines emphasize such motivations and focus on preventing common, expensive failures.

Rule Statement and Scope

A precise description of what is recommended, including when the rule applies and where exceptions may be justified.

Bad Example and Good Example

Minimal code examples that show what changes and why. This book uses `minted` to keep examples readable and consistent.

Enforcement Notes

A realistic description of what tools can and cannot enforce reliably. Tool-enforceable rules are especially important because the Guidelines explicitly value enforcement and large-scale consistency.

Migration Strategy

A step-by-step approach for applying the rule safely in legacy code without destabilizing a large system.

How to Navigate by the Most Common Needs

This section provides the fastest entry points, based on what usually triggers guideline questions in real teams.

If You Are Doing Code Review

Use this manual as a shared review vocabulary:

- When you see unclear ownership, jump to the resource management and lifetime chapters.
- When you see unchecked indexing or pointer arithmetic, jump to bounds safety topics.
- When you see narrowing conversions or risky casts, jump to type safety topics.

The Core Guidelines themselves organize rules and profiles specifically to reduce recurring defect categories.

If You Are Fixing a Bug

Map the bug to a defect class:

- Use-after-free, dangling pointers, and unsafe reference lifetimes → **Lifetime safety**.
- Out-of-range access, mismatched pointer/size, unchecked container access → **Bounds safety**.
- Unsafe casts, narrowing, type confusion → **Type safety**.

This mapping aligns with the Core Guidelines safety profile direction and with the broader ongoing work on enforceable safety profiles in the C++ standards community.

If You Are Responding to Static-Analysis Warnings

If a warning comes from a guideline-related checker, use the book in the following order:

1. Look up the warning family (tool section).
2. Identify the guideline intent (rule section).
3. Apply the recommended transformation (good example).
4. Validate the change by tests and re-running the analyzer.

Microsoft documents that Visual Studio supports a subset of the Core Guidelines as part of its C++ Code Analysis tools. LLVM documents `clang-tidy` as an extensible tool for diagnosing and fixing typical programming errors, and its check list includes many `cppcoreguidelines-*` checks.

Using Tooling with This Book

clang-tidy: Practical Use for Guideline-Style Checks

`clang-tidy` provides named checks, including a large family under `cppcoreguidelines-*`. A common workflow is to enable selected checks incrementally.

```
# Example: .clang-tidy (illustrative subset)
Checks: >
  -*
  ,cppcoreguidelines-pro-bounds-*
  ,cppcoreguidelines-pro-type-*
  ,cppcoreguidelines-owning-memory
WarningsAsErrors: ''
```

This book uses the same check family names as documented by LLVM so you can quickly map a chapter topic to the tool's vocabulary.

Visual Studio: C++ Core Guidelines Checkers

Microsoft documents the **C++ Core Guidelines checkers** as part of its Code Analysis tools, supporting a subset of guideline rules. When this book discusses a rule that maps to a supported checker, the rule section will include a short note referencing the relevant checker group or warning family.

A Recommended Adoption Plan for Teams

The Core Guidelines are broad, and large codebases rarely adopt everything at once. This manual is designed to support incremental adoption.

Phase 1: Establish Safe Defaults for New Code

Begin by applying guideline principles to new or actively modified code:

- Prefer RAII and deterministic lifetime management.
- Make ownership and preconditions explicit at interfaces.
- Avoid unchecked indexing and pointer arithmetic where practical.

These themes are central in the Core Guidelines.

Phase 2: Adopt High-Impact Safety Topics

Focus next on the categories that most directly reduce severe failures:

- Lifetime safety.
- Bounds safety.

- Type safety.

These categories align with the Core Guidelines emphasis on safety and with active standards proposals aimed at enforceable profiles.

Phase 3: Turn On Enforcement Gradually

Enable checks incrementally, avoid overwhelming a team with a flood of warnings, and treat enforcement as a long-term quality program. Both Microsoft and LLVM tool ecosystems support guideline-aligned checking, but not every rule is fully automatable.

How to Use Examples in This Book

Examples Are Minimal, but They Are Not Toy Problems

Each example is designed to represent a real pattern seen in production systems. The goal is to show:

- What the risky pattern looks like.
- What the guideline-recommended replacement looks like.
- Why the replacement reduces defect risk.

A Typical Transformation Pattern

Many guideline-driven improvements follow the same approach: move from implicit, fragile assumptions to explicit, checkable intent.

```
// Before: implicit contract (caller must pass valid pointer and correct length)
int sum(const int* p, int n) {
```

```
    int s = 0;
    for (int i = 0; i < n; ++i) s += p[i];
    return s;
}
```

```
// After: explicit range view (intent is clearer; size travels with the view)
#include <gsl/span>

int sum(gsl::span<const int> xs) {
    int s = 0;
    for (int v : xs) s += v;
    return s;
}
```

This style reflects the Core Guidelines focus on safer interfaces and on making intent visible to both humans and tools.

How to Cross-Reference with the Canonical Guidelines

The Core Guidelines are published as a living document. This manual is written to remain aligned with the canonical structure and terminology:

- Chapter sections mirror the major guideline families.
- Rule discussions preserve the original rule intent and the standard vocabulary.
- Tooling references use the official names as documented by tool vendors.

What to Do When a Rule Seems to Conflict with Your Constraints

In real systems, trade-offs exist:

- Performance constraints.
- ABI constraints.
- Legacy code and migration risk.
- Platform limitations.

When a rule conflicts with constraints, use the book's template:

1. Re-check the **intent** of the rule.
2. Prefer an alternative that preserves the intent.
3. Document the exception and isolate it.
4. Consider enabling enforcement elsewhere, even if one location must remain exceptional.

This matches the Guidelines' practical spirit: better defaults, measurable reduction of risk, and incremental improvement.

Guidelines vs. Mandatory Rules

Why This Distinction Matters

In professional C++ development, engineers often use the word *rule* for many different things:

- Requirements of the **ISO C++ Standard** (what the language and library actually guarantee).
- Requirements of **your organization** (coding standards, security policies, certification constraints).
- Recommendations from expert guidance such as the **C++ Core Guidelines**.

These are not equivalent. Confusing them leads to two common failures:

- Treating *guidelines* as if they were *language law*, which blocks pragmatic engineering.
- Treating *mandatory requirements* as optional style, which creates undefined behavior, security risk, or nonconforming products.

Mandatory Rules: What "Must" Means in C++ Engineering

ISO C++ Standard Requirements (Language and Library)

The ISO C++ Standard defines what programs mean, which constructs are well-formed, and what behavior is guaranteed. If code relies on **undefined behavior**, the implementation is not required to behave predictably, and the program is outside the safety envelope of the language model.

Conformance Language and Normative Wording

International Standards distinguish requirements from recommendations using established wording conventions. In ISO normative documents, **“shall”** indicates a requirement, while **“should”** indicates a recommendation. Documents without requirements (i.e., without “shall”) are not intended for conformity assessment.
This matters because:

- The ISO C++ Standard is a normative specification used to define conformance.
- The C++ Core Guidelines are not written as a conformance standard; they are practical guidance.

Organizational and Regulatory Requirements

In many environments, additional mandatory constraints exist:

- Security policies (for example, restrictions on unsafe APIs).
- Safety standards and certification rules (aviation, automotive, medical).
- Platform constraints (ABI stability rules, forbidden language features, restricted dynamic allocation).

These requirements can be stricter than the ISO C++ Standard and are binding for the project even if the language would allow a construct.

Guidelines: What "Should" Means in the Core Guidelines

Guidelines Are Expert Recommendations, Not Language Law

The **C++ Core Guidelines** describe themselves as a set of guidelines, rules, and best practices for using C++ well, with a focus on modern ISO C++. They aim to help developers write code that is simpler, safer, and more maintainable, while preserving performance goals.
A guideline typically expresses:

- A **preferred default** for most codebases.
- A **risk model** (what defect class it reduces).
- An **enforcement strategy** (how tools can help detect violations).

Profiles: Organized Adoption Rather Than All-or-Nothing

The Core Guidelines introduce the idea of **profiles**: sets of related guidelines that target specific problem areas and can be adopted incrementally (for example, profiles related to type safety, bounds safety, and lifetime safety).

A Practical Taxonomy: Four Kinds of "Rules" You Will Encounter

To use this book correctly, always identify which category a statement belongs to:

Category A: Language-Model Requirements

These are non-negotiable if you want defined behavior. Examples:

- Avoid undefined behavior (e.g., out-of-bounds access, use-after-free).
- Respect object lifetime rules.
- Follow the One Definition Rule (ODR) constraints.

Category B: Contractual/Regulatory Requirements

These are mandatory because your project environment requires them. Examples:

- “No dynamic allocation after initialization.”
- “No exceptions across module boundaries.”
- “Use only approved crypto libraries.”

Category C: Core Guidelines Recommendations

These are strongly recommended defaults, designed to reduce defects and improve maintainability. Examples:

- Prefer RAII for resource management.
- Make ownership and preconditions explicit at interfaces.
- Prefer safer range-based interfaces over raw pointer arithmetic.

Category D: Local Style Conventions

These help consistency but often have lower safety impact. Examples:

- Naming conventions.
- Header ordering.
- Formatting preferences.

When Guidelines Are "Effectively Mandatory"

Although guidelines are not language requirements, many teams make selected guidelines mandatory by policy:

- A team can adopt a profile and enforce it in CI.
- A safety-critical project can promote rules that reduce memory corruption into non-negotiable standards.

This is common because the Core Guidelines were designed with tool enforcement in mind. The Guidelines explicitly discuss tool support, including `clang-tidy` and Microsoft's C++ Core Guidelines checker rules.

Enforcement: How Tools Treat Guidelines vs. Mandatory Rules

Compilers Enforce the Language Rules

Compilers enforce syntax, well-formedness rules, and diagnostics required by the standard (within the limits of what can be detected at compile time). However, many critical issues (especially undefined behavior) are not guaranteed to be diagnosed.

Static Analysis and Linters Enforce Guideline Subsets

clang-tidy is documented as a clang-based linter framework for diagnosing typical programming errors and certain bug patterns deduced by static analysis. It includes checks in the `cppcoreguidelines-*` family, mapping to subsets of Core Guidelines recommendations. Microsoft documents that Visual Studio supports a subset of the Core Guidelines via its C++ Code Analysis tools (the C++ Core Guidelines checkers).

Examples: “Legal C++” vs. “Recommended C++”

Example 1: Code Can Be Legal but Still a Guideline Problem

This compiles and may work, but it hides a precondition (the pointer must not be null) and invites misuse:

```
void set_value(int* p) {
    *p = 7; // if p is null, behavior is undefined
}
```

A guideline-driven interface makes the precondition explicit. One common approach is to use a non-null wrapper type (commonly provided by the Guidelines Support Library ecosystem) to represent intent.

```
#include <gsl/gsl>

void set_value(gsl::not_null<int*> p) {
    *p = 7; // p cannot be null by construction
}
```

Example 2: A Mandatory Rule Violation (Undefined Behavior)

Out-of-bounds access is not merely “bad style”; it can trigger undefined behavior and memory corruption:

```
#include <vector>

int read_first(const std::vector<int>& v) {
    return v[0]; // undefined behavior if v is empty
}
```

A guideline-oriented fix is to make the precondition explicit or handle the empty case:

```cpp
#include <vector>
#include <stdexcept>

int read_first(const std::vector<int>& v) {
    if (v.empty()) throw std::out_of_range("empty");
    return v[0];
}
```

The key point is the classification:

- The **requirement** is: do not perform out-of-bounds access.
- The **guideline** helps you design interfaces and checks to avoid ever getting there.

How This Book Handles Exceptions and Trade-offs

Guidelines Allow Justified Deviations

The Core Guidelines are intended for practical adoption. In real systems, you may face constraints such as ABI stability, legacy interoperability, or strict performance budgets. In such cases:

1. Identify the **intent** of the guideline.
2. Preserve the intent with the best available alternative.
3. Isolate and document deviations.
4. Prefer automated enforcement everywhere else.

Mandatory Rules Do Not Allow "Style-Based" Exceptions

If the issue is a language-model requirement (e.g., avoiding undefined behavior) or a regulatory requirement, exceptions are not matters of taste. They require a controlled engineering argument and usually a different design, not a waiver.

A Quick Decision Checklist

When you read a statement in this book, ask:

1. Is this a **language requirement** (risk of undefined behavior or ill-formed code)?
2. Is this a **project requirement** (policy, certification, platform constraints)?
3. Is this a **Core Guideline recommendation** (best practice that reduces defects)?
4. Can it be **tool-enforced**, and do we want to enforce it?

When and Why a Rule May Be Broken

The Core Principle: Not All "Rules" Are Equal

Before deciding to break any rule, you must first classify what kind of rule it is:

Category 1: Mandatory Requirements (Do Not Break)

Some requirements are non-negotiable if you want a correct and conforming program. In ISO drafting conventions, **"shall"** expresses a **requirement** from which **no deviation is permitted** if conformance is to be claimed, while **"should"** expresses a **recommendation**.
For C++ engineering, this means:

- If breaking the rule introduces **undefined behavior**, it is not a harmless exception. It is a correctness failure.
- If breaking the rule violates **project compliance constraints** (safety, security, certification, contractual requirements), it is typically unacceptable unless the governing process explicitly permits a deviation.

Category 2: Guidelines (May Be Broken With Justification)

The **C++ Core Guidelines** are explicitly a set of **guidelines, rules, and best practices** for using modern ISO C++ effectively. They are not the ISO Standard; they are expert guidance

designed for practical adoption, including selective adoption through **profiles**.
Therefore, a Core Guideline may be broken **only when a disciplined engineering justification exists** and the risk is controlled.

Why Guideline Exceptions Exist

Guidelines Are Defaults, Not Absolute Law

The Core Guidelines aim to improve safety, clarity, and maintainability in real codebases. However, real systems face constraints such as:

- ABI compatibility and binary stability constraints
- Legacy interoperability requirements
- Performance or latency budgets
- Toolchain limitations (specific compilers, targets, or analysis constraints)
- Specialized domains with stricter external standards

The Guidelines themselves emphasize practical adoption and the ability to apply **sets of related guidelines** (profiles) incrementally. This structure implicitly recognizes that not every guideline will be equally relevant in every codebase at every time.

Engineering Trade-offs Are Sometimes Real

A rule may be broken when:

- Following it would cause a measurable regression (performance, memory, latency) that violates requirements.

- Following it would break required compatibility (ABI, protocol, or platform contracts).
- The guideline cannot be applied locally without a larger refactor that is currently too risky.

In such cases, the correct approach is not informal permission. It is a controlled deviation.

Valid Reasons to Break a Core Guideline

This section lists common *valid* reasons, with the expectation that the deviation is documented, localized, and reviewed.

Reason 1: Compatibility Constraints (ABI, API, or Platform)

Examples:

- You must expose a C API boundary or interoperate with a stable ABI.
- You must keep a public interface unchanged for binary compatibility.

Typical pattern:

- Keep the constrained interface minimal.
- Convert immediately to safer internal types at the boundary.

Reason 2: Proven Performance Constraint

If a guideline-recommended abstraction measurably violates hard performance targets, an exception may be warranted, but only when:

- There is evidence (benchmarks, profiling, latency measurements).

- The exception is localized and does not spread.
- You keep the safer design everywhere else.

Reason 3: Incremental Migration of Legacy Code

Large C++ systems are rarely modernized all at once. If a guideline demands a broad refactor, it can be temporarily violated while you:

- Establish safer defaults for new code.
- Migrate high-risk areas first (lifetime, bounds, type).
- Enable enforcement gradually to avoid overwhelming noise.

This aligns with the Guidelines' explicit support for gradual adoption through profiles.

Reason 4: Tool Limitations and False Positives

Some rules are difficult to enforce perfectly by tools, or produce false positives in certain constructs. The Core Guidelines explicitly discuss tool support and name `clang-tidy` and Microsoft's Core Checkers as enforcement mechanisms, which implies that selective suppression may be needed in exceptional cases.

Invalid Reasons (What Is Not Acceptable)

"We Always Do It This Way"

Habit is not justification.

“It Compiles”

Compilation does not imply correctness. Many severe bugs exist in code that compiles cleanly.

Breaking Rules That Prevent Undefined Behavior

If breaking a rule creates potential undefined behavior (for example: out-of-bounds access, lifetime violations, or data races), it is not a safe exception. At minimum, the design must change so the behavior is defined and checkable.

A Professional Deviation Procedure

When an exception is necessary, use a procedure that scales to teams and audits.

Step 1: Identify the Rule and Its Intent

Write down:

- Which guideline you are deviating from
- What defect class it prevents
- What the guideline is trying to achieve

Step 2: Prove the Constraint

Provide objective evidence:

- ABI requirement (public interface contract, platform rule)
- Performance proof (profile, benchmark)
- Migration risk (refactor scope, risk analysis)

Step 3: Choose the Safest Possible Alternative

Preserve the intent as much as possible:

- Restrict the exception to the boundary layer
- Add precondition checks or strong types
- Encapsulate unsafe operations behind a narrow interface

Step 4: Localize, Isolate, and Document

The exception must be:

- Local (small scope)
- Isolated (contained behind an interface)
- Documented (reason, risk, mitigation, owner)

Step 5: Enforce Everywhere Else

A key property of a good deviation is that it does not weaken the whole codebase. Keep enforcement enabled for the rest of the project.

How to Document a Deviation in Code

A good deviation note is short, factual, and actionable. It should include:

- The rule/check being suppressed
- The reason (constraint)

- The mitigation (how risk is controlled)
- The exit plan (if temporary)

```
// Deviation: cppcoreguidelines-pro-bounds-pointer-arithmetic
// Reason: ABI boundary with legacy C API requires pointer arithmetic here.
// Mitigation: validated length; confined to this translation unit; covered by tests.
// Exit plan: replace once upstream API supports span-like interface.
void parse_legacy_buffer(const unsigned char* p, int n) {
    // ...
}
```

How to Break a Rule Without Breaking the Build

When deviations are justified, you still want your tooling to stay useful. This section shows *controlled suppression* mechanisms from mainstream tools.

clang-tidy: `NOLINT` and `NOLINTNEXTLINE`

LLVM documents that `NOLINT` suppresses warnings on the same line, and `NOLINTNEXTLINE` suppresses warnings on the next line.

```
int legacy_id(); // external contract

int id = legacy_id(); // NOLINT(cppcoreguidelines-avoid-non-const-global-variables)
```

```
// NOLINTNEXTLINE(cppcoreguidelines-pro-bounds-pointer-arithmetic)
auto byte = *(p + offset);
```

Use this sparingly:

- Suppress the narrowest scope possible (one line).
- Prefer suppressing a specific check, not all checks.

MSVC C++ Core Guidelines Checkers: Targeted Suppression

Microsoft documents ways to suppress specific analysis warnings for the Core Guidelines checkers (for example, disabling a specific warning number such as `C26400` via `/wd26400` for a file or project).

```
/wd26400
```

Microsoft also describes improvements in suppression mechanisms for their C++ code analysis, including `[[gsl::suppress]]` (intended for Microsoft C++ Code Analysis and Core Guidelines checks) and standard warning suppression mechanisms.

```
[[gsl::suppress(26400)]] // example: suppress a specific Core Guidelines checker warning
void legacy_api_adapter(int* p) {
    // justified deviation in a narrow scope
    *p = 1;
}
```

Rule Breaking That Is Actually Good Engineering

The highest-quality exceptions share these properties:

The Exception Preserves the Guideline's Intent

Even if the letter of the guideline cannot be followed, the intent can often be preserved through:

- Boundary conversion (unsafe outside, safe inside)
- Encapsulation (one isolated module)
- Explicit contracts (validated preconditions)

The Exception Is Reviewed and Owned

A deviation should have an owner. It is a design choice, not an accident.

The Exception Is Measurable and Testable

If the deviation exists for performance, you must keep the benchmark. If it exists for migration, you must keep a tracking plan.

Summary Checklist

A rule may be broken only when all of the following are true:

1. The rule is a **guideline**, not a mandatory requirement.
2. The constraint is real and documented (compatibility, performance, migration).
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 of the Philosophy Rules

The Philosophy rules are intentionally high-level. They provide the conceptual basis for the more concrete and mechanically checkable rules in later sections of the C++ Core Guidelines. Without this foundation, detailed rules can feel arbitrary and are harder to apply consistently.

1.2 How to Read P-Rules in Practice

1.2.1 They Are Principles, Not Micro-Rules

Most P-rules are not fully enforceable by tools in the general case. Instead, they guide design choices, interface design, and the selection of safer patterns and libraries.

1.2.2 They Lead to Checkable Code

A recurring theme is: prefer designs that make correctness *visible* and *checkable* by compilers, analyzers, tests, and runtime checks where needed.

1.3 P.1: Express Ideas Directly in Code

1.3.1 Rule

P.1: Express ideas directly in code.

1.3.2 Reason

Comments and design documents are often ignored or become outdated. What is expressed in code has defined semantics and can be checked by compilers and tools.

1.3.3 Example: Make Meaning Explicit

```
class Date {
public:
    Month month() const;  // do: explicit type and const
    int month();          // don't: unclear meaning and mutability
};
```

1.3.4 Prefer Library Expressions of Intent

```
#include <algorithm>
#include <vector>
#include <string>

void f(std::vector<std::string>& v, const std::string& val) {
```

```
    auto it = std::find(v.begin(), v.end(), val); // clearer intent than a manual loop
    // ...
}
```

1.3.5 Enforcement Notes

In general, this is hard to enforce directly. However, individual aspects can be checked, such as consistent `const` usage and suspicious casts.

1.4 P.2: Write in ISO Standard C++

1.4.1 Rule

P.2: Write in ISO Standard C++.

1.4.2 Reason

The Core Guidelines are written for ISO Standard C++. Extensions may be necessary in some environments, but they often have less rigorously defined semantics and can harm portability.

1.4.3 Guidance: Localize Extensions

If you must use compiler or platform extensions:

- Keep extension usage in small, well-defined areas.
- Encapsulate extensions behind portable interfaces.
- Ensure the rest of the codebase remains standard C++.

This approach is explicitly recommended by the Guidelines.

1.4.4 Enforcement Notes

Use an up-to-date compiler mode that rejects extensions where possible.

1.5 P.3: Express Intent

1.5.1 Rule

P.3: Express intent.

1.5.2 Reason

Code is read far more often than it is written. If intent is not visible, reviewers and maintainers cannot reliably judge correctness.

1.5.3 Example: Prefer Range-Based Loops

```
#include <vector>

void g(const std::vector<int>& v) {
    for (const auto& x : v) {
        // do something with x
    }
}
```

1.5.4 Enforcement Notes

Look for patterns where better alternatives express intent more clearly:

- simple `for` loops vs. range-`for`
- `f(T*, int)` interfaces vs. `f(span<T>)`-style interfaces

- too-wide scope for loop variables
- naked `new`/`delete`
- many parameters of built-in types where stronger types would help

1.6 P.4: Ideally, a Program Should Be Statically Type Safe

1.6.1 Rule

P.4: Ideally, a program should be statically type safe.

1.6.2 Reason

Full compile-time type safety is not possible in C++, but we should strive for it. Certain areas are known sources of crashes and security problems:

- unions
- casts
- array decay
- range errors
- narrowing conversions

1.6.3 Preferred Alternatives (Examples)

The Guidelines recommend replacing or minimizing these hazards where possible:

- use `std::variant` instead of unions

- minimize casts; use templates and better types
- use `span` to avoid array decay and enable bounds-aware access
- use `narrow` / `narrow_cast` when narrowing is necessary

1.7 P.5: Prefer Compile-Time Checking to Run-Time Checking

1.7.1 Rule

P.5: Prefer compile-time checking to run-time checking.

1.7.2 Reason

Compile-time checks improve clarity and performance, and eliminate the need for runtime error handling when a problem can be detected earlier.

1.7.3 Example: Replace a Run-Time Check with `static_assert`

```
#include <cstdint>

using Int = int32_t;
static_assert(sizeof(Int) >= 4); // compile-time check
```

1.7.4 Example: Prefer Interfaces the Compiler Can Help With

```
#include <gsl/span>

void read(gsl::span<int> r); // range carries its size
```

```
void demo() {
    int a[100]{};
    read(a); // better: compiler knows the array size
}
```

1.7.5 Enforcement Notes

Look for pointer arguments and run-time range checks that could be replaced by safer interfaces.

1.8 P.6: What Cannot Be Checked at Compile Time Should Be Checkable at Run Time

1.8.1 Rule

P.6: What cannot be checked at compile time should be checkable at run time.

1.8.2 Reason

Leaving hard-to-detect errors in a program invites crashes and wrong results. While not all errors can be caught at compile time, designs should allow practical run-time checking when needed.

1.8.3 Example: Do Not Hide the Size Information

```
extern void f(int* p); // bad: no size information

void g(int n) {
    f(new int[n]);     // bad: impossible for f to validate bounds
}
```

1.8.4 Better: Pass a Size-Carrying Type or a View

```
#include <vector>
#include <gsl/span>

extern void f4(std::vector<int>&);
extern void f4(gsl::span<int>);

void g3(int n) {
    std::vector<int> v(n);
    f4(v);                // pass a reference, retain ownership
    f4(gsl::span<int>{v}); // pass a view, retain ownership
}
```

This design keeps the element count as part of an object, making run-time checking feasible.

1.8.5 Enforcement Notes

A key signal is a `(pointer, count)` interface that prevents realistic checking.

1.9 P.7: Catch Run-Time Errors Early

1.9.1 Rule

P.7: Catch run-time errors early.

1.9.2 Reason

Early detection prevents mysterious crashes and avoids long chains of computation that produce silently wrong results.

1.9.3 Example: Prefer Early, Central Validation

Error-prone code:

```
void increment1(int* p, int n) {          // bad: error-prone interface
    for (int i = 0; i < n; ++i) ++p[i];
}
```

A safer direction is to use a size-aware view such that checks can happen earlier and more naturally at the boundary.

1.9.4 Performance Note

Early checking must still respect cost. Avoid checks that change the asymptotic complexity of an interface (for example, adding an `O(n)` check to an interface expected to be `O(1)` on average).

1.10 P.8: Don't Leak Any Resources

1.10.1 Rule

P.8: Don't leak any resources.

1.10.2 Reason

Even slow leaks eventually exhaust resources, especially in long-running programs. Resource leaks are a correctness and reliability failure.

1.10.3 Example: Prefer RAII

```
#include <fstream>
```

```
void f(const char* name) {
    std::ifstream input{name};  // RAII-managed resource
    // ...
    if (/*something*/) return;  // OK: no leak; destructor releases the handle
    // ...
}
```

1.10.4 Important Clarification from the Guidelines

The Guidelines note that a leak is best understood as "something that can no longer be cleaned up" (for example, losing the last pointer to a heap allocation).

1.10.5 Enforcement Notes

Focus on:

- classifying pointers into owners vs. non-owners
- replacing owners with standard resource handles where feasible
- flagging naked `new`/`delete`
- flagging known resource allocators returning raw pointers (e.g., `fopen`, `malloc`)

1.11 P.9: Don't Waste Time or Space

1.11.1 Rule

P.9: Don't waste time or space.

1.11.2 Reason

The Guidelines summarize the motivation concisely: *This is C++*. Efficiency is a design goal, but time and space spent to achieve safety, clarity, and testability is not automatically waste.

1.11.3 Example: Avoid Gratuitous Work

The Guidelines provide examples showing how manual memory management, redundant copying, and disabling move semantics can cause serious inefficiency.

1.11.4 Enforcement Notes

Many specific rules across the Guidelines support this goal. One example enforcement note is to flag unused return values from user-defined postfix increment/decrement and prefer prefix where appropriate.

1.12 P.10: Prefer Immutable Data to Mutable Data

1.12.1 Rule

P.10: Prefer immutable data to mutable data.

1.12.2 Reason

Immutable data is easier to reason about, enables optimizations, and eliminates data races on that data.

1.12.3 Practical Guidance

Make immutability the default:

- prefer `const` where mutation is not required
- use const-correct member functions and parameters
- design APIs that minimize shared mutable state

This is aligned with the Guidelines and reinforced by the dedicated section on constants and immutability.

1.13 P.11: Encapsulate Messy Constructs, Rather Than Spreading Through the Code

1.13.1 Rule

P.11: Encapsulate messy constructs, rather than spreading through the code.

1.13.2 Reason

Messy low-level code hides bugs, is harder to write correctly, and encourages more messy code. A good interface is easier and safer to use.

1.13.3 Example: Replace Manual Allocation Logic with `std::vector`

```
#include <vector>
#include <iostream>

std::vector<int> read_ints() {
    std::vector<int> v;
    v.reserve(100);
    for (int x; std::cin >> x; ) {
        // validate x if needed
```

```
        v.push_back(x);
    }
    return v;
}
```

This matches the Guidelines' example: use libraries that encapsulate complexity rather than repeating fragile low-level patterns.

1.13.4 Enforcement Notes

Look for complex pointer manipulation and casting outside the implementation of abstractions.

1.14 P.12: Use Supporting Tools as Appropriate

1.14.1 Rule

P.12: Use supporting tools as appropriate.

1.14.2 Reason

Many tasks are done better by machines. Tools do not get tired and can perform repetitive checking consistently.

1.14.3 Example: Use Static Analysis Intentionally

The Guidelines explicitly recommend running a static analyzer to verify that code follows the guidelines you want it to follow.

```
Example practice:
- Enable a defined set of guideline checks in CI.
- Treat new warnings as regressions.
- Fix issues incrementally in legacy code.
```

1.14.4 Caution

Avoid becoming dependent on over-elaborate or over-specialized toolchains that make otherwise portable code non-portable.

1.15 A Short Summary Checklist for P.1–P.12

- Prefer code that states meaning directly (P.1, P.3).
- Prefer standard, portable C++ and localize unavoidable extensions (P.2).
- Push correctness into types and compile-time checks where possible (P.4, P.5).
- Where compile-time checks are not possible, design for feasible runtime checks (P.6, P.7).
- Use RAII and avoid leaks (P.8).
- Respect performance, but do not confuse safety and clarity with waste (P.9).
- Prefer immutability to reduce reasoning burden and race risk (P.10).
- Encapsulate low-level complexity behind good interfaces and libraries (P.11).
- Use tools to raise baseline quality and consistency (P.12).

Chapter 2

Design Philosophy of C++

2.1 What This Chapter Covers

C++ was designed to let engineers build **high-performance, resource-aware, long-lived systems** while still supporting **abstraction** and **large-scale software design**. This chapter summarizes the design philosophy that repeatedly appears in authoritative C++ sources and is reflected throughout the C++ Core Guidelines.

2.2 Core Design Goals of C++

2.2.1 Direct, Efficient Use of Hardware

A defining characteristic of C++ is preserving a machine model that supports **direct and efficient use of hardware**. The intent is that fundamental operations and object layouts can map naturally to machine-level concepts, enabling predictable performance and control.

2.2.2 Abstraction Without Losing Performance

C++ aims to support **high-level abstraction** while keeping performance close to what a skilled developer could write by hand at a lower level. This goal is commonly summarized by the **zero-overhead principle**.

2.2.3 A Minimal Assumed Run-Time Environment

Another recurring design constraint is that C++ should not require a heavy runtime system. This supports embedded systems, kernels, HPC, games, real-time workloads, and other environments where runtime costs or dependencies must be controlled.

2.2.4 Long-Term Evolution and Stability

C++ is designed to evolve while protecting a vast existing codebase. This means the language must balance innovation with compatibility and predictable semantics. The Core Guidelines reflect this by focusing on *how to use modern ISO C++ well* rather than prescribing a new language.

2.3 The Zero-Overhead Principle

2.3.1 What It Means

A widely cited formulation of the zero-overhead principle is:

1. You do not pay for what you do not use.
2. What you do use should be as efficient as what you could reasonably write by hand.

This principle constrains language evolution: features should not impose hidden costs that programmers cannot control.

2.3.2 Why It Matters

This principle explains why C++ often prefers **static polymorphism** and compile-time composition (templates, inlining, concepts) where appropriate: the compiler can eliminate abstraction overhead.

2.3.3 Example: Zero-Overhead Abstraction via Templates

```
#include <vector>
#include <numeric>

template <class Range>
auto sum(const Range& r) {
    using T = typename Range::value_type;
    return std::accumulate(r.begin(), r.end(), T{});
}

int main() {
    std::vector<int> v{1,2,3,4};
    return sum(v); // abstraction: generic; cost: typically optimized away
}
```

2.4 Deterministic Resource Management (RAII)

2.4.1 Why RAII Is Central

C++ strongly favors **deterministic cleanup** through object lifetime and destructors. This supports robust systems programming by making resource release predictable even in the presence of early returns and exceptions. This emphasis is repeatedly described as part of the "C++ model," and it is a foundation of the Core Guidelines' advice on resource management.

2.4.2 Example: Resource Safety by Construction

```cpp
#include <fstream>
#include <string>

std::string read_first_line(const std::string& path) {
    std::ifstream in(path);       // acquires resource
    std::string line;
    std::getline(in, line);       // use resource
    return line;                  // releases resource automatically
}
```

2.5 Expressiveness with Strong Static Types

2.5.1 Make Intent and Invariants Visible

C++ encourages expressing intent in types and interfaces so that correctness becomes easier to review and (sometimes) tool-check. This is explicitly stated in the Core Guidelines philosophy rules (e.g., express ideas directly, express intent, prefer compile-time checking).

2.5.2 Example: Use Types to Express Intent

```cpp
#include <cstdint>

struct Meters {
    std::int64_t value;
};

struct Seconds {
    std::int64_t value;
};
```

```
Meters operator+(Meters a, Meters b) { return {a.value + b.value}; }
// Seconds + Meters would be a compile error (intent protected by types)
```

2.6 Multi-Paradigm by Design

2.6.1 Not One Style, but Controlled Choice

C++ intentionally supports multiple paradigms:

- procedural and low-level programming
- object-oriented design
- generic programming
- functional-style patterns (lambdas, algorithms)

The philosophy is not "pick one paradigm forever," but rather **use the right model where it fits** while keeping performance and correctness visible. The Core Guidelines are written to help teams use this flexibility safely and consistently.

2.6.2 Example: Algorithms Express Intent

```
#include <algorithm>
#include <vector>

bool any_negative(const std::vector<int>& v) {
    return std::any_of(v.begin(), v.end(), [](int x){ return x < 0; });
}
```

2.7 Prefer Compile-Time Guarantees When Possible

2.7.1 Shift Errors Left

A recurring C++ theme is catching problems as early as possible:

- compile-time errors are cheaper than runtime failures
- compile-time constraints improve documentation and correctness

This is reflected in the Core Guidelines philosophy rules and in the general direction of modern C++ design.

2.7.2 Example: Enforce Assumptions with `static_assert`

```
#include <type_traits>

template <class T>
T add(T a, T b) {
    static_assert(std::is_arithmetic_v<T>, "add() requires an arithmetic type");
    return a + b;
}
```

2.8 Safety as an Engineering Goal, Not a Separate Language

2.8.1 C++ Improves Safety Through Defaults, Libraries, and Tooling

C++ does not claim that all programs are automatically safe; instead, it provides mechanisms and encourages patterns that can be made safer through:

- RAII and deterministic lifetime

- safer interfaces and stronger types
- modern library facilities
- analysis tools and guideline-enforcing checks

The Core Guidelines explicitly emphasize **practical safety improvement** and **tool-assisted enforcement** (including clang-tidy guideline checks and Microsoft analyzer support).

2.8.2 Design Implication

Rather than enforcing a single mandatory style at the language level, C++ often enables multiple approaches and expects engineering teams to choose safe defaults, enforce them with tools, and localize exceptions when needed. This "engineering over ideology" approach is consistent with the Guidelines' profile-based adoption model.

2.9 Encapsulation of Low-Level Complexity

2.9.1 Abstractions Should Contain the Mess

C++ design favors keeping low-level details local and exposing clean interfaces. Stroustrup explicitly lists "encapsulate messy constructs"-style thinking as part of the enduring C++ approach to managing complexity, and the Core Guidelines restate this in philosophy rules.

2.9.2 Example: Hide Raw Memory Behind a Type

```
#include <vector>
#include <cstddef>

class Buffer {
    std::vector<unsigned char> data_;
```

```
public:
    explicit Buffer(std::size_t n) : data_(n) {}
    unsigned char* data() noexcept { return data_.data(); }
    std::size_t size() const noexcept { return data_.size(); }
};
```

2.10 Summary: The C++ Design Philosophy in One Page

C++ can be understood as a language engineered around these combined commitments:

- **Direct map to hardware** when needed.
- **Zero-overhead abstractions** to manage complexity without hidden cost.
- **Deterministic resource management** via lifetime and destructors (RAII).
- **Strong static typing and compile-time checking** whenever practical.
- **Multi-paradigm flexibility** guided by engineering intent.
- **Continuous evolution** while preserving stability and practical usability.

Chapter 3

Zero-overhead Abstraction

3.1 Definition and Meaning

3.1.1 The Zero-overhead Principle

A foundational design principle of C++ is the **zero-overhead principle**, often summarized in two statements:

1. You do not pay for what you do not use.
2. What you do use should be as efficient as what you could reasonably write by hand.

This formulation is presented in widely used C++ reference material and in ISO C++ community resources.

3.1.2 What the Principle Is (and Is Not)

Is:

- A **design constraint** on language and library features.

- A goal to enable **high-level abstraction** without hidden runtime and space costs beyond what a careful low-level implementation would require.

Is not:

- A guarantee that every abstraction is free in every build configuration.
- A claim that compilation time or binary size is always minimal.
- A substitute for measurement: if performance matters, you still benchmark and profile.

3.2 Why Zero-overhead Matters in C++

3.2.1 C++ Targets Performance-Critical Domains

C++ is widely used for systems programming and performance-critical software where predictable cost matters. The Core Guidelines capture this mindset explicitly in **P.9: Don't waste time or space** ("This is C++"), while also clarifying that time and space spent well to achieve goals such as safety and testability is not waste.

3.2.2 The Goal: Abstraction Without Surprise Costs

The core engineering promise is that you can write code that is:

- more readable and maintainable,
- safer by construction,
- still efficient after optimization,

because the language is designed so that many abstractions can be resolved at compile time or optimized away.

3.3 How C++ Achieves Zero-overhead Abstraction

3.3.1 Compile-time Composition (Templates and Inlining)

Many C++ abstractions are implemented through templates and inlining. When a generic function is instantiated with specific types, the compiler can generate specialized code and remove abstraction layers.

```
#include <vector>
#include <numeric>

template <class Range>
auto sum(const Range& r) {
    using T = typename Range::value_type;
    return std::accumulate(r.begin(), r.end(), T{});
}

int main() {
    std::vector<int> v{1,2,3,4};
    return sum(v); // typically compiles to a simple loop after optimization
}
```

3.3.2 RAII: Safety With Deterministic Cost

Resource Acquisition Is Initialization (RAII) uses constructors/destructors to manage resources deterministically. The abstraction cost is usually the same as the manual equivalent, but it is safer because cleanup is automatic along all control paths (including early returns and exceptions). This aligns with the Core Guidelines emphasis on avoiding leaks and writing maintainable, correct code without sacrificing performance goals.

```
#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;            // resource released automatically
}
```

3.3.3 Value Semantics and Move Operations

Modern C++ encourages value semantics where appropriate, relying on optimizations and moves to keep code both clean and efficient. The principle is not "avoid abstractions," but "choose abstractions whose costs are visible and optimizable."

3.4 Where Costs Appear: Abstraction Is Not Magic

3.4.1 Static vs. Dynamic Polymorphism

Static polymorphism (templates) is often optimizable away, but dynamic polymorphism (virtual functions) can add runtime indirection and can inhibit some optimizations. This is not "bad"; it is a trade-off chosen intentionally when you need runtime substitution.

```
struct IShape {
    virtual ~IShape() = default;
    virtual double area() const = 0; // dynamic dispatch cost is real and intentional
};
```

The key is that the cost is **explicit** (virtual dispatch, vtable), not hidden.

3.4.2 Type Erasure Can Add Indirection

Some convenient standard abstractions, such as `std::function`, typically involve type erasure, which may allocate or add indirection depending on implementation and callable size. Again, the point is not to forbid it, but to recognize it as a deliberate trade-off when you need runtime polymorphism of callables.

```
#include <functional>

void run(std::function<int(int)> f) { // may add indirection and other overhead
    (void)f(42);
}
```

3.4.3 Checks and Safety Are Not Automatically "Waste"

The Core Guidelines explicitly note under **P.9** that time and space spent well to achieve goals like resource safety or simpler testing is not wasted. The principle is not "never add checks," but "understand and control costs."

3.5 Zero-overhead in Practice: Rules of Thumb

3.5.1 Prefer Abstractions the Compiler Can See Through

In performance-sensitive paths:

- prefer templates and `constexpr` computations when they improve clarity and remove runtime work,
- prefer range-based loops and algorithms when they express intent and still compile efficiently,
- keep interfaces simple enough for inlining and optimization where it matters.

3.5.2 Measure, Then Decide

The zero-overhead principle is a guiding design rule, but engineering decisions still require measurement:

- Benchmark realistic workloads.
- Profile hot paths.
- Avoid speculative "micro-optimizations" in cold code.

3.5.3 Encapsulate "Paying" Abstractions

When you must use an abstraction with runtime cost (dynamic polymorphism, type erasure, synchronization, allocations), prefer to:

- localize it at system boundaries,
- keep hot inner loops free of unnecessary indirections,
- document the trade-off clearly.

3.6 Relationship to the Core Guidelines

The C++ Core Guidelines are aligned with the zero-overhead philosophy:

- They encourage expressing intent and using safer patterns.
- They explicitly value efficiency (**P.9**), while rejecting the false idea that all safety and clarity costs are waste.

In other words: modern C++ guidance is not about choosing between safety and performance. It is about learning how to achieve both with controlled, visible costs.

Chapter 4

Expressing Intent Clearly

4.1 What "Express Intent" Means

In the C++ Core Guidelines, **expressing intent** means writing code so that a reader (and often a tool) can quickly understand:

- what the code *is trying to do*,
- what assumptions it relies on (preconditions),
- what it guarantees (postconditions),
- who owns resources and how long objects live,
- which operations are safe, expected, or forbidden.

This idea is central in the Philosophy rules, especially **P.1** (express ideas directly in code) and **P.3** (express intent).

4.2 Why Clear Intent Is a Reliability Feature

Clear intent is not about style preferences. In real C++ systems, unclear intent is a frequent cause of:

- misuse of APIs (wrong units, wrong ownership, wrong lifetime),
- silent undefined behavior (dangling pointers, out-of-bounds access),
- fragile maintenance (changes that look safe but violate hidden assumptions),
- inconsistent code review decisions.

The Core Guidelines emphasize designs that are easier to reason about, easier to check, and safer by default, while still respecting performance constraints.

4.3 Principles for Expressing Intent

4.3.1 Prefer Code That States Meaning Directly

If intent matters, encode it in the language and library types, not in comments alone. Comments can help, but they can drift. Code has enforceable meaning. This is the motivation behind P.1 and P.3.

4.3.2 Prefer Interfaces That Make Misuse Hard

An interface that requires the caller to remember a hidden rule is an interface that will eventually be misused. The Guidelines repeatedly recommend choosing parameter types and return types that carry meaning: size, ownership, nullability, and error expectations.

4.3.3 Prefer Designs That Are Checkable

The Guidelines explicitly discuss tool support and show how subsets of rules can be enforced using mainstream tools such as `clang-tidy` and Microsoft C++ Code Analysis (Core Guidelines checkers).

4.4 Techniques: How to Make Intent Visible in C++

4.4.1 Technique 1: Use Names That Communicate Semantics

Names are a primary channel for intent. Prefer names that answer:

- what does this represent,
- in what units,
- under what constraints.

```
// Weak intent: What is "t"? milliseconds? ticks? seconds?
int t = 250;

// Stronger intent: units and meaning are visible.
int timeout_ms = 250;
```

4.4.2 Technique 2: Make Units and Domain Meaning Part of the Type

A strong type can prevent entire bug classes (e.g., mixing units). This shifts mistakes from runtime to compile time.

```
#include <cstdint>

struct Milliseconds { std::int64_t value; };
```

```
struct Bytes        { std::int64_t value; };

Milliseconds operator+(Milliseconds a, Milliseconds b) { return {a.value + b.value}; }

// Bytes + Milliseconds is a compile-time error (intent preserved by type).
```

4.4.3 Technique 3: Use `const` to Express Non-Mutation

Immutability communicates intent: "this does not change state." It also reduces reasoning cost and makes concurrency safer.

```
#include <string>

int count_digits(const std::string& s) {  // intent: read-only
    int n = 0;
    for (char c : s) if (c >= '0' && c <= '9') ++n;
    return n;
}
```

The Core Guidelines explicitly encourage preferring immutable data and using const-correctness to make intent clear.

4.4.4 Technique 4: Use RAII to Express Ownership and Lifetime

RAII expresses intent: "this object owns a resource and releases it deterministically." This makes ownership obvious and reduces leaks and cleanup bugs.

```
#include <fstream>
#include <string>

std::string load_line(const std::string& path) {
    std::ifstream in(path);   // intent: file is owned by 'in'
    std::string line;
```

```
    std::getline(in, line);
    return line;              // resource released automatically
}
```

4.4.5 Technique 5: Avoid Pointer Arithmetic; Prefer Size-Carrying Views

Pointer arithmetic is fragile because it hides bounds and makes checking difficult. The Core Guidelines include bounds-safety recommendations, and tool support reflects this. For example, `clang-tidy` provides checks such as `cppcoreguidelines-pro-bounds-pointer-arithmetic` and recommends safer alternatives like span.

```
// Weak intent: pointer + length is easy to mismatch.
int sum_raw(const int* p, int n) {
    int s = 0;
    for (int i = 0; i < n; ++i) s += p[i];
    return s;
}
```

```
#include <gsl/span>

// Stronger intent: the size travels with the view.
int sum(gsl::span<const int> xs) {
    int s = 0;
    for (int v : xs) s += v;
    return s;
}
```

4.4.6 Technique 6: Make Non-Null a Type-Level Requirement

If null is not a valid value, express that in the type. The Guidelines ecosystem commonly uses `gsl::not_null` for this purpose, and Microsoft C++ Code Analysis explicitly references Core Guideline F.23 in diagnostics related to `not_null`.

```
#include <gsl/gsl>

void write_value(gsl::not_null<int*> p) {
    *p = 42;  // intent: caller must provide a valid non-null target
}
```

4.4.7 Technique 7: Prefer Narrow, Scoped Enumerations Over Integers

An `enum class` communicates a closed set of meanings and prevents accidental mixing with unrelated integers.

```
enum class Mode { read_only, read_write };

void open_file(Mode m);

open_file(Mode::read_only); // clear intent
```

4.4.8 Technique 8: Replace "Magic Numbers" with Named Constants

A raw literal often hides meaning. Tooling reflects this practice: `clang-tidy` lists `cppcoreguidelines-avoid-magic-numbers` as an alias related to magic-number checks.

```
#include <cstddef>

constexpr std::size_t kMaxPacketBytes = 4096;

bool valid_packet_size(std::size_t n) {
    return n <= kMaxPacketBytes;
}
```

4.4.9 Technique 9: Prefer Algorithms and Range-Based Loops to Manual Control Flow

Algorithms often communicate the intent of the operation more directly than an ad-hoc loop.

```
#include <algorithm>
#include <vector>

bool any_negative(const std::vector<int>& v) {
    return std::any_of(v.begin(), v.end(), [](int x){ return x < 0; });
}
```

4.4.10 Technique 10: Keep Scopes Small and Bind Names Where They Are Used

Smaller scopes reduce accidental misuse and make it obvious where a name matters. The Guidelines contain rules such as keeping scopes small and declaring names in `for` initializers/conditions to limit scope.

```
#include <vector>

int find_first_positive(const std::vector<int>& v) {
    for (std::size_t i = 0; i < v.size(); ++i) {
        if (v[i] > 0) return static_cast<int>(i);
    }
    return -1;
}
```

4.5 Intent and Tooling: Making the Design Enforceable

4.5.1 clang-tidy and the `cppcoreguidelines-*` Checks

LLVM documents a large family of `clang-tidy` checks under `cppcoreguidelines-*`, including bounds- and type-related checks that are directly aligned with the Core Guidelines intent of making unsafe patterns visible and reducible.

4.5.2 Microsoft C++ Core Guidelines Checkers

Microsoft documents a Core Guidelines checker reference that maps specific diagnostics to specific guideline IDs (for example, several diagnostics reference guideline **F.23** and `gsl::not_null`).
The practical takeaway is:

- expressing intent in types (`not_null`, `span`-style interfaces, strong enums) improves readability,
- and it can also improve automated enforcement and consistency across large teams.

4.6 A Practical Checklist for Code Review

When reviewing C++ code, ask these intent questions:

1. Can I tell who owns each resource and how long it lives?
2. Are nullability and bounds requirements explicit at the interface boundary?
3. Are units and domain meanings encoded in names or types?
4. Can a maintainer misuse this function without noticing?
5. Can tooling check the critical assumptions (at compile time or with analyzers)?

This checklist aligns directly with the Core Guidelines emphasis on intent, checkability, and safer interfaces.

Chapter 5

Making Invalid States Unrepresentable

5.1 The Core Idea

A large part of writing reliable C++ is preventing bugs *before* they happen. One of the strongest ways to do that is to design your types and interfaces so that **invalid states cannot be represented in the first place**.
This philosophy is directly aligned with the C++ Core Guidelines emphasis on:

- expressing intent in code (P.1, P.3),
- striving for static type safety (P.4),
- preferring compile-time checking where possible (P.5),
- and using supporting libraries and tools to enforce safer patterns (P.12).

5.2 Why This Matters in C++

C++ gives you enough freedom to represent many nonsensical states:

- a null pointer where null is not acceptable,
- a length that does not match a buffer,
- a value in the wrong unit (bytes vs. elements, ms vs. s),
- a partially initialized object,
- a "mode" encoded as an `int` where any number is accepted.

When these states are representable, they eventually appear in real systems (especially long-lived codebases and evolving APIs). The goal is to shift these bugs from *runtime surprises* to *compile-time impossibilities*.

5.3 Principles: How to Make Invalid States Unrepresentable

5.3.1 Principle 1: Put Meaning in Types, Not Comments

If something has domain meaning (units, roles, constraints), represent that meaning in a type so misuse becomes difficult or impossible.

5.3.2 Principle 2: Narrow Interfaces to the Valid Domain

Prefer interfaces that only accept valid inputs. If an input can be invalid, represent that explicitly (e.g., optional values, error channels) rather than overloading a primitive type.

5.3.3 Principle 3: Carry Required Context Together

If a pointer needs a size, design a type that carries both together (e.g., `span`-style views). This supports checking and prevents mismatches. The Core Guidelines and tool ecosystem explicitly encourage bounds-aware designs and flag pointer arithmetic as fragile.

5.4 Common C++ Patterns That Encode Validity

5.4.1 Pattern 1: Scoped Enumerations Instead of Integers

If a parameter represents a small closed set of meanings, do not use `int`. Use `enum class`.

```
enum class Mode { read_only, read_write };

void open_file(Mode m);

open_file(Mode::read_only);   // valid
// open_file(2);              // impossible: invalid state is unrepresentable
```

The Core Guidelines explicitly recommend preferring `enum class` over plain `enum` for safety and clarity (Enum.3).

5.4.2 Pattern 2: Strong Types for Units and Domain Values

A strong type makes unit mistakes and role confusion compile-time errors.

```
#include <cstdint>

struct Bytes { std::int64_t value; };
struct Elements { std::int64_t value; };

Bytes operator+(Bytes a, Bytes b) { return {a.value + b.value}; }

// Bytes + Elements is a compile-time error: wrong domain.
```

This directly supports the Philosophy rules about expressing intent and aiming for static type safety.

5.4.3 Pattern 3: Non-null by Construction (`gsl::not_null`)

If null is not a valid value, do not accept a raw pointer that can be null.
Microsoft's C++ Core Guidelines checker documentation explicitly recommends using `gsl::not_null` to mark values that should never be null. Microsoft GSL describes `not_null` as restricting a pointer or smart pointer to non-null values.

```
#include <gsl/gsl>

void write_value(gsl::not_null<int*> p) {
    *p = 7;  // valid by type contract
}
```

Compared to an `assert(p != nullptr)`, this expresses the requirement at the interface boundary and is easier to keep consistent as code evolves.

5.4.4 Pattern 4: Size-Carrying Views (span) Instead of (Pointer, Length)

A pointer alone does not carry bounds information, which makes checking hard and mismatches easy. The Core Guidelines tool ecosystem flags pointer arithmetic as a common source of invalid pointers and recommends span<T> as a safer alternative.
Microsoft GSL describes `gsl::span` as a view over contiguous memory and notes that `gsl::span` enforces bounds checking (in the GSL implementation).

```
#include <gsl/span>

int sum(gsl::span<const int> xs) {
    int s = 0;
    for (int v : xs) s += v;
    return s;
}
```

Now “a range of ints” is representable; “a pointer with an unknown or mismatched length” is not part of the interface.

5.4.5 Pattern 5: Optional Values (`std::optional`) Instead of Sentinel Values

If something *may be absent*, represent absence explicitly instead of using magic sentinel values like `-1` or `nullptr` where null is not an acceptable object state.
`std::optional` is defined as managing an optional contained value that may or may not be present.

```
#include <optional>
#include <string>

std::optional<std::string> find_user_name(int id);

if (auto name = find_user_name(42)) {
    // present
} else {
    // absent
}
```

This makes the “missing” case part of the type, not a hidden convention.

5.4.6 Pattern 6: Closed Alternatives (`std::variant`) Instead of Unions or Ad-hoc Tags

If a value is *one of several types*, represent that explicitly with a type-safe union.
`std::variant` is specified as a type-safe union that holds one alternative at a time.

```
#include <variant>
```

```
#include <string>

using Token = std::variant<int, double, std::string>;
```

With `variant`, the invalid state "tag says string but payload is int" is eliminated by construction.

5.4.7 Pattern 7: Expected-or-Error (`std::expected`) Instead of Unstructured Error Codes

For operations that may fail in expected (recoverable) ways, `std::expected<T,E>` represents either an expected value or an unexpected error value, and it is specified to never be valueless.

```
#include <expected>
#include <string>

enum class ParseError { empty, invalid_digit };

std::expected<int, ParseError> parse_int(std::string_view s) {
    if (s.empty()) return std::unexpected(ParseError::empty);
    int v = 0;
    for (char c : s) {
        if (c < '0' || c > '9') return std::unexpected(ParseError::invalid_digit);
        v = v * 10 + (c - '0');
    }
    return v;
}
```

This makes error handling explicit and prevents the invalid state "returned value is meaningless but caller assumes success."

5.5 Encoding Invariants Inside Types

Not all invalid states can be eliminated solely with library wrappers. Many are domain-specific:

- a port number must be in a certain range,
- a percentage must be 0–100,
- a string must match a grammar,
- a buffer must be non-empty.

In these cases, the key technique is: **make construction the validation point**.

5.5.1 Private Construction + Validated Factory

```
#include <expected>
#include <cstdint>

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("port out of range");
        return Port(static_cast<std::uint16_t>(v));
    }

    std::uint16_t value() const noexcept { return value_; }
};
```

Now “a Port outside 0..65535” is not representable as a `Port` object. Only validated construction can produce a value.

5.6 Tooling Support and Enforceability

This philosophy is strongest when supported by tools:

- The Core Guidelines document discusses tool support and notes that `clang-tidy` has checks named `cppcoreguidelines-*`, and Microsoft code analysis provides Core Guidelines checkers.
- The clang-tidy check `cppcoreguidelines-pro-bounds-pointer-arithmetic` flags pointer arithmetic because it can lead to invalid pointers, and it points to `span<T>` as a safer type for array access.
- Microsoft documentation explicitly recommends `gsl::not_null` for resources that should never have a null value.

The practical outcome: once your interfaces represent validity in types, it becomes far easier for static analysis to flag violations early and consistently.

5.7 Checklist: Are You Making Invalid States Unrepresentable?

Use this checklist during API design and code review:

1. Are meaningful categories represented by `enum class` instead of integers?
2. Are units and roles represented by strong types instead of primitive aliases?
3. If null is invalid, is it impossible to pass null (e.g., `not_null`, reference, validated handle)?
4. If a range is required, does the interface carry size (`span`, container, view) rather than raw pointer arithmetic?

5. If absence is possible, is it represented explicitly (`optional`) rather than with sentinels?
6. If an operation may fail, is failure represented explicitly (`expected`) rather than via ambiguous return codes?
7. Are domain invariants enforced at construction, with invalid objects impossible to create?

Chapter 6

Clarity versus Cleverness

6.1 What This Chapter Means by “Clarity” and “Cleverness”

6.1.1 Clarity

In the C++ Core Guidelines, **clarity** means that the code’s intent, assumptions, and consequences are easy to see and hard to misunderstand. This aligns with the Philosophy rules **P.1** (express ideas directly in code) and **P.3** (express intent).

6.1.2 Cleverness

In this chapter, **cleverness** means writing code that is:

- shorter than necessary,
- dependent on subtle language rules (precedence, sequencing, conversions),
- impressive to the author but expensive to maintain,
- fragile under change, refactoring, or different compilers/settings.

The Guidelines explicitly warn against relying on subtle evaluation behavior and recommend avoiding complicated expressions.

6.2 Why Clarity Wins in Professional C++

6.2.1 Maintainability Scales, Cleverness Does Not

Most C++ code lives for years and is read far more than it is written. Clarity reduces:

- review time,
- bug density,
- onboarding cost,
- risk during refactoring.

The Core Guidelines promote designs and coding practices that make correctness easier to reason about and easier to verify with tools.

6.2.2 Clever Code Often Hides Undefined or Unspecified Behavior

Some clever one-liners depend on sequencing or evaluation order that is **undefined** or **unspecified**. The Guidelines explicitly address these hazards:

- **ES.43**: Avoid expressions with undefined order of evaluation.
- **ES.44**: Don't depend on order of evaluation of function arguments.

6.3 Guideline Anchors: Where the Core Guidelines Stand

6.3.1 Prefer Simplicity and Readable Structure

The Core Guidelines recommend avoiding complicated expressions (**ES.40**) and parenthesizing when precedence is not obvious (**ES.41**).

6.3.2 Prefer Abstractions Over Handcrafted Tricks

The Guidelines explicitly recommend:

- **ES.1**: Prefer the standard library to other libraries and to handcrafted code.
- **ES.2**: Prefer suitable abstractions to direct use of language features.

This is a direct clarity principle: well-known library algorithms communicate intent more clearly than custom clever loops.

6.3.3 Avoid Macro Cleverness

Macros are a classic source of cleverness-related defects. The Guidelines recommend:

- **ES.30**: Don't use macros for program text manipulation.
- **ES.31**: Don't use macros for constants or "functions".

6.4 Common "Clever" Patterns and Clear Alternatives

6.4.1 Pattern 1: Complicated Expressions

6.4.1.1 Problem

Complicated expressions compress logic into a form that is hard to review and easy to misread. The Guidelines directly address this with **ES.40**.

6.4.1.2 Example: Too Much in One Expression

```
// don't: compact but unclear; hard to debug
if ((x = f()) && (y = g(x)) && (z = h(y))) {
    use(z);
}
```

6.4.1.3 Clear Alternative: Name Steps

```
// do: explicit intent and easy breakpoints
x = f();
if (!x) return;

y = g(x);
if (!y) return;

z = h(y);
use(z);
```

6.4.2 Pattern 2: Precedence Tricks

6.4.2.1 Problem

Relying on subtle precedence is a common readability trap. The Guidelines say: **ES.41: If in doubt about operator precedence, parenthesize.**

6.4.2.2 Example

```
// don't: is this (a & b) == 0 or a & (b == 0)?
```

```
if (a & b == 0) { /* ... */ }
```

6.4.2.3 Clear Alternative

```
// do: make precedence explicit
if ( (a & b) == 0 ) { /* ... */ }
```

6.4.3 Pattern 3: Undefined or Unspecified Order of Evaluation

6.4.3.1 Problem

Some "clever" expressions depend on evaluation order. The Guidelines explicitly warn against it (**ES.43**, **ES.44**).

6.4.3.2 Example: Modifying and Reading in One Expression

The Guidelines include a specific warning and example:

```
// don't: undefined behavior (classic example)
v[i] = ++i;
```

They also note a practical risk: even though C++17 tightened some evaluation rules, code can still be compiled as pre-C++17 (for example via reuse or older toolchains), so "don't be too clever."

6.4.3.3 Clear Alternative

```
// do: separate steps; intent is obvious
++i;
v[i] = i;
```

6.4.3.4 Example: Function Arguments

```
// don't: order of evaluation of function arguments is unspecified
int i = 0;
f(++i, ++i);
```

```
// do: make sequencing explicit
int i = 0;
int a = ++i;
int b = ++i;
f(a, b);
```

The Guidelines explain that even in C++17 and later, the argument evaluation order remains unspecified, so you should not depend on it.

6.4.4 Pattern 4: Clever Loops That Hide Intent

6.4.4.1 Problem

Loop cleverness often hides the purpose and introduces subtle bugs. The Guidelines recommend using standard algorithms and suitable abstractions (**ES.1**, **ES.2**).

6.4.4.2 Example: Handcrafted Loop

```
// don't: verbose, purpose less clear
double sum = 0.0;
for (int i = 0; i < static_cast<int>(v.size()); ++i) {
    sum += v[i];
}
```

6.4.4.3 Clear Alternative: Standard Algorithm

```
#include <numeric>
```

```
// do: communicates intent directly
auto sum = std::accumulate(v.begin(), v.end(), 0.0);
```

The Guidelines explicitly use this style of example to illustrate why standard algorithms improve clarity.

6.4.5 Pattern 5: "Clever" Use of Loop Indices

The Guidelines explicitly discourage modifying loop control variables inside the body of raw `for` loops (**ES.86**) because it makes reasoning difficult and invites errors.

```
// don't: surprises readers and can break invariants
for (int i = 0; i < n; ++i) {
    if (skip(i)) ++i;
    use(i);
}
```

```
// do: make the stepping rule explicit
for (int i = 0; i < n; ) {
    if (skip(i)) { ++i; continue; }
    use(i);
    ++i;
}
```

6.4.6 Pattern 6: Macro Cleverness

Macros can create code that looks simple but behaves unexpectedly. The Guidelines recommend avoiding macros for program text manipulation and for constants/functions (**ES.30**, **ES.31**).

```
// don't: macro side effects and unclear evaluation
#define SQUARE(x) ((x) * (x))

int y = SQUARE(i++); // increments twice
```

```
// do: real function; single evaluation
constexpr int square(int x) noexcept { return x * x; }

int y = square(i++);
```

6.5 When “Clever” Is Acceptable

6.5.1 When It Is Local, Measured, and Documented

In performance-critical code, a less obvious technique may be justified if:

- it is measured and proven necessary,
- it is localized behind a clear interface,
- it is documented with the reason and constraints.

This is consistent with the Core Guidelines’ broader stance: prefer clarity by default, but allow justified exceptions where engineering constraints demand them.

6.5.2 A Good Rule of Thumb

If a reviewer must mentally simulate evaluation order, precedence, or implicit conversions to understand a line, that line is a clarity failure. The Guidelines address this directly through rules like ES.40, ES.41, ES.43, and ES.44.

6.6 Checklist: Enforcing Clarity in Daily Work

1. Replace handcrafted loops with standard algorithms when they express the intent (**ES.1**).
2. Avoid complicated expressions; name intermediate results (**ES.40**).

3. Parenthesize when precedence is not immediately obvious (**ES.41**).

4. Avoid undefined/unspecified evaluation order tricks (**ES.43**, **ES.44**).

5. Avoid macro cleverness; use language facilities instead (**ES.30**, **ES.31**).

Chapter 7

Engineering Simplicity

7.1 What “Engineering Simplicity” Means

Engineering simplicity is not about writing the shortest code, and it is not about avoiding advanced C++ features. It is about designing and writing code so that:

- intent is obvious,
- correctness is easy to argue and verify,
- maintenance cost stays low as the system evolves,
- performance costs are visible and controllable.

This perspective matches the C++ Core Guidelines philosophy rules that prioritize expressing ideas directly, expressing intent, avoiding waste, and encapsulating messy constructs.

7.2 Why Simplicity Is a Professional Requirement

7.2.1 Software Lives Longer Than Its Authors

In real systems, most cost is paid during change: bug fixes, feature additions, refactoring, porting, and onboarding new engineers. Simplicity reduces that cost because it reduces the number of assumptions a maintainer must hold in their head.

7.2.2 Simplicity Reduces Defect Density

Many expensive C++ failures come from hidden contracts:

- ownership and lifetime assumptions,
- pointer/length mismatches,
- subtle evaluation order,
- unclear error signaling.

The Core Guidelines repeatedly steer designs toward checkable contracts and safer interfaces, precisely because they reduce these failures at scale.

7.2.3 Simplicity Is Compatible with Performance

The Core Guidelines explicitly state that time and space spent well to achieve an engineering goal (such as resource safety or simplification of testing) is not "waste." This is a key point: engineering simplicity is often the fastest way to reach reliable performance.

7.3 Guideline Anchors for Simplicity

Engineering simplicity in this book is grounded in the following Core Guidelines themes:

7.3.1 Express the Idea, Not the Mechanism

P.1 and **P.3** emphasize expressing ideas directly and expressing intent. Simplicity follows when the code states *what* it means, not merely *how* it happens to work.

7.3.2 Encapsulate Messy Constructs

P.11 recommends encapsulating messy constructs rather than spreading them through the codebase. This is simplicity at system scale: concentrate the complexity in one place and keep most code clean.

7.3.3 Prefer Suitable Library Abstractions

The Guidelines encourage preferring the standard library and suitable abstractions over handcrafted code in many cases, because well-known abstractions communicate intent and reduce error surface.

7.4 Principles of Engineering Simplicity in C++

7.4.1 Principle 1: Make the Valid Path the Easy Path

If the correct usage is the simplest usage, misuse becomes rare. This is why designs that make invalid states unrepresentable (strong types, non-null types, size-carrying views) are not “extra abstraction”; they are simplicity through prevention.

7.4.2 Principle 2: Keep Invariants Local

Prefer types whose constructors establish invariants and whose public functions preserve them. This reduces the need for scattered checks and repetitive defensive code.

7.4.3 Principle 3: Prefer Narrow Interfaces with Clear Contracts

A narrow interface is simpler because there are fewer ways to use it incorrectly. In C++, contract clarity often comes from choosing the right types (references vs. pointers, `span`-style views vs. pointer+length, `optional`/`expected` vs. sentinel values).

7.4.4 Principle 4: Separate Policy from Mechanism

When you mix policy (business rules) with mechanism (memory management, parsing details, platform workarounds), every change becomes risky. Encapsulation is the simplicity tool for that separation.

7.4.5 Principle 5: Use Tools to Maintain Simplicity

The Guidelines explicitly recommend using supporting tools as appropriate. Tools keep code simple over time by preventing slow drift into inconsistent patterns.

7.5 Common Complexity Traps and Simpler Alternatives

7.5.1 Trap 1: Pointer + Length Interfaces

Pointer+length is easy to get wrong and hard to check globally.

```
// Harder to reason about: caller must keep (p, n) consistent.
int sum_raw(const int* p, int n) {
    int s = 0;
    for (int i = 0; i < n; ++i) s += p[i];
    return s;
}
```

A size-carrying view expresses the actual concept: “a contiguous range of ints”.

```
#include <gsl/span>

int sum(gsl::span<const int> xs) {
    int s = 0;
    for (int v : xs) s += v;
    return s;
}
```

This supports **P.3** (intent) and **P.6** (what cannot be checked at compile time should be checkable at run time) because a view carries bounds information.

7.5.2 Trap 2: Hidden Ownership and Manual Cleanup

Manual cleanup logic is a classic source of complexity (early returns, exceptions, partial initialization).

```
// Error-prone: cleanup must be remembered on every path.
FILE* f = std::fopen(name, "rb");
if (!f) return;
if (bad_input()) { std::fclose(f); return; }
use(f);
std::fclose(f);
```

RAII expresses ownership directly and makes cleanup automatic, which is simpler and safer.

```
#include <fstream>

void use_file(const char* name) {
    std::ifstream in(name);  // RAII
    if (!in) return;
    if (bad_input()) return; // no leak
    use(in);
}
```

This aligns with **P.8** (don't leak resources) and **P.11** (encapsulate messy constructs).

7.5.3 Trap 3: Sentinel Values for "Not Found" or "No Value"

Sentinel values force every caller to remember conventions and edge cases.

```
// Convention-heavy: -1 means "not found".
int find_user_id(std::string_view name);
```

An explicit type communicates meaning and enforces handling.

```
#include <optional>
#include <string_view>

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

This is simplicity because the type describes the domain: *a value may be absent*.

7.5.4 Trap 4: Overly Clever Expressions

Cleverness compresses logic into forms that are difficult to debug and easy to misunderstand. The Guidelines explicitly caution against complicated expressions and subtle evaluation behavior.

```
// Hard to read, hard to debug:
if ((x = f()) && (y = g(x)) && (z = h(y))) {
    use(z);
}
```

```
// Simpler and more maintainable:
x = f();
if (!x) return;
y = g(x);
if (!y) return;
z = h(y);
use(z);
```

7.6 Simplicity Patterns That Scale

7.6.1 Pattern 1: Small, Single-Purpose Functions

Simplicity is improved when each function has one clear responsibility, one clear error policy, and a narrow interface.

7.6.2 Pattern 2: Convert at the Boundary

When forced to interact with complex or unsafe interfaces (legacy C APIs, ABI constraints), keep that complexity at the boundary and convert immediately to safer internal types. This is a practical application of **P.11**.

7.6.3 Pattern 3: Prefer Strong Types for Domain Values

If a parameter represents units, roles, or restricted values, represent that in types. This supports **P.4** (static type safety) and **P.5** (prefer compile-time checking).

```
#include <cstdint>

struct Milliseconds { std::int64_t value; };
struct Bytes        { std::int64_t value; };

void set_timeout(Milliseconds t);
void set_buffer_limit(Bytes n);
```

7.6.4 Pattern 4: Prefer a Standard Algorithm When It Matches the Intent

A standard algorithm often says what you mean more clearly than a custom loop, and it reduces opportunities for indexing and boundary mistakes. This approach is consistent with the Guidelines' preference for suitable library abstractions.

```cpp
#include <algorithm>
#include <vector>

bool any_negative(const std::vector<int>& v) {
    return std::any_of(v.begin(), v.end(), [](int x){ return x < 0; });
}
```

7.6.5 Pattern 5: Make "Fast" Code Simple by Design

Performance-sensitive code becomes fragile when it relies on scattered micro-optimizations. A more robust approach is:

- keep the design simple and localize performance-critical code,
- measure and profile,
- encapsulate special-case optimizations behind stable interfaces.

This is aligned with the C++ focus on controllable cost and with the Guidelines' efficiency mindset.

7.7 A Practical Review Checklist

Use this checklist during design review and code review:

1. Can a reader tell the intent of each function from its signature?
2. Are ownership and lifetime responsibilities explicit (RAII, no hidden cleanup rules)?
3. Are invalid states unrepresentable (strong types, optional/expected, non-null, size-carrying views)?

4. Is complexity encapsulated rather than spread (one boundary module instead of many call sites)?

5. Are standard library abstractions used where they match the problem?

6. Are tools used to prevent drift (formatting, static analysis, guideline checks)?

Part II

Interfaces (I)

Chapter 8

I.1 – I.26

Scope of this chapter. In the ISO C++ Core Guidelines, an *interface* is a contract between two parts of a program. A good interface makes correct use easy, incorrect use hard, and testing practical. The Guidelines emphasize that interface design is a primary determinant of long-term maintainability, performance, and safety.

8.1 Rule Map (Current Numbering)

The *Interfaces (I)* section currently defines the following rules in the published Core Guidelines: *I.1–I.13* and *I.22–I.26* (with additional rules beyond this range, such as I.27 and I.30). The identifiers *I.14–I.21* do not appear as active rules in the current rule summary and are effectively unused/retired identifiers in the present numbering.

8.2 Shared Design Goals for Interface Rules

- **Make contracts explicit:** dependencies, required inputs, and promised results should be stated, not implied.

- **Prefer strong types over conventions:** types are checked by compilers and tools, reducing ambiguity and improving optimization opportunities.
- **Make errors hard to ignore:** especially for failures to perform a required task.
- **Express non-nullness, ownership, and bounds directly:** to prevent lifetime and bounds mistakes across interfaces.
- **Keep interfaces small and stable:** fewer parameters, fewer surprises, fewer ways to misuse.

8.3 I.1: Make interfaces explicit

Rule

Avoid passing information across an interface through non-local or implicit state. Prefer parameters, return values, and explicit object state.

Why

Unstated assumptions are easy to miss and hard to test. Hidden dependencies increase coupling and reduce reliability.

Example

```
// Bad: behavior depends on hidden global state
bool round_up = false;

int round_value(double d)
{
```

```
    return round_up ? static_cast<int>(std::ceil(d))
                    : static_cast<int>(d);
}

// Better: policy is explicit in the interface
int round_value(double d, bool round_up_policy)
{
    return round_up_policy ? static_cast<int>(std::ceil(d))
                           : static_cast<int>(d);
}
```

8.4 I.2: Avoid non-const global variables

Rule

Avoid non-`const` global (namespace-scope) variables.

Why

They hide dependencies, allow unpredictable changes, complicate testing, and can create data races in concurrent code.

Preferred patterns

- Use `const` / `constexpr` globals for immutable values.
- Prefer explicit state carried by objects and passed by reference to `const`.

8.5 I.3: Avoid singletons

Rule

Avoid singletons.

Why

They are effectively complicated global objects, increasing coupling, hiding dependencies, and complicating testing and lifetime control.

Important note

The Guidelines recognize the usefulness of *function-local static initialization* to avoid initialization-order problems (initialization on first use), while still warning about the broader singleton pattern and its variants.

8.6 I.4: Make interfaces precisely and strongly typed

Rule

Use precise, strong types for parameters and results; avoid weakly typed interfaces such as `void*` and ambiguous "bags of primitives."

Why

Types are the simplest documentation, are compile-time checked, improve readability, and can enable better optimization.

Example

```
// Bad: weak and under-specified
void pass(void* data);

// Better: use a precise type (example only)
void pass(const std::vector<int>& data);
```

8.7 I.5: State preconditions (if any)

Rule

State preconditions of a function (what must be true before the call).

Why

Arguments have meaning that may constrain correct use. Clear preconditions prevent misuse and help tools and tests.

Practical guidance

- Put preconditions in the interface documentation and (when appropriate) in code.
- Treat preconditions as part of the contract between caller and callee.

8.8 I.6: Prefer `Expects()` for expressing preconditions

Rule

Use `Expects(condition)` to express preconditions.

Why

It clarifies that the condition is a precondition and enables tool use.

Example

```
#include <gsl/gsl_assert>   // from a GSL implementation

int area(int h, int w)
{
    Expects(h > 0 && w > 0);
    return h * w;
}
```

Implementation note

`Expects`/`Ensures` are part of the Guideline Support Library (GSL) facilities used by the Core Guidelines.

8.9 I.7: State postconditions

Rule

State postconditions (what the function guarantees on return).

Why

They detect misunderstandings about results and help catch incorrect implementations.

8.10 I.8: Prefer `Ensures()` for expressing postconditions

Rule

Use `Ensures(condition)` to express postconditions.

Why

It clarifies that the condition is a postcondition and enables tool use.

Example

```
#include <gsl/gsl_assert>

void clear_buffer(char* buffer, std::size_t n)
{
    Expects(buffer != nullptr);
    std::memset(buffer, 0, n);
    Ensures(n == 0 || buffer[0] == 0);
}
```

8.11 I.9: If an interface is a template, document its parameters using concepts

Rule

If an interface is a template, document its parameters using concepts.

Why

It makes requirements precise and (in modern C++) compile-time checkable.

Example

```
#include <concepts>
#include <utility>

template <class T>
concept Incrementable = requires(T x) { ++x; };

template <Incrementable T>
T next_value(T x)
{
    ++x;
    return x;
}
```

8.12 I.10: Use exceptions to signal a failure to perform a required task

Rule

Use exceptions to signal failure to perform a required task.

Why

A failure that can be ignored may leave the system in an undefined or unexpected state. Exceptions are designed to be non-ignorable compared to many error-code conventions.

Important clarification

The Guidelines distinguish between:

- **Errors:** the function cannot achieve its advertised purpose (including postconditions) — use exceptions.
- **Expected outcomes/status:** e.g., a remote server may refuse a connection for ordinary reasons — return a result the caller must check.

8.13 I.11: Never transfer ownership by a raw pointer (`T*`) or reference (`T&`)

Rule

Do not transfer ownership through raw pointers or references.

Why

If ownership is unclear, you get leaks or premature destruction. Interfaces must make ownership explicit.

Preferred patterns

- Transfer ownership via `std::unique_ptr<T>` (exclusive ownership).

- Share ownership via `std::shared_ptr<T>` (shared ownership).
- For non-owning observation, use raw `T*` or `T&` (but make non-nullness explicit when required).

8.14 I.12: Declare a pointer that must not be null as `not_null`

Rule

If null is not a valid value, express that in the type (e.g., `gsl::not_null`).

Why

It documents intent, enables static analysis, and eliminates ambiguous "maybe-null" usage across interfaces.

Example

```
#include <gsl/pointers>

void process(gsl::not_null<int*> p)
{
    *p = 42; // no nullptr checks needed to express the contract
}
```

8.15 I.13: Do not pass an array as a single pointer

Rule

Do not pass arrays as a raw pointer without also passing size information in a safe form.

Why

A single pointer loses bounds information, making range errors easy.

Preferred patterns

- Use `std::span<T>` (C++20) for a view over contiguous memory.
- Use `gsl::span<T>` where you adopt GSL facilities (and want guideline-oriented behavior).

Example

```
#include <span>
#include <cstddef>

// Bad: no size information
void fill_bad(int* p)
{
    // unknown bounds -> easy to overflow
}

// Better: size travels with the view
void fill(std::span<int> s)
{
    for (std::size_t i = 0; i < s.size(); ++i) s[i] = 0;
}
```

8.16 I.14 – I.21: Reserved / Not Present in Current Rule Set

The current Interfaces rule summary lists I.1–I.13 and then continues at I.22 (followed by I.23–I.26, and additional rules such as I.27 and I.30). The identifiers I.14–I.21 are not present as active rules in the published summary and are treated here as unused/retired identifiers in the current numbering.

8.17 I.22: Avoid complex initialization of global objects

Rule

Avoid complex initialization of global objects.

Why

Global initialization order is not totally ordered across translation units, making behavior fragile. This is especially difficult in concurrent code, so avoiding globals is often best.

Preferred patterns

- Prefer local objects with clear lifetimes.
- If you need global access, prefer initialization-on-first-use via a function-local static (carefully), or pass dependencies explicitly.

8.18 I.23: Keep the number of function arguments low

Rule

Keep the number of function arguments low.

Why

Many parameters are a sign of a complicated interface, are error-prone to call, and are hard to evolve.

Preferred patterns

- Group cohesive parameters into a strong type (e.g., a configuration struct).
- Prefer objects with invariants over long parameter lists.

8.19 I.24: Avoid adjacent parameters that can be invoked by the same arguments in either order with different meaning

Rule

Avoid adjacent parameters of the same type that can be swapped accidentally.

Why

Such calls compile but can silently do the wrong thing.

Preferred patterns

- Use strong types (distinct wrapper types) for semantically different values.
- Use a parameter object with named members.

Example

```
// Risk: width/height swap compiles
void resize(int width, int height);

// Safer: strong types
struct Width  { int v; };
struct Height { int v; };

void resize(Width w, Height h);
```

8.20 I.25: Prefer empty abstract classes as interfaces to class hierarchies

Rule

Prefer empty abstract classes (no data members) as interfaces.

Why

Interfaces without state are more likely to remain stable than base classes with data, reducing ABI and behavioral coupling.

Example

```
struct IWriter {
    virtual ~IWriter() = default;
    virtual void write(const std::string& s) = 0;
};
```

8.21 I.26: If you want a cross-compiler ABI, use a C-style subset

Rule

If you need a cross-compiler ABI, use a C-style subset.

Why

Different compilers can implement different binary layouts for classes, exception handling, name mangling, and other details. A C-style interface is the most widely portable contract at the binary level.

Practical guidance

- Prefer `extern "C"` functions for ABI boundaries.
- Use POD-like structs with explicit layout rules and avoid C++ features that are ABI-sensitive across compilers (e.g., class hierarchies with virtuals) when cross-compiler ABI is a strict requirement.

8.22 Summary Checklist (Interfaces)

- Make contracts explicit: no hidden global controls (I.1–I.3).
- Use strong typing, not conventions or `void*` (I.4).
- State contracts in code where practical: pre/postconditions (I.5–I.8).
- Specify template requirements using concepts (I.9).
- Use exceptions for true failures-to-perform-required-task (I.10).
- Make ownership/non-nullness/bounds explicit across interfaces (I.11–I.13).
- Avoid fragile global initialization and overly wide signatures (I.22–I.24).
- Prefer stable interface forms (I.25) and C-style boundaries when ABI portability is mandatory (I.26).

Chapter 9

Designing Clear Interfaces

9.1 Purpose of This Chapter

Clear interfaces are the primary mechanism for keeping C++ systems correct, maintainable, and testable over many years. In the ISO C++ Core Guidelines, interface design is treated as a first-class engineering topic: interfaces should express intent, make misuse hard, and enable both compile-time and tool-based checking.

9.2 What an "Interface" Is in C++

An interface is the boundary where one part of the program relies on another:

- function signatures (parameters, return types, overload sets),
- class public APIs (invariants, ownership rules, lifetime rules),
- module/library boundaries (ABI constraints, error policies),
- generic interfaces (template requirements).

The Core Guidelines emphasize making such boundaries explicit and strongly typed (I.1, I.4), stating preconditions and postconditions (I.5–I.8), and documenting template requirements using concepts (I.9).

9.3 Design Goals of a Clear Interface

A clear interface should achieve these goals simultaneously:

1. **Make the contract explicit:** required inputs, allowed states, and promised outcomes are visible.
2. **Make correct use easy:** the most natural call is the correct call.
3. **Make incorrect use hard:** invalid states are unrepresentable or rejected early.
4. **Be checkable:** by the compiler (types, concepts) and by tools (guideline checks).
5. **Be stable under change:** the interface resists misuse as code evolves.

These themes appear repeatedly across the Interfaces rules (I. x) and the Philosophy rules (P. x).

9.4 A Practical Process for Designing an Interface

9.4.1 Step 1: Identify the Meaning, Not the Mechanism

Start by writing the domain contract in plain words:

- What must the caller provide?
- What does success mean?
- What failures are possible?

- Who owns what?
- What are the lifetime/bounds requirements?

Then encode the answers into the signature using types and conventions recommended by the Guidelines.

9.4.2 Step 2: Encode Constraints in Types

Prefer types that carry meaning:

- `enum class` for a closed set of options,
- `std::chrono::duration` for time intervals (unit-safe),
- `std::string_view` for read-only string views,
- `std::span` for contiguous ranges with size,
- `std::optional` / `std::expected` to make absence/failure explicit,
- `gsl::not_null` when null is invalid.

This is the core meaning of "make interfaces precisely and strongly typed" (I.4).

9.5 Parameters: Choosing the Right Shapes

9.5.1 Prefer the Most Specific Acceptable Type (I.4)

Weakly typed signatures (such as `void*`, raw integer flags, or pointer-only arrays) hide intent and allow accidental misuse. Stronger types communicate meaning and enable checking.

9.5.2 Avoid Adjacent Same-Type Parameters (I.24)

If two adjacent parameters have the same type but different meaning, swapping them compiles and silently breaks behavior. Prefer strong types or a parameter object.

```
struct Width  { int v; };
struct Height { int v; };

void resize(Width w, Height h); // cannot be swapped accidentally
```

9.5.3 Keep Parameter Counts Low (I.23)

Many parameters often indicate that the interface is doing too much or that related arguments should be grouped. Group cohesive parameters into a configuration type that can enforce invariants.

```
#include <chrono>
#include <string_view>

struct ConnectOptions {
    std::chrono::milliseconds timeout;
    bool use_tls;
};

bool connect(std::string_view host, int port, const ConnectOptions& opt);
```

9.6 Bounds and Ranges: Avoid “Pointer-Only” Arrays (I.13)

9.6.1 Use `std::span` to Carry Bounds

The Core Guidelines warn against passing arrays as a single pointer because bounds are lost (I.13). A clear interface should represent “a contiguous range” explicitly using `std::span`.

```cpp
#include <span>
#include <cstddef>

int sum(std::span<const int> xs) {
    int s = 0;
    for (std::size_t i = 0; i < xs.size(); ++i) s += xs[i];
    return s;
}
```

9.6.2 Tool Support Reinforces This Choice

LLVM documents a clang-tidy check that flags pointer arithmetic because it is fragile and can lead to invalid pointers, and it explicitly points to `span<T>` as a safer type for array access.

9.7 Nullability and Ownership: Make Lifetime Rules Explicit

9.7.1 Never Transfer Ownership with Raw Pointers (I.11)

The Guidelines state that ownership transfer should not be expressed with raw `T*` or `T&`. Use smart pointers to make ownership explicit.

```cpp
#include <memory>

std::unique_ptr<Widget> make_widget();              // transfers ownership
void use_widget(const Widget& w);                   // non-owning use
void take_widget(std::unique_ptr<Widget> w);        // takes ownership
```

9.7.2 If Null Is Invalid, Say So in the Type (I.12)

When null is not permitted, mark that requirement in the interface (for example with `gsl::not_null`). Microsoft's C++ code analysis documentation explains that asserts do not

expose assumptions through the interface and recommends `gsl::not_null` for values that should never be null.

```
#include <gsl/pointers>

void write_value(gsl::not_null<int*> out) {
    *out = 42;
}
```

9.8 Preconditions and Postconditions (I.5–I.8)

9.8.1 State Preconditions and Postconditions

Clear interfaces include explicit contracts:

- Preconditions: what must be true for correct use.
- Postconditions: what will be true after successful execution.

The Guidelines recommend stating these and, when appropriate, expressing them using `Expects()` and `Ensures()` from the Guideline Support Library (GSL).

```
#include <gsl/gsl_assert>

int area(int h, int w) {
    Expects(h > 0 && w > 0);
    int a = h * w;
    Ensures(a > 0);
    return a;
}
```

9.8.2 Contract Semantics Note

The Core Guidelines maintainers have discussed that `Expects`/`Ensures` represent contract violations that are intended to terminate rather than be handled as ordinary errors. This matters for interface design: preconditions define what callers must satisfy, not recoverable outcomes.

9.9 Error Handling: Make Failure Explicit

9.9.1 Exceptions for Failure to Perform a Required Task (I.10)

The Guidelines recommend using exceptions to signal failure to perform a required task (I.10). The central interface idea is that certain failures must not be silently ignored.

9.9.2 Make "Must-Check" Results Hard to Ignore

For functions where ignoring the result is typically a bug, use `[[nodiscard]]`. The C++ standard attribute encourages compilers to warn when a nodiscard return value is discarded.

```
#include <string_view>

enum class ParseError { invalid };

struct [[nodiscard]] ParseResult {
    int value;
    ParseError error;
};

[[nodiscard]] ParseResult parse_int(std::string_view s);
```

9.10 Read-only Views: Prefer `std::string_view` for Observation

If the callee only needs to *observe* a string, `std::string_view` expresses non-ownership and avoids unnecessary allocation or copying. The standard defines `basic_string_view` as a view over a contiguous sequence of characters.

```
#include <string_view>

bool starts_with_http(std::string_view s) {
    return s.size() >= 4 && s.substr(0, 4) == "http";
}
```

Lifetime rule: because `string_view` does not own data, the interface must ensure that the viewed data outlives the view usage. This is a contract responsibility of the API.

9.11 Generic Interfaces: Use Concepts to Document Requirements (I.9)

If an interface is a template, the Core Guidelines recommend documenting template parameters using concepts (I.9). This makes constraints explicit and produces clearer diagnostics.

```
#include <concepts>
#include <utility>

template <class T>
concept Addable = requires(T a, T b) { a + b; };

template <Addable T>
T add(T a, T b) { return a + b; }
```

9.12 Stability and ABI Boundaries (I.26)

When cross-compiler ABI compatibility is required, the Guidelines recommend using a C-style subset (I.26). C++ ABIs are not universally identical across compilers and platforms, so a stable ABI boundary often uses `extern "C"` functions and simple data layouts.

```
extern "C" {
    struct CPoint { int x; int y; };
    int process_points(const CPoint* p, int n);
}
```

9.13 Tooling: Keep Interfaces Clear Over Time

Clear interfaces are easier to enforce with tools:

- LLVM clang-tidy provides checks aligned with Core Guidelines, including bounds-related checks that flag pointer arithmetic.
- Microsoft documents Core Guidelines checkers and includes diagnostics that recommend using `gsl::not_null` for pointers that should never be null.

9.14 Interface Design Checklist

Use this checklist in API reviews:

1. Does the signature communicate intent without reading the implementation? (I.1, I.4)
2. Are ownership and lifetime rules explicit? (I.11, I.12)
3. Are bounds explicit for ranges? (I.13)
4. Are parameters hard to mix up? Is the parameter list small? (I.23, I.24)

5. Are preconditions/postconditions stated and enforceable? (I.5–I.8)

6. Are failure modes explicit and hard to ignore? (I.10, `[[nodiscard]]`)

7. For templates, are requirements stated with concepts? (I.9)

8. If ABI portability is required, is the boundary C-style? (I.26)

Chapter 10

Separation of Interface and Implementation

10.1 What Separation Means in C++

10.1.1 Interface

In C++, the **interface** is what clients are allowed to rely on:

- public declarations (function signatures, class public members, exported names),
- documented contracts (preconditions, postconditions, error policy),
- visible types and invariants.

10.1.2 Implementation

The **implementation** is everything clients should not depend on:

- private data layout choices,

- helper functions and internal algorithms,
- headers needed only to implement details,
- platform-specific code and optimization tricks.

The C++ Core Guidelines emphasize that good interfaces are explicit and minimize unintended dependencies, because those dependencies make software harder to evolve.

10.2 Why Separation Matters

10.2.1 Stability Under Change

If clients depend only on the interface, you can change the implementation (data structures, algorithms, helper libraries) without forcing widespread rebuilds and without breaking users.

10.2.2 Build Time and Dependency Control

Implementation details often pull in large headers and transitive includes. The Core Guidelines recommend organizing source files to avoid accidental dependency on implementation details that appear in headers, and to avoid fragile reliance on transitive includes.

10.2.3 Binary Compatibility (ABI) Considerations

For libraries that must remain binary-compatible across versions or across compilers, the separation between interface and implementation becomes an ABI design problem:

- exposing data members and inline implementation details can couple clients to your binary layout,
- hiding layout behind an opaque interface improves stability,

- for strict cross-compiler ABI, the Guidelines advise using a C-style subset at the boundary.

10.3 Primary Techniques in C++

10.4 Technique 1: Header/Source Separation (Traditional C++ Model)

10.4.1 The Basic Rule

Place **declarations** in a header (`.h`/`.hpp`) and **definitions** in a source file (`.cpp`):

- clients include the header to compile against the interface,
- the source file includes the header and provides the implementation.

```
// widget.h
#pragma once
#include <string>

class Widget {
public:
    explicit Widget(std::string name);
    void run();
private:
    std::string name_;
};
```

```
// widget.cpp
#include "widget.h"
#include <iostream>
```

```
Widget::Widget(std::string name) : name_(std::move(name)) {}

void Widget::run() {
    std::cout << "Running: " << name_ << "\n";
}
```

10.4.2 Header Hygiene (Avoid Accidental Dependencies)

A client including your header should not have to guess what your header depends on. The Core Guidelines Source File rules emphasize that headers should include what they use, and warn against accidentally relying on implementation details and transitive includes.

10.5 Technique 2: Abstract Interfaces (Pure Abstract Classes)

10.5.1 When You Need Full Runtime Substitution

If you need to swap implementations at runtime, C++ can separate interface from implementation via an **abstract base class** (ABC) with pure virtual functions:

- clients depend only on the interface type,
- implementations live in derived classes compiled separately.

The ISO C++ FAQ explicitly describes using an ABC to separate interface from implementation. The Core Guidelines recommend preferring empty abstract classes as interfaces when you want an interface style (I.25) and explicitly note this as a technique for complete separation (C.122).

```
#include <string>
```

```
struct IWriter {
    virtual ~IWriter() = default;
    virtual void write(std::string_view s) = 0;
};

class FileWriter final : public IWriter {
public:
    explicit FileWriter(const char* path);
    void write(std::string_view s) override;
private:
    // implementation details in the .cpp file
    int fd_ = -1;
};
```

10.5.2 Trade-offs

- Pros: strong separation, runtime interchangeability, reduced coupling.
- Costs: dynamic dispatch, heap allocation patterns are common (but not required), and ABI considerations for virtual interfaces.

10.6 Technique 3: PImpl (Pointer to Implementation)

10.6.1 What PImpl Is

The **PImpl idiom** moves implementation details (including private data members) into a separate `Impl` type and stores only an opaque pointer in the public class. cppreference describes PImpl as a technique that removes implementation details from a class's object representation by placing them into a separate class accessed through an opaque pointer. The Core Guidelines also mention PImpl as a technique related to separating interface and implementation.

10.6.2 A Clear, Modern PImpl Pattern (with `std::unique_ptr`)

```
// widget.h
#pragma once
#include <memory>
#include <string>

class Widget {
public:
    Widget(std::string name);
    ~Widget();                       // out-of-line
    Widget(Widget&&) noexcept;       // out-of-line
    Widget& operator=(Widget&&) noexcept;

    Widget(const Widget&) = delete;
    Widget& operator=(const Widget&) = delete;

    void run();

private:
    struct Impl;                     // forward declaration
    std::unique_ptr<Impl> p_;        // opaque pointer to implementation
};
```

```
// widget.cpp
#include "widget.h"
#include <iostream>

struct Widget::Impl {
    std::string name;
    void run() { std::cout << "Running: " << name << "\n"; }
};

Widget::Widget(std::string name) : p_(std::make_unique<Impl>(Impl{std::move(name)})) {}
```

```
Widget::~Widget() = default;
Widget::Widget(Widget&&) noexcept = default;
Widget& Widget::operator=(Widget&&) noexcept = default;

void Widget::run() { p_->run(); }
```

10.6.3 What PImpl Buys You

- **Stable header surface:** fewer includes in the header, fewer transitive dependencies for clients.
- **Reduced recompilation:** changes to private members often stay in the `.cpp`.
- **Binary stability potential:** the public type size can remain stable even if private details change.

10.6.4 PImpl Costs

- extra indirection for member access,
- typically one heap allocation per object (though custom allocation strategies exist),
- more boilerplate (special member functions often defined out-of-line).

10.7 Technique 4: Modules (C++20 and Later)

10.7.1 Interface Units vs. Implementation Units

C++20 modules provide a language-supported separation:

- **module interface unit:** contains exported declarations (uses `export module`),

- **module implementation unit:** provides definitions and non-exported details.

cppreference defines module interface units as those whose module declaration has `export`, and module implementation units as the others. Microsoft documentation similarly describes module interface units and implementation units and clarifies that a module has exactly one primary module interface unit.

10.7.2 Example Structure

```
// math.ixx  (module interface unit)
export module math;
export int add(int a, int b);
```

```
// math.cppm (module implementation unit)
module math;
int add(int a, int b) { return a + b; }
```

10.7.3 Why Modules Help

- They reduce accidental dependencies caused by textual inclusion.
- They make “what is exported” explicit.
- They can improve build scalability when used correctly (toolchain support matters).

10.8 Special Case: Templates and “Header-Only” Reality

10.8.1 Why Many Templates Must Be Visible

A key reason interface/implementation separation is harder with templates is that template definitions often need to be visible at the point of instantiation. That is why many template

libraries place definitions in headers or in included implementation files. This is a language model constraint, not a style preference.

10.8.2 Practical Approaches

- Keep templates small and delegate heavy work to non-template implementation functions (often called a "non-template core implementation" in guideline discussions about stability and bloat).
- Consider explicit instantiation for selected types when you control the set of instantiations.
- Use modules where available to improve organization and reduce textual inclusion, while still meeting visibility requirements.

10.9 Guideline-Oriented Checklist

A clear interface/implementation separation typically satisfies these points:

1. The header exposes only what clients must use (no private headers leaked).
2. Ownership, nullability, and bounds requirements are explicit in the interface (e.g., `unique_ptr`, `not_null`, `span`).
3. Heavy includes are kept out of headers when possible; headers include what they use and do not rely on transitive includes.
4. If ABI stability matters, avoid exposing layout-sensitive details; consider PImpl or a C-style boundary as appropriate.
5. If runtime interchangeability matters, use an abstract interface (pure abstract base class).

6. For C++20+ codebases, consider modules to make exported interfaces explicit and reduce inclusion-based coupling.

Chapter 11

Minimizing Dependencies

11.1 Why Dependencies Matter

In C++, dependencies are not only a design issue. They are also a build-system and tooling reality because most code is compiled from *translation units* formed by textual inclusion of headers via `#include`. This makes dependency growth directly visible in:

- compile times and incremental build times,
- rebuild cascades after a small change,
- accidental coupling between components,
- fragile “it compiles on my machine” include-order bugs.

The ISO C++ Core Guidelines explicitly treat dependency control as a key engineering topic, especially in the **SF (Source files)** rules and in the general Interface philosophy.

11.2 Core Goals of Dependency Minimization

A dependency-minimized C++ design aims to:

1. **Expose only what clients must know** (stable interface).
2. **Hide implementation details** (allow change without widespread rebuilds).
3. **Make each header self-sufficient** (no reliance on transitive includes).
4. **Prevent cyclic dependencies** (simplify reasoning and build graphs).
5. **Support tool-based enforcement** (static analysis and include-graph tooling).

These goals align with the Core Guidelines emphasis on explicit interfaces and on source-file rules that reduce accidental coupling.

11.3 The Fundamental Rule: Headers Define Your Dependency Surface

11.3.1 A Header Is a Contract

If a type or function appears in a public header, it becomes part of the dependency surface:

- clients must parse it,
- clients may indirectly rely on it,
- any change can force rebuilds.

11.3.2 A Header Must Be Safe to Include Multiple Times

The Core Guidelines recommend include guards in headers to avoid multiple inclusion, and they provide guidance on naming guards to avoid collisions. Some implementations offer `#pragma once`, but the Guidelines note it is not standard and not portable, and recommend writing ISO C++.

```
// file: library_widget.h
#ifndef MYLIB_LIBRARY_WIDGET_H
#define MYLIB_LIBRARY_WIDGET_H

// declarations ...

#endif // MYLIB_LIBRARY_WIDGET_H
```

11.4 Practical Techniques to Minimize Dependencies

11.5 Technique 1: Prefer Forward Declarations in Headers When Possible

11.5.1 Idea

If a header only needs to *mention* a type (as a pointer, reference, or incomplete type member such as `std::unique_ptr<T>`), you can often forward-declare the type instead of including its full definition.

11.5.2 Example

```
// widget.h
#pragma once
```

```
#include <memory>

class Engine; // forward declaration

class Widget {
public:
    explicit Widget(std::unique_ptr<Engine> e);
    void run();
private:
    std::unique_ptr<Engine> engine_; // Engine can be incomplete here
};
```

```
// widget.cpp
#include "widget.h"
#include "engine.h" // full definition needed here

Widget::Widget(std::unique_ptr<Engine> e) : engine_(std::move(e)) {}
void Widget::run() { engine_->step(); }
```

11.5.3 Limits

Forward declarations are not always sufficient:

- you need a complete type for by-value parameters/returns,
- you need a complete type for members stored by value,
- you need the definition for inline functions that use the type.

11.6 Technique 2: Keep Headers “Clean” and Self-Contained

11.6.1 Include What You Use

A header should include the headers that provide the declarations it uses. Do not assume a transitive include will always be present. The Core Guidelines source-file rules explicitly address header organization to avoid fragile reliance on inclusion order.

11.6.2 Include Your Own Header First

A common, effective practice is that each `.cpp` includes its matching header first. This quickly reveals missing includes in the header and prevents hidden reliance on other includes. The Core Guidelines emphasize including `.h` files before other declarations in a file (SF.4).

```
// widget.cpp
#include "widget.h"   // include self first
#include <utility>
#include "engine.h"
```

11.7 Technique 3: Move Implementation Details Out of Headers

11.7.1 Avoid Heavy Includes in Public Headers

If a public header includes large third-party headers or complex STL-heavy internals, every client pays for parsing them.

11.7.2 Move Definitions to `.cpp`

The Core Guidelines recommend that `.h` files not contain object definitions or non-inline function definitions, supporting a clean separation between interface and implementation.

11.8 Technique 4: Use PImpl to Hide Private Data and Reduce Rebuilds

11.8.1 Why PImpl Helps

PImpl (pointer to implementation) moves private members and implementation-only headers into the .cpp, reducing what the public header must include.

11.8.2 Modern PImpl Example

```
// widget.h
#pragma once
#include <memory>
#include <string>

class Widget {
public:
    explicit Widget(std::string name);
    ~Widget();                       // out-of-line (Impl incomplete here)
    Widget(Widget&&) noexcept;
    Widget& operator=(Widget&&) noexcept;

    Widget(const Widget&) = delete;
    Widget& operator=(const Widget&) = delete;

    void run();

private:
    struct Impl;
    std::unique_ptr<Impl> p_;
};
```

```cpp
// widget.cpp
#include "widget.h"
#include <iostream>

struct Widget::Impl {
    std::string name;
    void run() { std::cout << "Running: " << name << "\n"; }
};

Widget::Widget(std::string name)
  : p_(std::make_unique<Impl>(Impl{std::move(name)})) {}

Widget::~Widget() = default;
Widget::Widget(Widget&&) noexcept = default;
Widget& Widget::operator=(Widget&&) noexcept = default;

void Widget::run() { p_->run(); }
```

11.8.3 Trade-offs

PImpl typically introduces:

- one extra indirection on member access,
- usually one heap allocation per object (unless specialized).

This is a conscious engineering trade: less compile-time coupling and more ABI stability potential, at a runtime cost that is often acceptable for boundary objects.

11.9 Technique 5: Prefer Non-Owning Views Over Owning Types for Read-Only Parameters

Using view types can reduce dependencies and clarify intent:

- `std::string_view` can avoid including heavy string-processing headers in some interfaces and prevents accidental ownership transfer.
- `std::span` represents "contiguous range" without exposing container type, and it keeps size with the pointer.

The standard defines module-related restrictions on header inclusion in module units (headers can only be included in the global module fragment). This reinforces a clean boundary between header inclusion and module interfaces.

```
#include <string_view>
#include <span>

void log_line(std::string_view s);
int sum(std::span<const int> xs);
```

11.10 Technique 6: Break Cycles by Refactoring the Interface

Cyclic header dependencies usually mean one of the following:

- two types know too much about each other,
- responsibilities are mixed,
- a common concept is missing.

Common refactor strategies:

1. **Extract a smaller common interface/header** that both depend on.
2. **Depend on an abstract interface** (pure abstract base class) rather than a concrete type.
3. **Use dependency inversion:** high-level code depends on an interface, and implementations depend on that interface.

The Core Guidelines recommend empty abstract classes as interfaces (I.25) and discuss abstract interfaces as a separation tool.

11.11 Technique 7: Use C++20 Modules to Reduce Inclusion-Based Coupling

11.11.1 What Modules Change

C++20 modules provide language support for separating *what is exported* from *how it is implemented*. A module has:

- **module interface units** (with `export module`),
- **module implementation units** (with `module`).

Microsoft documents that a module has exactly one primary module interface unit, and describes interface and implementation units explicitly. cppreference documents module units and the global module fragment used when headers must still be included.

11.11.2 Minimal Example

```
// math.ixx (module interface unit)
export module math;
export int add(int a, int b);
```

```cpp
// math.cpp (module implementation unit)
module math;
int add(int a, int b) { return a + b; }
```

11.11.3 Practical Note on Toolchains

Module builds depend heavily on compiler and build-system support. Clang documentation for Standard C++ modules describes the rule that a module should have one and only one primary module interface unit and explains implementation units.

11.12 Tooling Support for Dependency Control

11.12.1 Guideline-Driven Static Analysis

The Core Guidelines explicitly point to tool support (such as `clang-tidy` rule sets and Microsoft code analysis checkers) as part of sustainable guideline adoption.

11.12.2 Include-Graph and "Remove Unused Headers" Tools

Modern IDEs and tooling can help remove unused includes and optimize include graphs. For example, Microsoft's C++ team describes features for removing unused headers and optimizing include graphs.

11.13 Dependency-Minimization Checklist

Use this checklist during API and build reviews:

1. Does the public header expose only what clients must know?
2. Can you replace includes with forward declarations in headers where legal?

3. Are non-inline definitions kept out of headers (unless required)?
4. Does each `.cpp` include its own header first (to reveal missing includes)?
5. Are cycles eliminated by extracting a common interface or using abstraction?
6. Where rebuild cost is high, does PImpl reduce the public dependency surface?
7. If modules are available, is the exported surface explicit and minimal?
8. Are unused includes removed and include graphs monitored with tools?

Chapter 12

Resource-safe Interfaces

12.1 What "Resource-safe" Means at an Interface Boundary

A **resource** is anything that must be acquired and later released correctly: dynamically allocated memory, file descriptors, sockets, mutex locks, threads, database handles, GPU objects, and similar limited or externally managed entities.
A **resource-safe interface** is an interface that makes it easy to use resources correctly and difficult to misuse them. In the ISO C++ Core Guidelines, this is achieved by:

- managing resources automatically using **RAII** (**R.1**),
- treating raw pointers and references as **non-owning** (**R.3**, **R.4**),
- **never transferring ownership** through raw pointers/references (**I.11**),
- expressing **non-nullness** in types when null is invalid (**I.12**),
- expressing **bounds** for contiguous sequences (**I.13**).

12.2 Why Resource Safety Starts With Interfaces

Resource leaks and lifetime bugs rarely come from the hardest algorithmic code. They come from unclear contracts:

- Who owns this resource?
- Who releases it?
- Can the handle be null?
- How long does the referred object live?
- Where are the bounds?

The Core Guidelines explicitly treat ownership, nullability, and bounds as interface-level concerns (Interfaces: **I.11–I.13**; Resources: **R.1–R.4**).

12.3 The Core Rule: RAII Handles Own Resources (R.1)

12.3.1 Rule

R.1: Manage resources automatically using resource handles and RAII.

12.3.2 Meaning

RAII binds resource acquisition to object initialization and resource release to object destruction. cppreference describes RAII as a technique that binds the lifecycle of a resource to the lifetime of an object.

12.3.3 Interface Consequence

A resource-safe interface prefers returning and passing **resource handles** (objects that manage lifetime) instead of raw handles that require the caller to remember cleanup rules.

12.3.4 Example: Memory as a Resource

```
// bad: caller must remember to delete
int* make_array_bad(std::size_t n) {
    return new int[n];
}
```

```
#include <memory>

// good: ownership is explicit and cleanup is automatic
std::unique_ptr<int[]> make_array(std::size_t n) {
    return std::make_unique<int[]>(n);
}
```

12.4 Ownership in Interfaces: Never Transfer Ownership via Raw Pointers (I.11)

12.4.1 Rule

I.11: Never transfer ownership by a raw pointer (`T*`) or reference (`T&`).

12.4.2 Reason

If it is unclear who owns an object, leaks or premature destruction occur. The Guidelines explicitly recommend transferring ownership using smart pointers such as `std::unique_ptr` (exclusive) or `std::shared_ptr` (shared) when reference semantics are required.

12.4.3 Recommended Ownership Shapes

- **Return by value** when you can (often simplest and most efficient).
- **Transfer ownership** with `std::unique_ptr<T>` (move-only).
- **Share ownership** with `std::shared_ptr<T>` only when necessary.
- Use raw pointers/references **only as non-owning observations** (**R.3**, **R.4**).

12.4.4 Example: Transferring Ownership

cppreference notes that `std::unique_ptr` transfers ownership by move (it cannot be copied). Microsoft documentation also explains that `unique_ptr` can only be moved and that moving transfers ownership.

```
#include <memory>

struct Job { /*...*/ };

void enqueue(std::unique_ptr<Job> j); // takes ownership

int main() {
    auto j = std::make_unique<Job>();
    enqueue(std::move(j));            // ownership moved into enqueue
}
```

12.4.5 Annotation for Legacy Interfaces

The Core Guidelines mention that if older code cannot be changed (for ABI or other reasons), owning raw pointers may be marked with `owner` from the Guidelines Support Library as an annotation.

12.5 Non-nullness as a Contract: Use `not_null` (I.12)

12.5.1 Rule

I.12: Declare a pointer that must not be null as `not_null`.

12.5.2 Why It Improves Resource Safety

Null checks scattered inside implementations are weaker than an interface-level contract. Microsoft's C++ code analysis documentation explains that `assert` statements do not expose assumptions through the interface and recommends using `gsl::not_null` to mark values that should never be null.

12.5.3 Example

```
#include <gsl/pointers>

struct Buffer { /*...*/ };

void fill(Buffer* b); // unclear: can b be null?

void fill2(gsl::not_null<Buffer*> b) { // clear: null is invalid
    // no need for defensive null checks to express the contract
}
```

12.6 Bounds Safety for Resource-like Views: Use `span` (I.13)

12.6.1 Rule

I.13: Do not pass an array as a single pointer.

12.6.2 Reason

A pointer alone loses size information. This is a common source of out-of-bounds memory access, which is a resource safety failure (memory is a resource).

12.6.3 Preferred Interface Forms

- `std::span<T>` to represent a view over a contiguous sequence with bounds.
- Alternatively `gsl::span<T>` in codebases that adopt the GSL facilities.

12.6.4 Example

```
// bad: no size information
void zero_bad(int* p) {
    // unknown bounds; caller must remember a hidden contract
}
```

```
#include <span>
#include <cstddef>

// good: bounds are part of the interface
void zero(std::span<int> s) {
    for (std::size_t i = 0; i < s.size(); ++i) s[i] = 0;
}
```

12.6.5 Tool Support

Microsoft documents that warning C26481 supports Core Guidelines rule I.13 and recommends replacing pointer arithmetic with safer buffers such as `span<T>` or `vector<T>`.

12.7 Resource-safe Parameter Passing Patterns

This section summarizes interface patterns that keep ownership and lifetime rules explicit.

12.7.1 Pattern A: Borrowing (Non-owning Observation)

Use `T&` or `T*` to denote non-ownership (**R.3**, **R.4**). If null is invalid, use `not_null<T*>`.

```
struct Engine { void step(); };

void tick(Engine& e) { e.step(); }              // cannot be null
void tick2(Engine* e) { if (e) e->step(); }     // nullable observer
```

12.7.2 Pattern B: Taking Ownership

Use `std::unique_ptr<T>` by value to take ownership, or return `std::unique_ptr<T>` to give ownership.

```
#include <memory>

struct Widget { /*...*/ };

std::unique_ptr<Widget> make_widget();
void set_widget(std::unique_ptr<Widget> w); // takes ownership
```

12.7.3 Pattern C: Shared Ownership (Use Sparingly)

Use `std::shared_ptr<T>` when multiple owners are required. The Guidelines explicitly note that shared ownership is an alternative, but it is generally more complex than exclusive ownership.

```cpp
#include <memory>

struct Cache { /*...*/ };

void use_cache(std::shared_ptr<Cache> c); // shares ownership
```

12.8 RAII Beyond Memory: Locks, Files, and Other Handles

cppreference lists common RAII-managed resources such as file handles, sockets, and locked mutexes. The interface lesson is the same: represent such resources with handle objects that release in their destructors.

12.8.1 Example: Lock as a Resource

```cpp
#include <mutex>

std::mutex m;

void critical() {
    std::lock_guard<std::mutex> lock(m); // acquires on construction, releases on
    ↪  destruction
    // critical section
}
```

12.8.2 Example: File as a Resource

```cpp
#include <fstream>
#include <string>

std::string read_first_line(const char* path) {
    std::ifstream in(path);   // RAII: closes file automatically
```

```
    std::string line;
    std::getline(in, line);
    return line;
}
```

12.9 Static Analysis Reinforcement for Resource-safe Interfaces

12.9.1 Ownership and Raw Pointers

Microsoft warning C26400 explains that it helps enforce I.11 (and relates it to R.3: raw pointers are non-owning) by warning when the result of `operator new` is stored in a raw pointer.

12.9.2 Non-nullness

Microsoft warning C26429 recommends `gsl::not_null` to mark values that should never be null.

12.9.3 Bounds

Microsoft warning C26481 supports I.13 and recommends safer buffer types such as `span<T>` or `vector<T>`.

12.10 Checklist: Designing a Resource-safe Interface

Use this checklist when reviewing an API boundary:

1. **Is ownership explicit?** No raw `T*` or `T&` transfers ownership (I.11).

2. **Are raw pointers non-owning observers?** Treat `T*` as non-owning (R.3) unless annotated for legacy constraints.

3. **Is nullability explicit?** If null is invalid, use `not_null` (I.12).

4. **Are bounds explicit?** Use `span` for contiguous sequences (I.13).

5. **Is RAII used for all resources?** Prefer handle objects that release in destructors (R.1).

6. **Are tool checks aligned with the interface?** Enable guideline checks to detect drift early.

Chapter 13

Interfaces That Express Ownership and Lifetime

13.1 Purpose of This Chapter

Many of the most expensive C++ bugs are not algorithmic mistakes. They are **ownership** and **lifetime** mistakes:

- who owns a resource and must release it,
- who only observes it,
- how long a pointed-to object must stay alive,
- whether a pointer may be null,
- whether a view has valid bounds.

The ISO C++ Core Guidelines treat these as *interface-level contracts* (not hidden implementation details), especially in the Interfaces rules (I. 11–I. 13) and Resource

Management rules (R. 1–R. 4).

13.2 Core Vocabulary

13.2.1 Ownership

Owning means an entity is responsible for releasing a resource (for example, deleting dynamically allocated memory, closing a file descriptor, releasing a mutex). In the Core Guidelines, transferring ownership through raw pointers or references is forbidden (I.11).

13.2.2 Lifetime

Lifetime describes how long an object remains valid. Non-owning pointers and references are safe only if the referred object outlives the use.

13.2.3 Observation vs. Ownership in Raw Types

The Core Guidelines explicitly state:

- **R.3:** a raw pointer (`T*`) is **non-owning**,
- **R.4:** a raw reference (`T&`) is **non-owning**.

Therefore, if your interface uses a raw pointer or reference, the default meaning should be *borrowing/observing*, not owning.

13.3 The Primary Rules for Ownership/Lifetime Interfaces

13.3.1 I.11: Never transfer ownership by a raw pointer or reference

Rule: do not transfer ownership via `T*` or `T&`.

Meaning: if your API gives or takes ownership, the ownership must be explicit in the type (typically using standard resource handles such as smart pointers, or returning by value).

13.3.2 I.12: Declare a pointer that must not be null as `not_null`

Rule: if null is not a valid value, express that in the type (e.g., `gsl::not_null`).
Microsoft's C++ code analysis documentation explains why relying on `assert` for nullness is weaker: assertions do not expose the assumption in the function signature, and they can drift from the actual code over time. The recommended solution is to use `gsl::not_null` as part of the interface.

13.3.3 R.1, R.3, R.4: RAII handles own; raw pointers/references observe

R.1 requires automatic resource management using RAII. **R.3** and **R.4** define the interpretation of raw pointers and references as non-owning.
These three rules together imply a simple design principle:

> *Owning should be visible (RAII handle types). Borrowing can be raw (pointers/references), but must be clear about nullability and lifetime constraints.*

13.4 Interface Shapes That Make Ownership Explicit

This section gives practical, guideline-aligned interface shapes for the most common ownership cases.

13.4.1 Case A: The callee does not store the argument (temporary borrow)

Use: `T&` when null is not allowed; `T*` when "no object" is a valid meaning. This matches **R.3** and **R.4**.

```
struct Engine { void step(); };

void tick(Engine& e) {            // non-null borrow
    e.step();
}

void tick_if(Engine* e) {         // nullable borrow
    if (e) e->step();
}
```

13.4.2 Case B: The callee requires non-null but still does not own

Use: `gsl::not_null<T*>` to express non-nullness at the boundary (I.12).
Microsoft GSL documentation specifies that `gsl::not_null<T>` restricts a pointer (or smart pointer) to non-null values and has no size overhead compared to `T`.

```
#include <gsl/pointers>

struct Buffer { void write(int); };

void write_value(gsl::not_null<Buffer*> b) {  // borrow + non-null
    b->write(42);
}
```

13.4.3 Case C: The callee takes ownership (exclusive ownership)

Use: `std::unique_ptr<T>` by value to take ownership, and return `std::unique_ptr<T>` to transfer ownership out.
cppreference states that only a non-const `unique_ptr` can transfer ownership to another `unique_ptr`, and it highlights its use for lifetime management and exception safety.

```
#include <memory>
```

```
struct Job { /*...*/ };

void enqueue(std::unique_ptr<Job> j);   // takes ownership

std::unique_ptr<Job> make_job();        // transfers ownership to caller

int main() {
    auto j = make_job();
    enqueue(std::move(j));              // ownership moved to enqueue
}
```

13.4.4 Case D: The callee shares ownership (shared lifetime)

Use: `std::shared_ptr<T>` when multiple owners are required.

cppreference defines `std::shared_ptr` as a smart pointer that retains shared ownership and states that the object is destroyed when the last owning `shared_ptr` is gone (destroyed or reset/assigned).

```
#include <memory>

struct Cache { /*...*/ };

void use_cache(std::shared_ptr<Cache> c);   // shares ownership
```

13.4.5 Case E: Observe a shared object without extending its lifetime

Use: `std::weak_ptr<T>`.

cppreference defines `std::weak_ptr` as a smart pointer holding a non-owning "weak" reference to an object managed by `std::shared_ptr`, and notes it must be converted to `std::shared_ptr` to access the object.

```
#include <memory>

struct Session { void send(); };

void try_send(std::weak_ptr<Session> w) {     // observes without owning
    if (auto s = w.lock()) {                 // obtains shared ownership if still alive
        s->send();
    }
}
```

13.5 Expressing Lifetime and Bounds for Views

Ownership types solve *who releases*. But many bugs come from *views* that outlive their data or run past bounds.

13.5.1 Non-owning string views

A string view expresses borrowing rather than owning. The key interface rule is:

> *If you accept a view, your contract must ensure the referenced data remains alive during use.*

This is a lifetime contract, not an implementation detail.

13.5.2 Bounds-carrying contiguous views

For arrays/buffers, the Interfaces rules recommend avoiding "pointer-only" array parameters (I.13). Using a bounds-carrying view (such as `span`) makes the interface safer and clearer.

```
#include <span>
#include <cstddef>
```

```
void zero(std::span<int> s) {
    for (std::size_t i = 0; i < s.size(); ++i) s[i] = 0;
}
```

13.6 GSL Annotations for Legacy Interfaces

Sometimes you cannot change an interface immediately (for example, ABI constraints or large legacy code). In such cases, the GSL provides annotations:

- `gsl::owner<T>` as an *alias/annotation* for a raw pointer to indicate ownership intent,
- `gsl::not_null<T>` to indicate non-nullness.

These are primarily to help humans and tools, not to magically fix lifetime rules.

13.7 Tooling Reinforcement: Making Contracts Enforceable

13.7.1 MSVC Core Guidelines checks (ownership)

Microsoft warning C26400 states that it helps enforce **I.11** and **R.3** by warning when the result of `operator new` is stored in a raw pointer (ownership is being hidden).

13.7.2 MSVC Core Guidelines checks (non-nullness)

Microsoft warning C26429 recommends marking pointers that should never be null with `gsl::not_null`, explaining that assertions do not expose the assumption in the interface.

13.8 Design Checklist: Ownership and Lifetime at the Boundary

Use this checklist when reviewing an API:

1. **Is ownership explicit?** No ownership transfer through `T*` or `T&` (I.11).
2. **Are raw pointers/references used only for borrowing?** (**R.3**, **R.4**).
3. **Is nullability expressed in the type?** Use `not_null` when null is invalid (I.12).
4. **Are bounds explicit for contiguous sequences?** Prefer a view with size (I.13).
5. **Is the lifetime contract stated?** Especially for views and observers (including `weak_ptr`).
6. **Is tooling enabled to prevent drift?** Ownership and nullness checks catch regressions early.

Part III

Functions (F)

Chapter 14

F.1 – F.60: Function Guidelines (Complete Numbered Set)

14.1 Scope and edition note

This chapter enumerates **all guideline numbers** from **F.1** through **F.60** exactly as numbered in the current published C++ Core Guidelines revision dated **Jul 8, 2025**. In that revision, some numbers are **intentionally not assigned** (gaps in numbering). Those numbers are included here explicitly and marked as *not assigned in this revision* so that no numbers appear missing.

14.2 F.def: Function definition rules

14.2.1 F.1: Package meaningful operations as carefully named functions

Rule: Extract coherent actions into well-named functions. **Why:** Naming clarifies intent, improves reuse, reduces tangling of unrelated concerns, and limits scope of variables.

```
#include <istream>
```

```
#include <ostream>
#include <stdexcept>

int read_int(std::istream& is) {
    int x{};
    if (!(is >> x)) throw std::runtime_error("input error");
    return x;
}

void print_int(std::ostream& os, int x) {
    os << x << '\n';
}
```

14.2.2 F.2: A function should perform a single logical operation

Rule: Keep one primary responsibility per function. **Why:** Easier to reason about, test, reuse, and maintain.

```
void process_record(/*...*/) {   // one clear purpose
    // validate
    // transform
    // persist
}
```

14.2.3 F.3: Keep functions short and simple

Rule: Prefer small functions with simple control flow. **Why:** Complexity hides defects; smaller functions are easier to review, test, and optimize.

14.2.4 F.4: If a function might have to be evaluated at compile time, declare it `constexpr`

Rule: Use `constexpr` when compile-time evaluation is part of the intended contract (when feasible). **Why:** Enables compile-time computation and stronger guarantees; still usable at run time.

```
constexpr int square(int x) { return x * x; }
static_assert(square(9) == 81);
```

14.2.5 F.5: If a function is very small and time-critical, declare it `inline`

Rule: Use `inline` only as an interface/ODR tool and (rarely) as a performance hint for tiny hot functions. **Why:** Overuse harms readability; compilers often decide inlining regardless.

```
inline int add1(int x) { return x + 1; }
```

14.2.6 F.6: If your function must not throw, declare it `noexcept`

Rule: Mark functions `noexcept` when the contract is non-throwing. **Why:** Communicates intent, enables optimizations, and affects exception-safety behavior (including move operations).

```
void release_handle(int h) noexcept;
```

14.2.7 F.7: For general use, take `T*` or `T&` arguments rather than smart pointers

Rule: In most APIs, accept a raw pointer/reference to use an object, not `unique_ptr/shared_ptr`. **Why:** Passing smart pointers in parameters often (unintentionally) communicates ownership transfer/sharing. Prefer expressing ownership at the call site or in the returned value.

```
#include <string>

struct User { std::string name; };

void greet(const User& u);   // uses u, does not own u
void greet(const User* u);   // allows "no object" via nullptr when that makes sense
```

14.2.8 F.8: Prefer pure functions

Rule: Prefer functions with no side effects and results depending only on inputs. **Why:** Easier reasoning/testing; often easier to optimize and parallelize.

```
template<class T>
T clamp(T v, T lo, T hi) {
    return (v < lo) ? lo : (hi < v) ? hi : v;
}
```

14.2.9 F.9: Unused parameters should be unnamed

Rule: If a parameter is intentionally unused, omit its name. **Why:** Improves readability and avoids unused-parameter warnings. If it is conditionally unused, use `[[maybe_unused]]`.

```
struct Hint {};

void f(int, Hint) { /* Hint currently unused */ }

template<class T>
void g(const T& x, [[maybe_unused]] Hint h) {
    if constexpr (sizeof(T) > 64) {
        (void)h; // used only in some builds/instantiations
    }
    (void)x;
}
```

14.2.10 F.10: If an operation can be reused, give it a name

Rule: Prefer a named function (or named lambda) over repeating complex logic. **Why:** A name documents intent and reduces duplication.

```
auto less_by_rank_then_value = [](const auto& a, const auto& b) {
    return (a.rank() < b.rank()) || (a.rank() == b.rank() && a.value() < b.value());
};
```

14.2.11 F.11: Use an unnamed lambda if you need a simple function object in one place only

Rule: Use a short inline lambda at the call site when it is truly local and simple. **Why:** Keeps the customization close to its use; avoid creating unnecessary named utilities for one-off trivial code.

```
#include <algorithm>
#include <vector>

int count_positive(const std::vector<int>& v) {
    return (int)std::count_if(v.begin(), v.end(), [](int x) { return x > 0; });
}
```

14.2.12 F.12: Not assigned in the Jul 8, 2025 revision

Status: No guideline with number F.12 exists in the published numbering for this revision.

14.2.13 F.13: Not assigned in the Jul 8, 2025 revision

Status: No guideline with number F.13 exists in the published numbering for this revision.

14.2.14 F.14: Not assigned in the Jul 8, 2025 revision

Status: No guideline with number F.14 exists in the published numbering for this revision.

14.3 F.call: Parameter passing expression rules

14.3.1 F.15: Prefer simple and conventional ways of passing information

Rule: Use ordinary value/reference/pointer parameter passing; avoid clever encodings. **Why:** Conventions reduce surprises and improve readability and tool support.

14.3.2 F.16: For "in" parameters, pass cheaply-copied types by value and others by reference to const

Rule: Use pass-by-value for small/cheap types; otherwise pass `const T&`. **Why:** Avoid unnecessary copies for large objects; keep calling code simple and clear.

```
#include <string>

void set_id(int id);                    // cheap by value
void set_name(const std::string& s);    // expensive by const reference
```

14.3.3 F.17: For "in-out" parameters, pass by reference to non-const

Rule: If the function modifies the argument, use `T&`. **Why:** Makes mutation explicit in the signature.

```
void normalize(double& x); // modifies x
```

14.3.4 F.18: For "will-move-from" parameters, pass by `X&&` and `std::move` the parameter

Rule: If the function will consume the argument, accept `T&&` (or `T` by value as an alternative design) and move from it. **Why:** Communicates consumption and enables efficient transfers.

```
#include <string>
#include <utility>

void store(std::string&& s) {
    std::string local = std::move(s); // consumes s
    (void)local;
}
```

14.3.5 F.19: For "forward" parameters, pass by `TP&&` and only `std::forward` the parameter

Rule: In forwarding functions, use forwarding references and forward exactly once. **Why:** Preserves value category correctly and avoids accidental copies/moves.

```
#include <utility>

template<class F, class... Args>
decltype(auto) call(F&& f, Args&&... args) {
    return std::forward<F>(f)(std::forward<Args>(args)...);
}
```

14.3.6 F.20: For "out" output values, prefer return values to output parameters

Rule: Prefer returning the result rather than writing to an output parameter. **Why:** Clearer interfaces, supports chaining, and works naturally with move/NRVO; reduces aliasing hazards.

```
#include <string>

std::string read_line(); // preferred
```

14.3.7 F.21: To return multiple “out” values, prefer returning a struct

Rule: If returning several values, return a named aggregate with meaningful member names (prefer over raw `pair`/`tuple` when feasible). **Why:** Improves readability and self-documentation.

```
#include <string>

struct ParseResult {
    int value{};
    bool ok{};
    std::string error;
};

ParseResult parse_int(const std::string& s);
```

14.4 F.call: Parameter passing semantic rules

14.4.1 F.22: Use `T*` or `owner<T*>` to designate a single object

Rule: Use `T*` to refer to a single object (possibly null) when you do not own it; use `owner<T*>` (a guideline annotation) only to mark ownership when raw pointers must be used. **Why:** Raw pointers are otherwise overloaded with too many meanings; the rule aims to make intent explicit.

```
void inspect(const int* p);     // points to one int (may be nullptr)
```

14.4.2 F.23: Use a `not_null<T>` to indicate that "null" is not a valid value

Rule: If a pointer must not be null, express that requirement with a non-null wrapper type.
Why: Makes preconditions explicit and enables checking/tools.

```
template<class T>
class not_null {
public:
    explicit not_null(T p) : p_(p) { if (!p_) throw 0; }
    T get() const noexcept { return p_; }
private:
    T p_;
};

void use_ptr(not_null<int*> p) { *p.get() = 42; }
```

14.4.3 F.24: Use a `span<T>` (or similar) to designate a half-open sequence

Rule: Pass arrays/ranges as a view type that carries pointer + size (half-open range). **Why:** Avoids separate pointer/length parameters, reduces bounds errors, improves clarity.

```
#include <cstddef>

template<class T>
struct span {
    T* p{};
    std::size_t n{};
    T* begin() const { return p; }
    T* end()   const { return p + n; }
};

int sum(span<const int> s) {
    int r = 0;
```

```
    for (auto x : s) r += x;
    return r;
}
```

14.4.4 F.25: Use a `zstring` (or `not_null<zstring>`) to designate a C-style string

Rule: If using C strings, make the zero-terminated requirement explicit in the type. **Why:** Differentiates from arbitrary `char*` that might point to non-terminated buffers.

```
using zstring = const char*; // guideline-style alias for zero-terminated string

std::size_t len(zstring s); // expects '\0'-terminated
```

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

Rule: Express ownership transfer with `unique_ptr`. **Why:** Makes single ownership explicit and prevents leaks/double deletes.

```
#include <memory>

struct Widget {};

std::unique_ptr<Widget> make_widget();

void take_ownership(std::unique_ptr<Widget> w); // consumes ownership
```

14.4.6 F.27: Use a `shared_ptr<T>` to share ownership

Rule: Use `shared_ptr` only when shared ownership is genuinely required. **Why:** Shared ownership is heavier and more complex; make it explicit where it exists.

```
#include <memory>

struct Node {};

void add_ref(std::shared_ptr<Node> p); // shared ownership
```

14.4.7 F.28: Not assigned in the Jul 8, 2025 revision

Status: No guideline with number F.28 exists in the published numbering for this revision.

14.4.8 F.29: Not assigned in the Jul 8, 2025 revision

Status: No guideline with number F.29 exists in the published numbering for this revision.

14.4.9 F.30: Not assigned in the Jul 8, 2025 revision

Status: No guideline with number F.30 exists in the published numbering for this revision.

14.4.10 F.31: Not assigned in the Jul 8, 2025 revision

Status: No guideline with number F.31 exists in the published numbering for this revision.

14.4.11 F.32: Not assigned in the Jul 8, 2025 revision

Status: No guideline with number F.32 exists in the published numbering for this revision.

14.4.12 F.33: Not assigned in the Jul 8, 2025 revision

Status: No guideline with number F.33 exists in the published numbering for this revision.

14.4.13 F.34: Not assigned in the Jul 8, 2025 revision

Status: No guideline with number F.34 exists in the published numbering for this revision.

14.4.14 F.35: Not assigned in the Jul 8, 2025 revision

Status: No guideline with number F.35 exists in the published numbering for this revision.

14.4.15 F.36: Not assigned in the Jul 8, 2025 revision

Status: No guideline with number F.36 exists in the published numbering for this revision.

14.4.16 F.37: Not assigned in the Jul 8, 2025 revision

Status: No guideline with number F.37 exists in the published numbering for this revision.

14.4.17 F.38: Not assigned in the Jul 8, 2025 revision

Status: No guideline with number F.38 exists in the published numbering for this revision.

14.4.18 F.39: Not assigned in the Jul 8, 2025 revision

Status: No guideline with number F.39 exists in the published numbering for this revision.

14.4.19 F.40: Not assigned in the Jul 8, 2025 revision

Status: No guideline with number F.40 exists in the published numbering for this revision.

14.4.20 F.41: Not assigned in the Jul 8, 2025 revision

Status: No guideline with number F.41 exists in the published numbering for this revision.

14.5 F.ret: Value return semantic rules

14.5.1 F.42: Return a `T*` to indicate a position (only)

Rule: Return a pointer when the result is best expressed as a position in a sequence (or "not found" via null). **Why:** A pointer naturally represents a location and can express "no position".

```
int* find_first_positive(int* b, int* e) {
    for (auto p = b; p != e; ++p) if (*p > 0) return p;
    return nullptr; // not found
}
```

14.5.2 F.43: Never (directly or indirectly) return a pointer or a reference to a local object

Rule: Do not return references/pointers to objects with automatic storage duration (locals) or to subobjects that will not outlive the caller. **Why:** Prevents dangling references/pointers and undefined behavior.

```
const int* bad() {
    int x = 7;
    return &x; // dangling
}
```

14.5.3 F.44: Return a `T&` when copy is undesirable and "returning no object" isn't needed

Rule: Return `T&` to refer to an existing object when the function must not create a copy and always has a valid object to return. **Why:** Conveys non-optional result and avoids copies.

```
#include <vector>
```

```
int& at_ref(std::vector<int>& v, std::size_t i) {
    return v.at(i); // always returns a reference or throws
}
```

14.5.4 F.45: Don't return a `T&&`

Rule: Avoid returning rvalue references. **Why:** Returned rvalue references frequently dangle or mislead callers; they complicate lifetime reasoning.

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

Rule: Use `int main()` (with standard signatures). **Why:** Required by the C++ standard; conveys process exit status.

```
int main() {
    return 0;
}
```

14.5.6 F.47: Return `T&` from assignment operators

Rule: `operator=` should return `T&`. **Why:** Matches established convention and enables chaining.

```
struct X {
    X& operator=(const X&) = default;
};
```

14.5.7 F.48: Don't return `std::move(local)`

Rule: Do not explicitly move a local on return. **Why:** It can inhibit copy elision and is unnecessary in normal cases; let return-value optimization (and moves when needed) occur naturally.

```cpp
#include <string>

std::string good() {
    std::string s = "abc";
    return s;              // allow elision / move
}

// std::string bad() { std::string s="abc"; return std::move(s); }
```

14.5.8 F.49: Don't return `const T`

Rule: Do not return `const` by value. **Why:** It interferes with move semantics and rarely provides a useful guarantee; constness is a property of the receiving object.

14.6 Other function rules

14.6.1 F.50: Use a lambda when a function won't do (capture local variables or write a local function)

Rule: Use lambdas for local behavior, especially when capturing local state. **Why:** Keeps local logic close to use and allows capturing without global/static state.

```cpp
#include <algorithm>
#include <vector>

int count_greater_than(const std::vector<int>& v, int limit) {
    return (int)std::count_if(v.begin(), v.end(), [limit](int x){ return x > limit; });
}
```

14.6.2 F.51: Where there is a choice, prefer default arguments over overloading

Rule: Prefer default parameters for small variations in behavior. **Why:** Reduces overload sets and keeps call sites readable.

```
#include <string>

void log(const std::string& msg, int level = 1);
```

14.6.3 F.52: Prefer capturing by reference in lambdas that will be used locally

Rule: For short-lived lambdas used immediately (e.g., in an algorithm call), capture by reference when safe. **Why:** Avoids unnecessary copying and keeps changes visible when intended.

```
#include <algorithm>
#include <vector>

void negate_all(std::vector<int>& v) {
    int factor = -1;
    std::for_each(v.begin(), v.end(), [&](int& x){ x *= factor; });
}
```

14.6.4 F.53: Avoid capturing by reference in lambdas that will be used non-locally

Rule: If a lambda may outlive the scope (returned, stored, heap-allocated, used on another thread), do not capture by reference. **Why:** Prevents dangling references and data races; capture by value (or store shared state explicitly) instead.

```
#include <functional>
#include <string>

std::function<void()> make_printer(std::string s) {
    // capture by value because the lambda escapes the function scope
    return [s]{ /* use s */ };
}
```

14.6.5 F.54: When writing a lambda that captures `this` or any class data member, don't use `[=]` default capture

Rule: Avoid `[=]` when capturing object state; capture explicitly. **Why:** `[=]` can capture `this` implicitly and lead to lifetime bugs (lambda escapes while the object dies) and unclear intent. Prefer explicit captures such as `[this]` or (when appropriate) capturing needed members by value.

```
#include <functional>
#include <string>

class C {
public:
    std::function<std::string()> make() const {
        // explicit intent:
        return [this]{ return name_; };
        // or: auto copy = name_; return [copy]{ return copy; };
    }
private:
    std::string name_{"x"};
};
```

14.6.6 F.55: Don't use `va_arg` arguments

Rule: Avoid C-style variadic functions with `va_list`/`va_arg`. **Why:** They are not type-safe and are error-prone. Prefer overloads, variadic templates, or type-safe formatting APIs.

```
template<class... Args>
void logf(Args&&... args) {
    // type-safe variadic interface (implementation omitted)
}
```

14.6.7 F.56: Avoid unnecessary condition nesting

Rule: Prefer early exits and clear structure over deep nesting. **Why:** Improves readability and reduces logic errors.

```
bool handle(int x) {
    if (x < 0) return false;
    if (x == 0) return true;
    // main path
    return true;
}
```

14.6.8 F.57: Not assigned in the Jul 8, 2025 revision

Status: No guideline with number F.57 exists in the published numbering for this revision.

14.6.9 F.58: Not assigned in the Jul 8, 2025 revision

Status: No guideline with number F.58 exists in the published numbering for this revision.

14.6.10 F.59: Not assigned in the Jul 8, 2025 revision

Status: No guideline with number F.59 exists in the published numbering for this revision.

14.7 F.call: Cross-cutting pointer/reference selection rule

14.7.1 F.60: Prefer `T*` over `T&` when "no argument" is a valid option

Rule: Use a pointer when "no object" is a meaningful possibility; use a reference when a valid object is required. **Why:** References are non-null by contract (in normal correct code). Pointers naturally support an explicit "none" value (`nullptr`) but require checking.

```
#include <string>

std::string to_string_or_empty(const char* p) { // p may be nullptr
    if (!p) return {};
    return std::string{p};
}

void print_size(const std::string& s) { // s must exist
    (void)s.size();
}
```

Chapter 15

Function Design Principles

15.1 Introduction

Functions are fundamental building blocks in C++. Good function design improves readability, correctness, maintainability, testability, and reuse. Poorly designed functions lead to bugs, unclear interfaces, and fragile code. This chapter explains the core principles of function design based on the ISO C++ Core Guidelines and modern C++ best practices.

15.2 Purpose of Function Design Principles

A function should:

- Represent a single, well-defined task.
- Have a clear interface that expresses intent.
- Minimize dependencies and hidden side effects.
- Respect ownership and lifetime rules.

- Be easy to test and reason about.

In following sections, we explain these principles with rationale and example code.

15.3 Fundamental Design Principles

15.3.1 Single Responsibility: One Function, One Purpose

Principle: A function should perform a single logical operation.
Reason: Functions that try to do many things become hard to understand, test, reuse, and maintain. Keeping functions focused improves clarity.
Guidelines:

- If a function does more than one conceptual thing, break it into smaller functions.
- Avoid combining input, processing, and output code in one function when it can be split.

```
// Bad: reads input and prints result
void handle_input_and_print() {
    int x;
    std::cin >> x;
    std::cout << "Value: " << x << '\n';
}

// Good: separate input and output responsibilities
int read_int(std::istream &is) {
    int x{};
    is >> x;
    return x;
}

void print_int(std::ostream &os, int x) {
```

```
    os << "Value: " << x << '\n';
}
```

15.3.2 Clear and Minimal Interfaces

Principle: A function's signature should express intent clearly.
Reason: The interface is the contract between the caller and the function. It should communicate what is required, what is optional, and what is produced.
Guidelines:

- Use parameter types to express intent: value, reference, pointer, or smart pointer.
- Minimize the number of parameters; group related parameters into structures if needed.
- Prefer expressive names for parameters and return values.

15.3.3 Prefer Return Values to Output Parameters

Principle: Use return values instead of modifying output parameters when possible.
Reason: Return values are easier to understand and compose. Modern compilers optimize return values effectively.

```
// Good: return a structured result
struct ParseResult {
    int value;
    std::size_t chars_used;
};

ParseResult parse_int(std::string_view s);

// Avoid: mutation through output parameter
void parse_int_bad(std::string_view s, int &value, std::size_t &used);
```

15.3.4 Minimize Side Effects

Principle: Functions should avoid hidden side effects.

Reason: Side effects make reasoning about code harder and increase coupling.

Guidelines:

- Prefer pure functions (no global state, no mutation outside scope).
- Limit side effects to clearly documented operations.

```
// Pure function: no side effects
int clamp(int x, int lo, int hi) {
    if (x < lo) return lo;
    if (x > hi) return hi;
    return x;
}
```

15.3.5 Express Ownership and Lifetime Correctly

Principle: Function parameters and return types should convey ownership semantics.

Reason: Ownership and lifetime miscommunication lead to resource leaks, use-after-free, and other bugs.

Guidelines:

- Use references for required, non-null arguments.
- Use pointers to denote optional (nullable) objects.
- Use `std::unique_ptr` and `std::shared_ptr` to express owning semantics.
- Avoid raw owning pointers unless interfacing with legacy code.

```
// Required, non-null argument
void process(const Widget &w);

// Optional argument
void maybe_process(Widget *w);

// Ownership transfer
void take_ownership(std::unique_ptr<Widget> w);
```

15.4 Parameter Passing Strategies

15.4.1 Pass by Value for Small, Cheap-to-Copy Types

Principle: For small or trivially copyable types, pass by value.

Reason: Passing by value can be as fast or faster than passing by reference for small types and avoids dangling references.

```
void set_timeout(int ms); // small int passed by value
```

15.4.2 Pass by `const&` for Read-Only Large Types

Principle: For larger objects not cheap to copy, use `const T&` for read-only parameters.

```
void log_message(const std::string &msg);
```

15.4.3 Pass by Reference for Mutating Parameters

Principle: Use non-const references (`T&`) for parameters the function will modify.

```
void scale_in_place(std::vector<float> &data, float factor);
```

15.4.4 Perfect Forwarding in Templates

Principle: Use forwarding references in generic code to preserve value category.

```
template <typename F, typename... Args>
decltype(auto) invoke(F &&f, Args &&...args) {
    return std::forward<F>(f)(std::forward<Args>(args)...);
}
```

15.5 Function Safety Principles

15.5.1 Use `noexcept` Where Applicable

Principle: Declare functions `noexcept` when they are guaranteed not to throw exceptions.

Reason: `noexcept` documents intent, enables optimization, and improves exception safety.

```
void clear_buffer() noexcept;
```

15.5.2 Consistent Error Reporting

Principle: Functions should use consistent and clear error reporting strategies.

Guidelines:

- Use exceptions for error propagation in general-use interfaces.
- Use expected-like types or status codes for performance-critical low-level code.
- Do not mix error reporting mechanisms arbitrarily.

15.6 Lambda and Local Function Design

15.6.1 Use Lambdas for Local Tasks

Principle: Use lambda expressions when a small, inline function is needed close to the call site.

```
std::sort(data.begin(), data.end(),
    [](auto &a, auto &b) { return a.value < b.value; });
```

15.6.2 Capture Variables Explicitly

Principle: Prefer explicit capture lists over default captures.

```
auto work = [count, &logger]() { /* body */ };
```

15.7 Summary Checklist

When designing functions, ask:

- Does the function have a single clear responsibility?
- Is the function interface clear and expressive?
- Are ownership and lifetime semantics properly expressed?
- Are side effects minimized and explicit?
- Are parameters passed in the most suitable way?
- Are lambda captures explicit?
- Is error handling consistent and clear?

Chapter 16

Parameter Passing

16.1 Introduction

Parameter passing is part of a function's **contract**. In modern C++, the parameter types you choose should make the caller immediately understand:

- what the function *needs* (**input**),
- what the function *modifies* (**in-out**),
- what the function *produces* (**output**),
- whether the function *consumes* the argument (**will-move-from**),
- whether the function *forwards* the argument (**perfect forwarding**),
- whether "no argument" is a meaningful option (**optional argument**).

These principles are summarized in the C++ Core Guidelines rules `F.15--F.21` and the semantic rules `F.22--F.27`, plus `F.60`.

16.2 Design goal: Make intent obvious at the call site

A good signature makes correct usage natural and incorrect usage difficult:

- **Read-only input**: use value (if cheap) or `const T&`.
- **Mutation**: use `T&` for in-out.
- **Output**: prefer return values; avoid "out-parameters" when possible.
- **Ownership**: use `std::unique_ptr` / `std::shared_ptr` only to express ownership.
- **Optional**: use `T*` (nullable) or `std::optional<T>` depending on meaning.

16.3 F.call: Core parameter passing rules (F.15–F.21)

16.3.1 F.15: Prefer simple and conventional ways of passing information

Rule: Prefer straightforward parameter passing forms. Use advanced techniques only after demonstrating a real need and document that need.
Rationale: Clever passing styles create surprises, slow comprehension, and encourage bugs. Optimizations that are not measured are often wrong or brittle across architectures and compilers.

16.3.2 A practical intent table (normal techniques)

The following table is a compact, adoption-friendly summary of the recommended "normal" techniques implied by `F.16--F.21`.

Intent	Preferred form	Notes
Input-only (cheap to copy)	`T` by value	Best for small scalars and small trivially copyable types.
Input-only (not cheap to copy)	`const T&`	Avoids copy; still accepts rvalues.
In-out (modify)	`T&`	Signals mutation clearly.
Output-only	return value	Self-documenting; composes well.
Multiple outputs	return `struct` or `tuple`	Prefer named `struct` for stable APIs.
Optional argument	`T*` (nullable)	Use `nullptr` to mean "no object".

16.3.3 F.16: For "in" parameters, pass cheaply-copied types by value and others by reference to `const`

Rule: If an input parameter is cheap to copy, pass it by value; otherwise pass by `const T&`.
Key idea: "Cheap to copy" depends on architecture, but very small objects (typically a few machine words) are often best passed by value. Passing a small object by `const&` can add indirection overhead.

```
#include <string>
#include <string_view>

void set_timeout_ms(int ms);              // cheap: pass by value
void set_ratio(double r);                 // cheap: pass by value

void log_line(const std::string& s);      // potentially expensive: const&
std::string normalize(std::string_view); // small view type: pass by value
```

16.3.4 F.17: For "in-out" parameters, pass by reference to non-`const`

Rule: If the function modifies the argument, use a non-const reference `T&`.

Rationale: The type tells the truth: callers should expect mutation.

```
#include <string>

void trim_right_in_place(std::string& s) {
    while (!s.empty() && (s.back() == ' ' || s.back() == '\t' || s.back() == '\n')) {
        s.pop_back();
    }
}
```

Important caution: A `T&` can carry both input and output information, which can be a source of non-obvious errors. Keep in-out parameters intentional and minimal.

16.3.5 F.18: For "will-move-from" parameters, pass by `X&&` and `std::move` the parameter

Rule: If the function will unconditionally move from an argument, accept it as `X&&` and move it in the function body.

Rationale: The signature communicates consumption; `std::move` makes the transfer explicit.

```
#include <string>
#include <vector>
#include <utility>

struct Sink {
    std::vector<std::string> storage;

    void put(std::string&& s) {
        storage.push_back(std::move(s)); // consume s
```

```
    }
};
```

Note: Move-only owner types like `std::unique_ptr` are also often fine to take *by value* for simplicity, since passing by value implies a move at the call site.

```
#include <memory>

struct Widget {};

void sink(std::unique_ptr<Widget> p) {
    // owns p; p destroyed on return unless moved onward
}
```

16.3.6 F.19: For "forward" parameters, pass by `TP&&` and only `std::forward` the parameter

Rule: Use a forwarding reference only when you truly intend to pass the argument onward while preserving constness and value category. In that case, forward exactly once on every path.

Rationale: A `TP&&` parameter is a declaration that this function itself does not want to interpret the argument, only forward it.

```
#include <utility>

template<class F, class... Args>
decltype(auto) invoke(F&& f, Args&&... args) {
    return std::forward<F>(f)(std::forward<Args>(args)...);
}
```

16.3.7 F.20: For "out" output values, prefer return values to output parameters

Rule: Prefer returning results rather than writing them into output parameters.
Rationale: Return values are self-documenting; an `&` parameter is ambiguous (in-out or out-only) and more easily misused. Modern C++ return-value optimization and move semantics make returning large objects practical.

```
#include <vector>

std::vector<int> filter_positive(const std::vector<int>& v) {
    std::vector<int> out;
    for (int x : v) if (x > 0) out.push_back(x);
    return out; // efficient in modern C++
}
```

16.3.8 F.21: To return multiple "out" values, prefer returning a `struct` or `tuple`

Rule: Return multiple results by returning a multi-member type: a named `struct` is often best for stable interfaces; `std::tuple` is acceptable for simple cases.

```
#include <cstddef>
#include <string_view>

struct ParseResult {
    int value;
    std::size_t chars_used;
};

ParseResult parse_int(std::string_view s) {
    int v = 0;
```

```
    std::size_t i = 0;
    for (; i < s.size(); ++i) {
        char c = s[i];
        if (c < '0' || c > '9') break;
        v = v * 10 + (c - '0');
    }
    return {v, i};
}

// Usage:
auto r = parse_int("123xyz");
// r.value == 123, r.chars_used == 3
```

16.4 Parameter passing semantic rules (F.22–F.27)

These rules focus on expressing **ownership**, **non-nullness**, and **sequence semantics** directly in types.

16.4.1 F.22: Use `T*` or `owner<T*>` to designate a single object

Rule: Use `T*` (non-owning) or `owner<T*>` (owning raw pointer, typically at legacy boundaries) to represent a pointer to a single object.
Guidance:

- Use `T*` for observation (non-owning).
- Avoid owning raw pointers in new code; prefer `std::unique_ptr<T>` or `std::shared_ptr<T>`.
- Use `owner<T*>` only when interoperability requires raw ownership.

```
#include <gsl/pointers>
```

```
#include <memory>

using gsl::owner;

struct Node {};

void observe(const Node* p);      // non-owning
void take_raw(owner<Node*> p);    // owning raw pointer (legacy boundary)

void take_unique(std::unique_ptr<Node> p); // preferred modern ownership transfer
```

16.4.2 F.23: Use `not_null<T>` to indicate that "null" is not a valid value

Rule: If null is never valid, say so in the type with `not_null`.
Rationale: Non-null is part of the contract; encoding it improves correctness and documentation.

```
#include <gsl/pointers>
using gsl::not_null;

struct Widget { void draw() const; };

void draw_widget(not_null<const Widget*> w) {
    w->draw();
}
```

16.4.3 F.24: Use `span<T>` to designate a half-open sequence

Rule: Use `std::span<T>` (or an equivalent) to represent a contiguous sequence as (pointer, length) with safer semantics.
Why it matters: Raw pointer + separate length is easy to mismatch; `span` keeps them together and supports range-style iteration.

```
#include <span>

void scale(std::span<float> a, float k) {
    for (float& x : a) x *= k;
}
```

16.4.4 F.25: Use `zstring` or `not_null<zstring>` to designate a C-style string

Rule: If the parameter is a zero-terminated C string, express that meaning with `zstring`. If it cannot be null, use `not_null<zstring>`.

```
#include <gsl/pointers>
#include <string>

using gsl::zstring;
using gsl::not_null;

std::string copy_nullable(zstring s) {
    if (!s) return {};
    return std::string{s};
}

std::string copy_nonnull(not_null<zstring> s) {
    return std::string{s};
}
```

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

Rule: When a function needs to take ownership of an object via a pointer, use `std::unique_ptr<T>`.

Rationale: It is the cheapest safe owning pointer and makes ownership transfer explicit.

```
#include <memory>

struct Shape { virtual ~Shape() = default; };

void set_current(std::unique_ptr<Shape> s); // ownership transfer

void demo(std::unique_ptr<Shape> s) {
    set_current(std::move(s)); // explicit transfer
}
```

16.4.6 F.27: Use `shared_ptr<T>` to share ownership

Rule: Use `std::shared_ptr<T>` only when you truly need shared ownership.

Guidance:

- Shared ownership is a strong coupling of lifetimes; do not use it by default.
- Prefer `unique_ptr` (single owner) whenever possible.

```
#include <memory>

struct Config {};

void start_worker(std::shared_ptr<const Config> cfg);

void demo() {
    auto cfg = std::make_shared<Config>();
    start_worker(cfg); // shared lifetime across threads/tasks
}
```

16.5 F.60: Optional arguments

16.5.1 F.60: Prefer `T*` over `T&` when "no argument" is a valid option

Rule: If "no object" is meaningful, use a pointer (`T*`) and allow `nullptr`. Do not use references for optional parameters.

Rationale: References cannot represent "no object"; pointers can.

```
#include <string>

void maybe_log(const std::string* msg) {
    if (!msg) return;
    // use *msg
}

void must_log(const std::string& msg) {
    // msg always valid
}
```

When to consider `std::optional<T>`:

- Use `std::optional<T>` when "no value" is a meaningful *value-level* state (especially for small value types),
- Use `T*` when the parameter denotes an *optional object identity* (an optional object to act on).

16.6 Common modern C++ patterns

16.6.1 Strings: prefer `std::string_view` for read-only input

For read-only string input, `std::string_view` is often the best "in" parameter: it avoids allocation/copy and accepts string literals, `std::string`, and other contiguous character ranges.

```
#include <string_view>

bool starts_with_http(std::string_view s) {
    return s.rfind("http", 0) == 0;
}
```

Important: A `string_view` does not own data. Do not store it unless you also guarantee the referenced data outlives the view.

16.6.2 Sequences: prefer `std::span` for contiguous ranges

`std::span` generalizes "pointer + length" and makes interfaces clearer and less error-prone.

```
#include <span>
#include <cstdint>

std::uint32_t checksum(std::span<const std::uint8_t> bytes) {
    std::uint32_t sum = 0;
    for (auto b : bytes) sum = (sum * 131u) + b;
    return sum;
}
```

16.6.3 Avoid "smart pointer parameters" unless you mean ownership

A common API mistake is taking `std::shared_ptr<T>` just to access an object. That silently forces shared ownership and reference counting on callers. Prefer `T&` or `T*` for non-owning access.

```cpp
struct Widget {};

// Avoid: suggests shared ownership required
void draw_bad(std::shared_ptr<Widget> w);

// Prefer: non-owning access
void draw_good(const Widget& w);
```

16.7 Pitfalls and how to avoid them

16.7.1 Pitfall: output parameters that look like inputs

If you must use an out-parameter (e.g., to avoid allocation in tight loops), make it unambiguous and document it clearly. Otherwise, return values are preferred.

16.7.2 Pitfall: forwarding references used "for performance"

Do not use `T&&` in non-template code just to appear efficient; many such micro-optimizations are false or brittle. Prefer the normal techniques (value or `const&`) unless you have measured proof.

16.7.3 Pitfall: dangling references/views

Any non-owning handle (`T*`, `T&`, `string_view`, `span`) requires that the referred object outlive the call and (if stored) outlive the handle. Treat this as part of the function's contract.

16.8 Practical checklist

Before finalizing a function signature, confirm:

- Is each parameter clearly **in**, **in-out**, **out**, **consume**, **forward**, or **optional**?
- Are large input objects passed as `const&` (unless intentionally copied)?
- Are in-out parameters `T&` and truly modified?
- Are outputs returned by value (or as a structured result) when reasonable?
- Do ownership types (`unique_ptr`/`shared_ptr`) appear only when ownership is part of the API?
- Are nullable parameters represented as `T*` (or `optional`) rather than misusing references?
- For sequences, is `span` used instead of raw pointer+length?

Chapter 17

Return Values

17.1 Introduction

A return value is often the **cleanest and safest** way for a function to communicate its result. Modern C++ is designed to make returning by value efficient (copy elision, move semantics), so return values are usually preferable to "out parameters" for conveying results.
This chapter focuses on the C++ Core Guidelines **value return semantic rules**: `F.42--F.49`.

17.2 Design goals for return values

A well-designed return type should:

- **Express meaning:** what does the function return (value, position, reference, ownership)?
- **Express lifetime:** how long is the returned object valid?
- **Enable safe usage:** make incorrect usage difficult (avoid dangling references).

- **Compose naturally:** allow chaining and use in expressions.
- **Avoid hidden coupling:** do not encode ownership or shared lifetime accidentally.

17.3 Return categories and when to use them

17.3.1 Return by value (`T`)

Return by value when the function produces a **new value** or a **result object**. This is the default choice for most APIs.

```
#include <vector>

std::vector<int> filter_positive(const std::vector<int>& v) {
    std::vector<int> out;
    for (int x : v) if (x > 0) out.push_back(x);
    return out; // efficient in modern C++
}
```

17.3.2 Return a pointer (`T*`)

Return a pointer when the function returns a **position** and "not found" is meaningful (`nullptr`). A pointer return should not imply ownership transfer.

17.3.3 Return a reference (`T&`)

Return a reference when the function returns an existing object that is owned elsewhere and **must exist** (no "not found" state).

17.3.4 Return an optional-like result

Use `std::optional<T>` when the return value is a **value** that may be absent. Use pointers for optional **object identity** (e.g., position in a container).

```
#include <optional>
#include <string_view>

std::optional<int> parse_int(std::string_view s) {
    if (s.empty()) return std::nullopt;
    int v = 0;
    for (char c : s) {
        if (c < '0' || c > '9') return std::nullopt;
        v = v * 10 + (c - '0');
    }
    return v;
}
```

17.4 C++ Core Guidelines: Value return semantic rules (F.42–F.49)

17.4.1 F.42: Return a `T*` to indicate a position (only)

Rule: Return a raw pointer to indicate a **position** (and use `nullptr` for "not found"), not to transfer ownership.

Key idea: A returned `T*` should mean "this points to an existing object in a structure" or "this is a location", not "you must delete this".

```
#include <string>

struct Node {
```

```
    std::string name;
    Node* left{};
    Node* right{};
};

// Returns a position in the tree; does NOT transfer ownership.
Node* find(Node* t, const std::string& s) {
    if (!t || t->name == s) return t;
    if (auto* p = find(t->left, s))  return p;
    if (auto* p = find(t->right, s)) return p;
    return nullptr;
}
```

17.4.2 F.43: Never (directly or indirectly) return a pointer or reference to a local object

Rule: Never return a pointer/reference that would dangle after the function returns.

Reason: Locals are destroyed at the end of the function. Returning their address or reference causes undefined behavior.

```
#include <string>

// Don't:
const std::string& bad_ref() {
    std::string s = "tmp";
    return s; // dangling reference
}

const char* bad_ptr() {
    std::string s = "tmp";
    return s.c_str(); // dangling pointer
}
```

```
// Do:
std::string good_value() {
    std::string s = "tmp";
    return s; // returned by value (copy elision / move)
}
```

Common indirect form: returning a pointer/reference to a local through another function still violates the rule.

17.4.3 F.44: Return a T& when copy is undesirable and "returning no object" isn't needed

Rule: Return an lvalue reference when you want to return an existing object, copying is undesirable, and there is no valid "empty" result.
Typical use: element accessors, lookup operations that are guaranteed to succeed (or throw).

```
#include <vector>
#include <cstddef>

struct Table {
    std::vector<int> data;

    int& at(std::size_t i) { return data.at(i); }            // throws if invalid
    const int& at(std::size_t i) const { return data.at(i); }
};
```

If "not found" is possible: prefer returning T* (F.42) or an optional-like result.

17.4.4 F.45: Don't return a T&&

Rule: Do not return an rvalue reference.

Reason: A return type of `T&&` almost always misrepresents lifetime or ownership and encourages dangling usage. If you want to return a new object, return by value.

```
// Don't:
std::string&& make_name_bad(); // almost always wrong

// Do:
std::string make_name_good();
```

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

Rule: `main` must return `int`.

```
int main() {
    // ...
    return 0;
}
```

17.4.6 F.47: Return `T&` from assignment operators

Rule: Assignment operators should return `T&` (usually `*this`).

Reason: Enables chaining and matches conventional assignment semantics.

```
struct S {
    int x{};

    S& operator=(const S& other) {
        x = other.x;
        return *this;
    }
};
```

17.4.7 F.48: Don't `return std::move(local)`

Rule: Do not `return std::move(local)` from a local variable.

Reason: It is redundant and can inhibit named return value optimization (NRVO). Returning the local by name allows copy elision; if elision is not performed, move can still occur.

```
#include <vector>

// Don't:
std::vector<int> make_bad() {
    std::vector<int> v{1,2,3};
    return std::move(v);
}

// Do:
std::vector<int> make_good() {
    std::vector<int> v{1,2,3};
    return v; // NRVO or move as needed
}
```

17.4.8 F.49: Don't return `const T`

Rule: Do not return `const T` by value.

Reason: Returning `const` by value prevents moving from the result and adds no meaningful safety. The caller already controls whether the returned value is treated as mutable.

```
#include <string>

// Don't:
const std::string make_bad();

// Do:
std::string make_good();
```

17.5 Practical patterns for robust return-value APIs

17.5.1 Pattern: “Maybe” results (absence is normal)

Use `std::optional<T>` for “no value” outcomes. Prefer a pointer return when the meaning is “position in an existing object graph”.

17.5.2 Pattern: Return structured results

When multiple values are logically related, return a named `struct`.

```
#include <cstddef>
#include <string_view>

struct ParseResult {
    int value;
    std::size_t used;
    bool ok;
};

ParseResult parse_int_relaxed(std::string_view s) {
    int v = 0;
    std::size_t i = 0;
    if (s.empty()) return {0, 0, false};

    for (; i < s.size(); ++i) {
        char c = s[i];
        if (c < '0' || c > '9') break;
        v = v * 10 + (c - '0');
    }
    return {v, i, i != 0};
}
```

17.5.3 Pattern: Error signaling

Choose one consistent strategy per API layer:

- Exceptions for general application-level APIs.
- Status codes / `std::optional` / expected-like results for low-level or performance-critical layers.

17.6 Common mistakes and how to avoid them

17.6.1 Returning views or pointers to temporary data

Do not return:

- pointers to internal buffers of local temporaries,
- references/views (`string_view`, `span`) that refer to objects that will not outlive the caller's use.

If you must return a view, ensure it refers to an object owned by the caller or by a longer-lived object.

17.6.2 Returning owning raw pointers

If the caller must own the result, return `std::unique_ptr<T>` (or `std::shared_ptr<T>` for shared ownership), not a raw pointer.

```
#include <memory>

struct Widget {};
```

```
std::unique_ptr<Widget> make_widget() {
    return std::make_unique<Widget>();
}
```

17.7 Checklist

Before finalizing a return type, verify:

- Does the return type clearly express meaning (value vs position vs existing object)?
- Is lifetime safe (no pointer/reference to local state)?
- If "no result" is possible, is it represented explicitly (pointer `nullptr`, `optional`, or expected-like)?
- Is ownership explicit (use `unique_ptr`/`shared_ptr` when ownership is returned)?
- Are you avoiding `T&&`, `return std::move(local)`, and `const T` return-by-value?

Chapter 18

`noexcept` Usage

18.1 Introduction

In modern C++, `noexcept` is both a **semantic contract** and a **tool for optimization**. It tells callers and the compiler that a function *must not allow an exception to leave it*. If an exception does leave a `noexcept` function, the C++ runtime will call `std::terminate()`, ending the program. This is not error handling; it is a strong guarantee.
The C++ Core Guidelines summarize the central rule as: **F.6: If your function must not throw, declare it `noexcept`.**

18.2 What `noexcept` means (and what it does not mean)

18.2.1 Guaranteed behavior when violated

Rule: If a function is declared `noexcept` and an exception exits that function, the program calls `std::terminate()`.

```
#include <stdexcept>
```

```
void must_not_throw() noexcept {
    throw std::runtime_error("boom"); // if this escapes, std::terminate is called
}
```

18.2.2 `noexcept` does not mean "no failures"

A `noexcept` function can still fail in other ways:

- It can return an error code or an optional/expected-like object.
- It can set an error flag in an output parameter.
- It can call `std::terminate()`, `std::abort()`, or similar as part of its design.

What it *cannot* do is let exceptions propagate to the caller.

18.3 Core Guidelines perspective (F.6 and related guidance)

18.3.1 F.6: If your function must not throw, declare it `noexcept`

Meaning: Use `noexcept` when throwing is a bug and callers must be able to rely on a non-throwing guarantee (especially for low-level utilities and library building blocks).
Practical consequences:

- Callers may safely use your function inside their own `noexcept` functions.
- Generic code may prefer your operations when they are `noexcept` (notably moves).
- If your promise is wrong, the failure mode is termination, not recovery.

18.4 Where `noexcept` matters most

18.4.1 Destructors and deallocation functions

Destructors are expected to not allow exceptions to escape during stack unwinding. Letting exceptions escape from destructors (or deallocation functions) can trigger termination and, in some cases, undefined behavior. Design destructors to be non-throwing and to handle errors internally (e.g., log, set a flag, or provide an explicit `close()` that can report errors).

```
#include <cstdio>

class File {
public:
    explicit File(const char* path) : f_(std::fopen(path, "rb")) {}

    ~File() noexcept {
        if (f_) std::fclose(f_); // ignore failure here (cannot throw)
    }

    // If error reporting is required, provide an explicit operation:
    bool close() noexcept {
        if (!f_) return true;
        const int rc = std::fclose(f_);
        f_ = nullptr;
        return rc == 0;
    }

private:
    std::FILE* f_{};
};
```

18.4.2 Move operations and performance-critical generic code

A key design point in modern C++ is that generic code (including standard containers) can select move operations preferentially when they are known not to throw. Therefore, many types should strive to make move construction/assignment `noexcept` when possible.

```
#include <utility>

struct S {
    S() = default;

    // If this truly cannot throw, say so:
    S(S&&) noexcept = default;
    S& operator=(S&&) noexcept = default;
};
```

Warning: Do not mark move operations `noexcept` if they can throw. If your move can throw, keep it potentially throwing and rely on strong exception-safety patterns or redesign the representation.

18.4.3 `swap` and other "must-not-fail" primitives

Non-throwing `swap` is a common building block for exception-safe code. When your type can provide a non-throwing swap, declare it `noexcept` and implement it in terms of non-throwing operations.

```
#include <utility>

class Buffer {
public:
    void swap(Buffer& other) noexcept {
        std::swap(p_, other.p_);
```

```
        std::swap(n_, other.n_);
    }

private:
    unsigned char* p_{};
    std::size_t n_{};
};
```

18.5 Unconditional vs conditional `noexcept`

18.5.1 Unconditional `noexcept`

Use unconditional `noexcept` only when you are confident the function cannot throw for any template instantiation or runtime path (including callees).

18.5.2 Conditional `noexcept` (best for templates and wrappers)

C++ allows `noexcept` to be conditional:

```
noexcept(condition)
```

This is often the correct approach for templates that depend on operations of `T`. It preserves correctness while giving generic code as much `noexcept` information as possible.

```
#include <type_traits>
#include <utility>

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

```
    T tmp = std::move(a);
    a = std::move(b);
    b = std::move(tmp);
}
```

18.6 The `noexcept` operator

The `noexcept(expr)` operator returns a constant expression of type `bool` indicating whether `expr` is known not to throw. It is widely used to propagate non-throwing guarantees through templates.

```
#include <utility>

template<class T>
constexpr bool move_is_noexcept =
    noexcept(T{std::declval<T&&>()}); // true if move construction is noexcept
```

18.7 API design guidance: when to use `noexcept`

18.7.1 Good candidates

Mark a function `noexcept` when:

- It is a low-level primitive where throwing is not part of the design.
- It is used in cleanup paths (destructors, `close()`, rollback helpers).
- It is a move constructor/assignment that truly cannot throw.
- It is a small utility that cannot allocate and cannot throw (e.g., simple arithmetic, pointer manipulations).

This aligns with the Core Guidelines intent of `F.6`.

18.7.2 Be careful (often *do not* use `noexcept`)

Avoid marking a function `noexcept` when:

- It can allocate memory, format strings, or perform I/O in ways that can throw.
- It calls user-provided callbacks or virtual functions that may throw.
- It depends on template operations of `T` that might throw (use conditional `noexcept` instead).
- You are using `noexcept` as a micro-optimization without a correctness-based reason.

If you cannot keep the promise, `noexcept` converts exceptions into program termination.

18.8 Patterns for `noexcept`-friendly design

18.8.1 Split "can-fail" from "cannot-fail"

A common pattern is to provide:

- a throwing function that reports errors by exception,
- and a non-throwing function that reports errors explicitly (status return).

```
#include <string>
#include <optional>

std::string load_text_throwing(const std::string& path); // may throw

std::optional<std::string> load_text_noexcept(const std::string& path) noexcept {
    try {
        return load_text_throwing(path);
```

```
    } catch (...) {
        return std::nullopt; // explicit error channel
    }
}
```

Note: Using `try/catch` to implement a `noexcept` boundary is valid, but use it intentionally: you are choosing to convert exceptions into explicit error results.

18.8.2 Keep `noexcept` functions "simple" and dependency-minimal

For a strong non-throwing contract, avoid:

- allocations,
- formatting,
- complex library calls with exception-based failure,
- code that depends on external resources with unclear failure modes.

In other words, `noexcept` functions should be designed to be robust under failure.

18.9 Common mistakes and how to avoid them

18.9.1 Mistake: declaring `noexcept` because "it probably won't throw"

Fix: Declare `noexcept` only if it *must not* throw. Otherwise, leave it potentially throwing. The cost of being wrong is termination.

18.9.2 Mistake: a `noexcept` function calls something that might throw

Fix: Either:

- remove `noexcept`, or
- guard and translate errors (catch and convert), or
- redesign to avoid throwing dependencies.

18.9.3 Mistake: returning error information by throwing from destructors

Fix: Provide explicit error-reporting operations and keep destructors non-throwing.

18.10 Checklist

Before adding `noexcept`, confirm:

- Is throwing considered a bug in this function's contract?
- Do all callees on all paths satisfy the non-throwing requirement?
- Are template-dependent operations protected with conditional `noexcept`?
- Is the failure policy acceptable (termination if violated)?
- For cleanup code, is error reporting handled explicitly (status return, logging, etc.)?

Chapter 19

Small vs. Large Functions

19.1 Introduction

In modern C++, a “good” function is not defined by a fixed line count. It is defined by how clearly it expresses intent, how safely it can be used, how easy it is to test, and how well it composes with other code.
The C++ Core Guidelines emphasize three closely related ideas for function structure:

- **F.2:** A function should perform a single logical operation.
- **F.3:** Keep functions short and simple.
- **F.5:** If a function is very small and time-critical, declare it `inline`.

This chapter explains the engineering trade-offs between small and large functions, and how to apply these rules responsibly in real projects.

19.2 Core design principles

19.2.1 One logical operation (F.2)

Rule (F.2): A function should perform a single logical operation.

Why it matters:

- A single-purpose function is easier to name accurately.
- It is easier to test in isolation.
- It is easier to reuse.
- It reduces accidental coupling between unrelated tasks.

```
#include <string>
#include <cctype>

// Bad: parses + validates + normalizes + reports errors
bool parse_and_report(const std::string& s) {
    if (s.empty()) { /* report */ return false; }
    for (char c : s) if (!std::isdigit(static_cast<unsigned char>(c))) { /* report */
    ↪  return false; }
    // normalize...
    return true;
}

// Good: single logical responsibilities
bool is_all_digits(const std::string& s) {
    if (s.empty()) return false;
    for (char c : s)
        if (!std::isdigit(static_cast<unsigned char>(c))) return false;
    return true;
}
```

19.2.2 Keep functions short and simple (F.3)

Rule (F.3): Keep functions short and simple.

What "short and simple" means in practice:

- Few levels of nesting.
- Clear control flow (guard clauses, early returns).
- Small number of local variables with minimal scope.
- Little duplication; repeated logic is extracted into helpers.

```
#include <string_view>

// Good: guard clauses keep the happy path flat and readable
bool is_valid_port(std::string_view s) {
    if (s.empty()) return false;

    int port = 0;
    for (char c : s) {
        if (c < '0' || c > '9') return false;
        port = port * 10 + (c - '0');
        if (port > 65535) return false;
    }
    return true;
}
```

19.3 Why small functions are usually better

19.3.1 Readability and naming

Small functions make code read like a specification:

- **Names become documentation.**
- Call sites become higher-level and more intention-revealing.
- The function body has less cognitive load.

```
bool has_permission(UserId u, Action a);
bool is_business_hours(TimePoint t);
bool is_rate_limited(UserId u);

bool allow_request(UserId u, Action a, TimePoint now) {
    if (!has_permission(u, a)) return false;
    if (!is_business_hours(now)) return false;
    if (is_rate_limited(u)) return false;
    return true;
}
```

19.3.2 Testability

A small function with a single purpose is easier to test:

- fewer branches to cover,
- fewer dependencies to mock,
- clearer expected outputs for given inputs.

19.3.3 Maintainability and change isolation

When logic is separated into small functions:

- changes tend to affect fewer lines,
- regressions are easier to localize,
- refactoring is safer.

19.4 When a larger function can be justified

Small functions are the default, but large functions sometimes serve engineering goals better.

19.4.1 Performance-critical tight loops

Sometimes splitting can interfere with performance, but this must be proven. The Guidelines explicitly warn that optimization must be based on measurement and real constraints.
Preferred approach:

- Start with clean, small functions.
- Profile.
- If function call overhead or missed inlining is a demonstrated bottleneck, then optimize.

19.4.2 Highly cohesive algorithms

A function can be longer if it expresses **one cohesive algorithm** and splitting would hide the flow.
Example: a parsing routine may be longer but still "single responsibility" if it is purely parsing and has clear phases.

19.4.3 Locality and invariants

Sometimes keeping steps in one function helps enforce invariants (e.g., acquire resource, update state, release resource) without exposing internal intermediate states.
Rule of thumb: If splitting forces you to expose internal details in the interface, keeping the function larger might be safer.

19.5 Common failure modes of large functions

Large functions tend to accumulate these issues:

- **Too many responsibilities:** validation + parsing + business logic + I/O.
- **Deep nesting:** hard to reason about and easy to break.
- **Long variable lifetimes:** variables survive longer than needed.
- **Hidden side effects:** changes to globals or shared state.

19.6 The performance dimension: inlining and code size

19.6.1 Inlining is not the same as `inline`

In C++, the `inline` keyword primarily affects linkage/ODR rules for functions defined in headers; it may also encourage inlining, but the compiler is free to inline or not inline regardless. The Core Guidelines recommend using `inline` only when a function is *very small and time-critical*.

19.6.2 F.5: Very small and time-critical functions may be `inline`

Rule (F.5): If a function is very small and time-critical, declare it `inline`.
Practical meaning:

- Use `inline` for small, frequently called utilities that are on hot paths.
- Prefer measurement over assumptions.
- Avoid turning large functions into headers “for speed”; that often increases code size and harms instruction cache behavior.

```
// Reasonable candidate: tiny, obvious, likely hot
inline bool is_ascii_digit(unsigned char c) {
    return c >= '0' && c <= '9';
}
```

19.6.3 Small functions and code bloat

Overusing small templated or header-defined functions can increase binary size due to:

- many instantiations (templates),
- duplicated inline bodies across translation units,
- missed opportunities for shared code.

Engineering response:

- Keep hot tiny helpers inline.
- Keep larger or rarely used logic out of headers.
- Prefer explicit instantiation for heavy templates when appropriate.

19.7 Technique: Extracting helpers without losing clarity

19.7.1 Extract “decision” helpers

Long `if` conditions reduce readability. Extract them into named predicates.

```
bool is_request_authorized(const Request& r);
bool is_request_well_formed(const Request& r);

bool handle(const Request& r) {
```

```
    if (!is_request_well_formed(r)) return false;
    if (!is_request_authorized(r))  return false;
    // main work...
    return true;
}
```

19.7.2 Extract phases, not random slices

Split by conceptual phases:

- validate,
- transform,
- compute,
- produce output.

Do not split just to reduce line count if it breaks cohesion.

19.8 Technique: Keep scopes small

Even if the function remains moderately sized, you can keep it simple by limiting scope.

```
int compute_total(const std::vector<int>& v) {
    int total = 0;

    for (int x : v) {
        if (x < 0) continue;
        total += x;
    }

    return total;
}
```

19.9 Practical guidelines: choosing small vs. large

19.9.1 Use small functions when

- the function naturally decomposes into named sub-operations (F.2).
- you want simpler testing and reuse (F.2).
- you can remove nesting and duplication (F.3).
- the function is part of an interface that should read clearly.

19.9.2 Allow larger functions when

- the code is one cohesive algorithm and splitting reduces clarity.
- performance constraints are proven and the split prevents optimization.
- splitting would force you to expose internal details or intermediate states.

19.10 Checklist (adoption-friendly)

Before deciding that a function is “too large” or “too small”, verify:

- **Single responsibility:** Does it do one logical operation (F.2)?
- **Control flow:** Can nesting be reduced (F.3)?
- **Naming:** Can key decisions be extracted into well-named predicates?
- **Scope:** Are variables declared as late as possible and kept local?
- **Duplication:** Is repeated logic extracted into helpers?

- **Performance claims:** Are they supported by measurement?
- **Inlining:** Is `inline` used only for truly tiny, hot functions (F.5)?

Chapter 20

Const Correctness

20.1 Introduction

Const correctness means using `const` (and related qualifiers) to express and enforce a key promise: *this operation does not modify the observable state of the object or data it receives as input*. This turns design intent into a compile-time contract and prevents accidental mutation. In the C++ Core Guidelines, const correctness is emphasized primarily in:

- **Con.1–Con.3:** defaults for immutability, `const` member functions, and passing pointers/references to `const`.
- **F.16–F.17:** parameter passing rules distinguishing "in" vs. "in-out".

This chapter focuses on const correctness specifically as it affects **function interfaces**, **member functions**, and **parameter passing**.

20.2 What `const` communicates in function design

20.2.1 Two essential questions

When designing a function, answer these two questions first:

- Does this function **modify** its arguments or the object it is called on?
- Does it need to **expose mutable access** to internal state?

Use `const` to express the answers directly in the type system.

20.2.2 Const correctness improves

- **Safety:** prevents unintended writes.
- **Readability:** shows intent at the call site.
- **Composability:** const objects can call const member functions; generic code becomes easier.
- **Optimization opportunities:** compilers may exploit known non-modification assumptions.

These benefits are a major reason the Core Guidelines recommend immutability by default.

20.3 Const correctness for input parameters

20.3.1 Default: input parameters are read-only

Guideline (Con.3): By default, pass pointers and references to `const`.
Guideline (F.16): For "in" parameters, pass cheap-to-copy types by value and others by reference to `const`.

```
#include <string>
#include <vector>
#include <string_view>

// cheap to copy -> by value
void set_timeout_ms(int ms);

// read-only, potentially large -> const&
void log_line(const std::string& s);

// non-owning view, still read-only -> by value (small type)
bool starts_with_http(std::string_view s);

// read-only sequence -> span of const elements
#include <span>
int sum(std::span<const int> a) {
    int total = 0;
    for (int x : a) total += x;
    return total;
}
```

20.3.2 Pointers: `T*` vs. `const T*`

A pointer parameter has two common meanings:

- **optional object identity:** `T*` may be null (see also F.60).
- **non-owning access:** it is not ownership transfer (see F.7 and F.22).

If the function does not modify the pointed-to object, use `const T*`.

```
struct Widget { void draw() const; };

// optional, read-only
```

```
void maybe_draw(const Widget* w) {
    if (!w) return;
    w->draw(); // OK: draw() is const
}
```

20.3.3 In-out parameters must be non-const

Guideline (F.17): For “in-out” parameters, pass by reference to non-`const`.

```
#include <string>

// in-out: must be non-const reference
void trim_right(std::string& s) {
    while (!s.empty() && (s.back() == ' ' || s.back() == '\t' || s.back() == '\n'))
        s.pop_back();
}
```

20.4 Const correctness for member functions

20.4.1 Default: make member functions `const`

Guideline (Con.2): By default, make member functions `const`.
Meaning: Mark a member function `const` unless it changes the object’s **observable state**. Inside a `const` member function, `*this` is treated as const, so calling non-const member functions is restricted unless you go through a non-const access path.

```
#include <vector>
#include <cstddef>

class Table {
public:
    // does not modify state -> const
```

```
    std::size_t size() const noexcept { return data_.size(); }

    // provides read-only element access -> const overload
    int at(std::size_t i) const { return data_.at(i); }

    // modifies state -> non-const
    void push(int x) { data_.push_back(x); }

private:
    std::vector<int> data_;
};
```

20.4.2 Provide const and non-const overloads for accessors

A common, safe pattern is:

- a `const` overload that returns `const T&` (read-only),
- a non-const overload that returns `T&` (mutable).

```
#include <vector>
#include <cstddef>

struct Buffer {
    std::vector<unsigned char> bytes;

    unsigned char& operator[](std::size_t i) { return bytes[i]; }
    const unsigned char& operator[](std::size_t i) const { return bytes[i]; }
};
```

This keeps const objects truly read-only while allowing mutation for non-const objects.

20.4.3 Logical constness and `mutable`

Sometimes a function should be `const` even though it updates internal *caches* or bookkeeping that does not affect the object's observable value. In such cases, you may use `mutable` for the cache member. This is a tool, not a default. Use it only when the external behavior is still const.

```
#include <string>
#include <optional>

class ParsedLine {
public:
    explicit ParsedLine(std::string s) : text_(std::move(s)) {}

    // logically const: parsing result is a cache of derived data
    int value() const {
        if (!cached_) cached_ = parse(text_);
        return *cached_;
    }

private:
    static int parse(const std::string&); // may throw; design accordingly
    std::string text_;
    mutable std::optional<int> cached_;   // cache only
};
```

20.5 Const correctness for return types

20.5.1 Avoid returning `const T` by value

Guideline (F.49): Don't return `const T`.

Reason: Returning `const` by value prevents moves from the result and adds no real safety. The caller decides whether to treat the returned value as mutable.

```cpp
#include <string>

// Don't:
const std::string make_bad();

// Do:
std::string make_good();
```

20.5.2 Return `T&` vs. `const T&`

If the caller should be able to modify the returned object, return `T&`. If the caller should only observe it, return `const T&` (typically from a `const` member function). This naturally follows from Con.2 and the const/non-const overload pattern.

20.6 Const correctness with ownership and non-owning access

20.6.1 Do not use smart pointers just to get const correctness

The Core Guidelines recommend taking `T*` or `T&` for non-owning use, and using smart pointers only when you need to express ownership. Const correctness is expressed with `const T*` / `const T&`, not by changing to `shared_ptr<const T>` unless shared ownership is actually required.

```cpp
struct Widget {};

// Avoid: implies shared ownership, not just read-only use
void draw_bad(std::shared_ptr<const Widget> w);

// Prefer: clear non-owning read-only use
void draw_good(const Widget& w);
```

20.7 Common mistakes and how to avoid them

20.7.1 Mistake: forgetting `const` on read-only member functions

This blocks calling the function on const objects and often forces unsafe workarounds. Follow Con.2: make member functions const by default.

20.7.2 Mistake: using `const_cast` to call non-const functions

Using `const_cast` to remove constness is usually a design failure. If the object is truly const, mutating it through a cast is undefined behavior. If it is not truly const, then the correct fix is the interface, not the cast.

20.7.3 Mistake: passing non-const references for input-only arguments

Non-const references imply mutation. For input-only parameters, prefer value or `const&` (Con.3 and F.16).

20.7.4 Mistake: returning references to internals without a const view

If you return a non-const reference to internal data from a const function, you break the const contract. Use const overloads and return `const T&` for const objects.

20.8 Checklist

Before finalizing a function interface, check:

- Are **input-only** parameters passed as value (cheap types) or `const T&` / `const T*` (Con.3, F.16)?

- Are **in-out** parameters passed as `T&` (F.17)?
- Are member functions that do not change observable state marked `const` (Con.2)?
- Do accessors provide const/non-const overloads when they expose references?
- Are you avoiding `return const T` by value (F.49)?
- Are caches implemented with `mutable` only when logically const?

Chapter 21

Overloads

21.1 Introduction

Function overloading allows multiple functions to share the same name while differing in parameter types and/or parameter counts. Used well, overloading makes interfaces **more expressive and easier to use**. Used poorly, it creates ambiguity, surprising behavior, and maintenance hazards.
In the C++ Core Guidelines, the most direct rule in the Functions section is:

- **F.51:** Where there is a choice, prefer default arguments over overloading.

This chapter explains how to design overload sets safely, when to avoid overloads, and how to prevent the most common overload-resolution traps.

21.2 What an overload set should mean

21.2.1 One name, one concept

A high-quality overload set follows a simple principle:

> **All overloads with the same name should represent the same conceptual operation.**

Overloading is most appropriate when the operation is clearly the same but the *types differ*, such as:

- numeric operations across numeric types,
- string input across string-like types (`std::string_view`, etc.),
- const and non-const access (`T&` vs. `const T&`),
- lvalue/rvalue qualifiers for member functions (ref-qualifiers) when semantics differ by value category.

If different overloads implement different semantics, readers cannot infer behavior from the name, and maintenance becomes fragile.

21.3 F.51: Default arguments vs. overloading

21.3.1 F.51: Where there is a choice, prefer default arguments over overloading

Rule (F.51): Prefer default arguments over overloads when the overloads differ only by trailing parameters that can reasonably have defaults.
Reason (Core Guidelines): Default arguments provide alternative interfaces to a single implementation, avoiding drift between multiple overload bodies and reducing code duplication.

21.3.2 Recommended pattern: one implementation + defaults

```
#include <string_view>

void connect(std::string_view host, int port = 443, bool use_tls = true);

void demo() {
    connect("example.com");          // 443, TLS
    connect("example.com", 80);      // 80, TLS
    connect("example.com", 80, false);
}
```

21.3.3 When overloading is better than defaults

Default arguments are not always a good fit. Prefer overloads when:

- You need **different parameter types**, not just missing parameters.
- You want different **preconditions** or **postconditions** that are best expressed as distinct functions.
- You want different **names** because the operations are not truly the same concept.

21.3.4 Critical caveat: default arguments and virtual functions

Default arguments are bound at the call site using the static type information, while virtual dispatch is dynamic. Therefore, default arguments on virtual functions can yield surprising results across base/derived calls. The Core Guidelines explicitly caution about default arguments in such contexts.

```
#include <iostream>

struct Base {
```

```
    virtual void f(int x = 1) { std::cout << "Base " << x << '\n'; }
    virtual ~Base() = default;
};

struct Derived : Base {
    void f(int x = 2) override { std::cout << "Derived " << x << '\n'; }
};

void demo() {
    Derived d;
    Base& b = d;
    b.f();     // calls Derived::f but uses Base's default argument (x = 1)
    d.f();     // calls Derived::f with Derived's default argument (x = 2)
}
```

Guidance:

- Avoid default arguments on virtual functions in public interfaces.
- Prefer overloads or non-virtual wrappers if you need "defaults" with polymorphic behavior.

21.4 Designing safe overload sets

21.4.1 Avoid "almost the same" overloads

Overload resolution follows strict ranking rules (exact match, promotions, conversions, user-defined conversions, etc.). Overloads that are too similar can select unexpectedly, especially with:

- integral promotions (`char`/`short` to `int`),
- signed/unsigned conversions,

- `bool` vs. integer,
- floating-point vs. integer,
- null pointer constants (`0` vs. `nullptr`).

```
void g(int);
void g(bool);

void demo() {
    g(0);      // calls g(int), not g(bool)
    g(false);  // calls g(bool)
}
```

Guidance:

- Prefer **distinct names** when overload candidates are likely to be confused.
- Prefer `nullptr` for pointer/null overload patterns.

21.4.2 Prefer explicitness over clever conversions

If an overload depends on implicit conversions to select the right behavior, callers may be surprised. When correctness matters, make conversions explicit at the call site or in the API design.

```
#include <string_view>

void open_file(std::string_view path);

void demo() {
    open_file("data.txt"); // clear
    // If callers have exotic string types, prefer explicit conversion to string_view.
}
```

21.4.3 Use `const` overloads for accessors

A classic and safe overload pattern is const/non-const access. This follows the Core Guidelines emphasis on const correctness and clear interfaces.

```
#include <vector>
#include <cstddef>

class Buffer {
public:
    unsigned char& operator[](std::size_t i) { return data_[i]; }
    const unsigned char& operator[](std::size_t i) const { return data_[i]; }

private:
    std::vector<unsigned char> data_;
};
```

21.4.4 Use ref-qualifiers when value category changes semantics

For member functions, ref-qualifiers can express different semantics for lvalues vs. rvalues. This can prevent returning references to ephemeral objects and can enable efficient moves.

```
#include <string>
#include <utility>

class Name {
public:
    explicit Name(std::string s) : s_(std::move(s)) {}

    // lvalue object: return reference
    const std::string& str() const & noexcept { return s_; }

    // rvalue object: return by value (move)
```

```
    std::string str() && noexcept { return std::move(s_); }

private:
    std::string s_;
};
```

21.5 Overloading vs. templates

21.5.1 Do not use forwarding references unless you are truly forwarding

A common mistake is to use a universal/forwarding reference to accept “anything” for convenience. This can create overly greedy overloads and surprising matches.
Prefer normal overloads for a small set of intended types. Use templates only when you truly want a generic operation and can constrain it appropriately.

```
#include <string_view>

// Good: explicit overload set
void log(std::string_view);
void log(int);

// Be careful with overly generic templates:
// template<class T> void log(T&&); // can steal calls unexpectedly
```

21.5.2 Constrain templates when they participate in overload resolution

If a template is part of an overload set, constrain it so that it only participates when it is intended to. Modern C++ provides constraints (concepts) to make overload sets safer and clearer.

```
#include <concepts>
#include <string_view>
```

```
void print(std::string_view);

template<std::integral T>
void print(T x);
```

21.6 Common overload-resolution traps

21.6.1 `std::initializer_list` preference

In some contexts, an overload taking `std::initializer_list` can be preferred and may change which overload is selected. Be cautious when adding initializer-list overloads, especially for types that also have other constructors or overloads.

21.6.2 Ambiguity with user-defined conversions

User-defined conversions can make overload selection surprising:

- the compiler may find multiple viable candidates,
- the “best” candidate may not match what the human reader expects,
- small changes can flip which overload is chosen.

Guidance:

- Prefer `explicit` constructors for single-argument conversions unless implicit conversion is essential.
- Prefer named functions for potentially ambiguous conversions.

21.7 Documentation and enforcement practices

21.7.1 Document overload set invariants

For public APIs, document what is common across the overload set:

- shared semantics,
- error behavior,
- complexity guarantees (if relevant),
- ownership/lifetime expectations.

21.7.2 Static analysis hint (Core Guidelines)

The Core Guidelines note that an overload set with a common prefix of parameters often indicates a default-argument opportunity (the motivating case for F.51).

21.8 Checklist

Before introducing or extending an overload set, verify:

- Do all overloads represent **one conceptual operation**?
- If overloads differ only by trailing optional parameters, should this be **default arguments** instead (F.51)?
- Is overload resolution likely to surprise callers due to conversions or promotions?
- Are you using `nullptr` (not `0`) where null is intended?

- Are const and ref-qualifier overloads used to express correct lifetime/value-category semantics?
- If templates participate, are they **constrained** so they do not steal calls?
- Are default arguments avoided on **virtual** functions in public interfaces?

Chapter 22

Lambdas

22.1 Introduction

A **lambda expression** creates an unnamed function object (a closure) that can be passed to algorithms, stored, returned, and executed later. Lambdas are fundamental to modern C++ because they enable:

- local, readable customization of behavior,
- composition with standard algorithms,
- safe capture of values and references,
- expressive callback-style code without global functions.

In the C++ Core Guidelines, lambda use is covered by the Function rules: **F.50–F.54**, and related guidance in the sections on interfaces, lifetime, and concurrency.

22.2 What a lambda is (precise model)

A lambda expression produces a **closure type** (an unnamed, unique class type) with a call operator `operator()`. Captures become data members of the closure object. The compiler generates the closure type; you use it via:

- `auto` (preferred for local use),
- templates (for generic use),
- `std::function` (type-erased callable, when needed).

The C++ language definition specifies the syntax, capture rules, and type properties of closure objects.

```
#include <vector>
#include <algorithm>

void demo(std::vector<int>& v) {
    int bias = 10;

    auto add_bias = [bias](int x) { return x + bias; }; // closure holds a copy of bias

    std::transform(v.begin(), v.end(), v.begin(),
                   [&](int x) { return add_bias(x); });
}
```

22.3 Why lambdas are preferred in modern C++

22.3.1 Locality of logic

When a small custom operation is used at a single point, a lambda keeps the logic near its use site, improving readability.

22.3.2 Better composition with algorithms

Many standard algorithms are designed to accept callables; lambdas make such code concise and intention-revealing.

22.3.3 Avoiding global helper functions

A global helper function is visible everywhere and may become a coupling point. A lambda limits the scope to where it is needed.

22.4 Core Guidelines rules for lambdas (F.50–F.54)

22.4.1 F.50: Use a lambda when a function won't do (to capture local variables, or write a local function)

Rule: Use a lambda when you need a local helper or need to capture local context.

Reason: Lambdas can use local variables without expanding the external interface or introducing extra global state.

```
#include <vector>

void add_offset(std::vector<int>& v, int offset) {
    auto f = [offset](int x) { return x + offset; }; // captures context
    for (int& e : v) e = f(e);
}
```

22.4.2 F.51: Where there is a choice, prefer default arguments over overloading

Although not a lambda-specific rule, it often arises in callback-style APIs: prefer a single callable interface with defaults rather than multiple overloads when only optional parameters differ.

22.4.3 F.52: Prefer capturing by reference in lambdas that will be used locally

Rule: For lambdas that stay local (not stored, returned, or executed asynchronously), capture by reference is usually best.

Reason: It avoids unnecessary copies and typically matches intent for immediate local use.

```
#include <algorithm>
#include <vector>

struct Item { int id; };

void sort_by_key(std::vector<Item>& items, const std::vector<int>& key) {
    std::sort(items.begin(), items.end(),
              [&](const Item& a, const Item& b) { return key[a.id] < key[b.id]; });
}
```

22.4.4 F.53: Avoid capturing by reference in lambdas that will be used non-locally

Rule: If a lambda may outlive the scope in which it was created (stored, returned, or used asynchronously), avoid capturing by reference.

Reason: Reference captures can dangle when the referenced objects go out of scope, causing undefined behavior.

```
#include <functional>

// Don't:
std::function<int()> make_bad() {
    int local = 42;
    return [&] { return local; }; // dangling reference after return
}

// Do:
std::function<int()> make_good() {
    int local = 42;
    return [local] { return local; }; // captured by value: safe
}
```

22.4.5 F.54: When capturing `this` (or data members), do not use `[=]` default capture

Rule: If you capture `this` or members, avoid `[=]` default capture; capture explicitly.

Reason: Default capture can silently capture more than intended and can hide lifetime hazards related to `this`. Explicit capture is clearer and safer.

```
#include <string>
#include <utility>

class X {
public:
    explicit X(std::string s) : name_(std::move(s)) {}

    auto make_task(int k) {
```

```
        // Prefer explicit capture:
        return [this, k] { return name_.size() + static_cast<std::size_t>(k); };
    }

private:
    std::string name_;
};
```

22.5 Capture semantics: value, reference, and lifetime

22.5.1 Capture by value

Capturing by value copies (or moves) the captured object into the closure. This is usually the right choice when:

- the lambda escapes the local scope (stored/returned),
- the lambda runs asynchronously,
- you need a snapshot of a value at capture time.

```
#include <thread>

void run_async(int x) {
    std::thread t([x] { /* safe snapshot of x */ });
    t.join();
}
```

22.5.2 Capture by reference

Capturing by reference stores a reference inside the closure. This is efficient but requires a strict lifetime rule:

All referenced objects must outlive every invocation of the lambda.

This is why reference capture is best restricted to local, immediate use (F.52) and avoided for non-local use (F.53).

22.5.3 Capturing `this` and object lifetime

Capturing `this` stores a pointer to the current object. If the lambda escapes and runs after the object is destroyed, the pointer dangles.

Guidance:

- If the lambda escapes, prefer capturing what you need by value (e.g., `[self = *this]` where appropriate), or use a lifetime-managed mechanism (e.g., shared ownership) only if ownership sharing is truly required.
- Keep escaping lambdas from members carefully designed and documented.

```
#include <string>
#include <utility>

// Snapshot the object (copy) when that is correct:
class Snapshot {
public:
    explicit Snapshot(std::string s) : s_(std::move(s)) {}

    auto task_copy() const {
        return [self = *this] { return self.s_.size(); }; // captures a copy of the object
    }

private:
    std::string s_;
};
```

22.6 Generic lambdas and type deduction

C++ allows generic lambdas using `auto` parameters. The parameters are templates generated by the compiler.

```
#include <vector>
#include <algorithm>

void demo(std::vector<int>& v) {
    std::sort(v.begin(), v.end(), [](const auto& a, const auto& b) { return a < b; });
}
```

Guidance:

- Use generic lambdas when you truly want the lambda to be type-generic.
- Otherwise, prefer explicit parameter types for clearer intent in APIs.

22.7 Mutable lambdas

By default, `operator()` of a lambda is `const`; captured-by-value members are treated as const. Adding `mutable` allows modifying the closure's internal state.

```
#include <cstddef>

auto counter() {
    std::size_t n = 0;
    return [n]() mutable { return ++n; }; // modifies captured copy
}
```

Interpretation:

- `mutable` modifies the *closure state*, not the original captured variables.

- This can be a useful technique for small stateful callables, but avoid it when it obscures intent.

22.8 Lambdas vs. `std::function`

22.8.1 Prefer templates/`auto` for performance and generality

Use `auto` or templates when the callable type can remain known at compile time. This avoids type-erasure overhead and allocations.

```
template<class F>
void for_each_n(int n, F f) {
    for (int i = 0; i < n; ++i) f(i);
}
```

22.8.2 Use `std::function` when you need type erasure

Use `std::function` when you need to store heterogeneous callables behind one runtime type, or when you need an ABI-stable callable type in an interface.

```
#include <functional>

struct Button {
    std::function<void()> on_click; // type-erased callback
};
```

Cost note: `std::function` may allocate and adds an indirection call. Use it deliberately.

22.9 Common mistakes and how to avoid them

22.9.1 Dangling captures

- Returning a lambda that captures locals by reference (violates F.53).
- Storing a lambda capturing `this` without managing object lifetime.

22.9.2 Overusing default capture

Default capture (`[=]` or `[&]`) can capture more than intended and hide dependencies. Prefer explicit capture lists for maintainability. This is especially important when capturing `this` (F.54).

22.9.3 Using lambdas where a named function improves clarity

If a lambda is long, complex, or reused, it is often better to name it (assign to a variable) or extract a function. This aligns with the general function design guidance: keep operations simple and well named.

22.10 Checklist

Before writing a lambda, verify:

- Is a lambda the best tool (local helper or capture needed) (F.50)?
- If it is local-only, is reference capture appropriate (F.52)?
- If it escapes, is capture-by-value used to avoid dangling (F.53)?
- Are captures explicit (especially when capturing `this`) (F.54)?
- Would a named function or named function object clarify intent?
- Is `std::function` truly needed, or can a template/`auto` parameter be used?

Part IV

Classes (C)

Chapter 23

Classes (C.1–C.183)

23.1 Scope and intent

This chapter is a **numbered reference** for the **Classes and class hierarchies** rules (`C.1--C.183`) from the ISO C++ Core Guidelines. It is written to help you *choose the right kind of type*, *define safe and regular semantics*, and *design class hierarchies that remain correct under evolution*.

23.2 A practical mental model

C++ classes typically fall into three design families:

- **Concrete/value types:** copyable, movable, comparable, self-contained, regular behavior.
- **Resource handles:** own something (memory, file, socket), enforce RAII, define move/copy intentionally.

- **Polymorphic hierarchies:** base-class interface + derived implementations; correctness depends on virtual destructor, slicing avoidance, and stable invariants.

The rules in this chapter guide you toward:

- **clear invariants** and **minimal exposure**,
- **safe initialization** and **correct special members**,
- **explicit ownership** and **lifetime safety**,
- **predictable overloading** and **safe unions/variants**.

23.3 Rule directory (C.1–C.183)

For ease of adoption, rules are grouped by the Core Guidelines structure.

23.3.1 C: General class rules (C.1–C.9)

C.1 Group related data into a `struct`/`class`.

C.2 Use `class` when an invariant exists; `struct` when members vary independently.

C.3 Use a class to separate interface from implementation.

C.4 Make a function a member only if it needs representation access.

C.5 Put helper functions in the same namespace as the class.

C.6 Make members `private` by default (encapsulation).

C.7 Do not define a type and a variable of that type in the same statement.

C.8 Prefer `class` over `struct` if any member is non-public.

C.9 Minimize exposure of members; expose operations, not data.

23.3.2 C.concrete: Concrete types (C.10–C.49)

C.10 Prefer concrete types for simple value-like concepts.

C.11 Keep a type's representation consistent with its abstraction.

C.12 Do not make data members `public` unless it is a true aggregate-like type.

C.13 Keep an object's invariants established by constructors.

C.14 Define constructors that establish a valid object state.

C.15 Prefer a minimal, complete set of constructors.

C.16 Prefer initialization over assignment in constructors.

C.17 Prefer `const` correctness for member functions and accessors.

C.18 Use `constexpr` when a type/operation is meant to work at compile time.

C.19 Avoid implicit conversions from single-argument constructors; use `explicit`.

C.20 If a type has an invariant, define operations to preserve it.

C.21 Prefer `=default` for special members when correct.

C.22 Prefer `=delete` to disable unwanted operations.

C.23 Prefer the *rule of zero* (no custom special members if possible).

C.24 If you define/desire one special member, consider the full set (copy/move/dtor).

C.25 Prefer value semantics for value-like types.

C.26 If a type is a value, give it regular/semiregular behavior when appropriate.

C.27 Prefer strong types (distinct types) over type aliases when meaning differs.

C.28 Use `enum class` for scoped enumerations (avoid implicit conversions).

C.29 Use `private` data to enforce invariants and prevent invalid states.

C.30 Define meaningful equality/ordering only when it matches the abstraction.

C.31 Prefer member functions for operations that are part of the abstraction.

C.32 Prefer free functions for symmetric operators and to reduce coupling.

C.33 Provide `swap` when it is meaningful and efficient.

C.34 Make `swap` `noexcept` when it truly cannot throw.

C.35 Prefer to keep member data private and provide coherent operations.

C.36 Avoid `protected` data (exposes representation to derived classes).

C.37 Prefer the interface first in a class definition (readability).

C.38 Prefer composition over inheritance for code reuse.

C.39 Make interfaces small and coherent; avoid “fat” classes.

C.40 Prefer types that are easy to use correctly and hard to misuse.

C.41 Ensure objects are always in a valid usable state after construction.

C.42 Use `noexcept` where non-throwing is required by design.

C.43 Avoid partially formed objects; keep invariants strong.

C.44 Prefer factory functions when construction needs complex validation.

C.45 Prefer a default constructor that leaves objects in a valid state (when meaningful).

C.46 Use RAII to manage resources; pair acquire/release in one type.

C.47 Define destructors to release resources; avoid throwing from destructors.

C.48 Avoid `new`/`delete` in user code; use resource-managing types.

C.49 Prefer standard library types and well-known patterns for basic abstractions.

23.3.3 C.ctor: Constructors, assignments, and destructors (C.50–C.99)

C.50 Use initialization lists to initialize members.

C.51 Use delegating constructors to share initialization logic.

C.52 Use `=default` where the default behavior is correct.

C.53 Use `=delete` to suppress unwanted copy/move operations.

C.54 Provide a default constructor only if it makes sense and yields a valid object.

C.55 Prefer to avoid implicit narrowing and surprising conversions in constructors.

C.56 Prefer `explicit` for single-argument constructors unless implicit conversion is desired.

C.57 Establish class invariants in constructors and maintain them in all operations.

C.58 Prefer member initializer lists over assignment in constructor bodies.

C.59 Do not call virtual functions from constructors/destructors.

C.60 Make destructor non-throwing; handle errors explicitly elsewhere.

C.61 Prefer the rule of zero; otherwise follow rule of five when owning resources.

C.62 Make copy operations correct or disable them for unique ownership.

C.63 Make move operations `noexcept` when correct and important for performance.

C.64 Implement copy assignment carefully (self-assignment safety, strong guarantees as needed).

C.65 Use `swap` or copy-and-swap only when it matches your guarantees and costs.

C.66 Prefer destructors to release resources; avoid manual cleanup calls.

C.67 Ensure move leaves source object in a valid state.

C.68 If you define one of copy/move/dtor, review all special members.

C.69 Prefer `=default` special members when they express intent and stay correct.

C.70 Avoid "clever" special member tricks; prefer simple, correct semantics.

C.71 Avoid ownership confusion: define whether your type is value, unique-owner, or shared-owner.

C.72 Prefer standard smart pointers for ownership transfer/sharing.

C.73 Prefer `unique_ptr` for exclusive ownership; `shared_ptr` only for shared ownership.

C.74 Use `weak_ptr` to break cycles when shared ownership exists.

C.75 Avoid raw owning pointers in class members.

C.76 Provide consistent copy/move/destruction behavior for resource handles.

C.77 Prefer RAII wrappers around OS/library handles.

C.78 Ensure resource acquisition is exception-safe.

C.79 Prefer constructors that cannot fail silently; use exceptions or explicit error channels.

C.80 Prefer factory functions when construction may fail and you want explicit error return.

C.81 Avoid leaving objects half-initialized; keep invariants strong.

C.82 Prefer member initialization order consistent with declaration order.

C.83 Do not use `memcpy`/`memset` on non-trivially-copyable types.

C.84 Avoid explicit destructor calls except in low-level placement-new patterns.

C.85 Make base classes with virtual functions have virtual destructors.

C.86 Prevent slicing: do not copy/assign polymorphic base objects by value.

C.87 Prefer `=delete` for base-class copy operations when slicing would occur.

C.88 If a base class is polymorphic, make it a proper interface/abstract base.

C.89 Use `override` consistently for overriding virtual functions.

C.90 Use `final` where further derivation is not intended.

C.91 Prefer non-virtual interface (NVI) pattern when invariants must be preserved around customization points.

C.92 Avoid public data in base classes.

C.93 Avoid `protected` data in base classes.

C.94 Ensure destruction through base pointers is safe (virtual dtor or non-polymorphic).

C.95 Keep special members aligned with ownership and invariants.

C.96 Prefer explicit and simple semantics for copying/moving resource-owning types.

C.97 Avoid user-defined conversion operators unless truly necessary; keep them explicit.

C.98 Prefer to keep constructors lightweight; move heavy work to factories if needed.

C.99 Avoid surprising side effects in special members.

23.3.4 C.con: Containers and other resource handles (C.100–C.109)

C.100 Follow the STL design when defining a container.

C.101 Give a container value semantics (copy/move behave like values).

C.102 Give a container efficient move operations.

C.103 Provide an `initializer_list` constructor when appropriate.

C.104 Provide a default constructor that creates an empty container.

C.105 Define container invariants and preserve them under all operations.

C.106 Provide iterator and element access consistent with standard conventions.

C.107 Provide `==`/`!=` consistent with container semantics.

C.108 Do not expose representation; expose standard-style operations.

C.109 If a handle has pointer semantics, provide `*` and `->`.

23.3.5 C.lambdas: Function objects and lambdas (C.150–C.170)

C.150 Use lambdas for simple, local function objects.

C.151 Prefer capturing by reference for local use; by value for escaping use.

C.152 Avoid capturing `this` implicitly when lifetime is uncertain; capture explicitly.

C.153 Avoid default capture when it hides dependencies.

C.154 Prefer stateless lambdas when possible.

C.155 Use `std::function` only when type erasure is required.

C.156 Prefer templates/`auto` parameters for generic callables.

C.157 Keep lambda bodies small; extract named functions when they grow.

C.158 Use generic lambdas when you need multiple call signatures.

C.159 Avoid storing reference-capturing lambdas beyond the referenced lifetime.

C.160 Prefer explicit capture lists for maintainability.

C.161 Prefer to avoid heap allocation in hot-path call wrappers.

C.162 Use `mutable` only when closure state must change intentionally.

C.163 Keep callable objects regular when they are stored/passed widely.

C.164 Avoid surprising conversions in callable wrappers.

C.165 Prefer `constexpr` call operators when meaningful.

C.166 Prefer `noexcept` call operators only when non-throwing is guaranteed.

C.167 Avoid capturing heavy objects by value unless necessary; capture views or `shared_ptr` only when ownership is required.

C.168 Avoid hidden dependencies in default captures.

C.169 Prefer clear naming of stored callables (named variables) for readability.

C.170 If you feel like "overloading" lambdas, use a generic lambda.

23.3.6 C.hier: Class hierarchies (OOP) (C.120–C.149)

C.120 Use class hierarchies only when runtime polymorphism is needed.

C.121 If a base class is an interface, make it a pure abstract class.

C.122 Use inheritance to represent substitutability (is-a), not mere code sharing.

C.123 Avoid "fat" base classes; keep interfaces small and stable.

C.124 Provide a virtual destructor for polymorphic base classes.

C.125 Do not use public data in base classes.

C.126 Avoid `protected` data; prefer protected functions.

C.127 Use `override` for overriding functions; catch mistakes.

C.128 Separate interface and implementation responsibilities in the hierarchy.

C.129 Avoid slicing; pass polymorphic objects by reference/pointer.

C.130 Redefine or disable copying for base classes; prefer virtual `clone`.

C.131 Avoid copying polymorphic objects through base interfaces.

C.132 Use `virtual` only for functions meant to be overridden.

C.133 Avoid making destructors virtual unless the class is meant for polymorphic use.

C.134 Keep access levels consistent; do not mix public/protected for the same state.

C.135 Use multiple inheritance to represent multiple distinct interfaces.

C.136 Use multiple inheritance to represent a union of implementation attributes only with care.

C.137 Use virtual bases only to solve real diamond problems; keep them rare.

C.138 Use `using Base::f;` to create an overload set spanning base and derived.

C.139 Use `final` sparingly and intentionally.

C.140 Prefer composition for implementation reuse; keep inheritance for polymorphism.

C.141 Avoid deep hierarchies; they are hard to reason about and evolve.

C.142 Keep base class invariants simple and well documented.

C.143 Do not call virtual functions from constructors/destructors.

C.144 Avoid downcasts; design interfaces to remove the need.

C.145 Prefer `dynamic_cast` only when runtime type checks are truly needed.

C.146 Prefer visitor or type-safe alternatives when downcasting becomes frequent.

C.147 Ensure base class destructors and ownership policies are consistent.

C.148 Avoid exposing representation through base pointers/references.

C.149 Keep polymorphic interfaces stable; avoid adding pure virtuals to widely used bases.

23.3.7 C.over: Overloading and overloaded operators (C.160–C.179)

C.160 Define operators only when they have conventional meaning for the abstraction.

C.161 Use non-member operators for symmetric operations.

C.162 Keep overloaded operators consistent with built-in expectations.

C.163 Provide `operator==` only when equality is well-defined.

C.164 Define ordering only when a strict weak ordering makes sense.

C.165 Do not overload operators to encode unrelated actions.

C.166 Prefer named functions when operator meaning could be ambiguous.

C.167 Keep conversions explicit; avoid surprising implicit conversions.

C.168 Prefer `explicit` conversion operators unless implicit conversion is essential.

C.169 Ensure arithmetic-like operators preserve invariants and expected algebraic properties where applicable.

C.170 Use generic lambdas when you need multiple call signatures.

C.171 Avoid overloading on closely related types when it causes ambiguity.

C.172 Prefer clarity over clever overload tricks.

C.173 Do not overload `&&` and `||` expecting short-circuit semantics.

C.174 Do not overload `,` expecting sequencing like the built-in comma.

C.175 Prefer to avoid user-defined literals unless they strongly improve clarity.

C.176 Keep operator overload sets minimal; avoid API surprise.

C.177 If you overload, follow the standard conventions for semantics and complexity.

C.178 Avoid operators that hide expensive operations.

C.179 Prefer named functions for non-obvious effects (I/O, allocation, locking).

23.3.8 C.union: Unions (C.180–C.183)

C.180 Use unions only when memory saving is necessary.

C.181 Avoid “naked” unions; always track the active member (tagged union).

C.182 Use anonymous unions only as an implementation technique for tagged unions.

C.183 Do not use unions for type punning.

23.4 Representative examples (core patterns you will reuse)

This section provides compact, high-value examples that implement the intent behind many of the rules.

23.4.1 Invariants and encapsulation (C.2, C.6, C.9)

```
#include <stdexcept>

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_; }

    void set(int v) {
        if (v < 0 || v > 100) throw std::out_of_range("Percentage");
        v_ = v;
    }
```

```
private:
    int v_; // invariant: 0..100
};
```

23.4.2 Rule of zero / five with RAII (C.23, C.61, C.75)

```
#include <memory>
#include <cstddef>

class Buffer {
public:
    explicit Buffer(std::size_t n) : n_(n), p_(std::make_unique<unsigned char[]>(n)) {}

    // Rule of zero style: the defaults are correct because members manage themselves.
    Buffer(const Buffer&) = delete;             // unique ownership
    Buffer& operator=(const Buffer&) = delete;

    Buffer(Buffer&&) noexcept = default;
    Buffer& operator=(Buffer&&) noexcept = default;

    unsigned char* data() noexcept { return p_.get(); }
    const unsigned char* data() const noexcept { return p_.get(); }
    std::size_t size() const noexcept { return n_; }

private:
    std::size_t n_{0};
    std::unique_ptr<unsigned char[]> p_;
};
```

23.4.3 Safe polymorphic base (C.121, C.124, C.127, C.130)

```
#include <memory>
```

```
struct Shape {
    virtual ~Shape() = default;                    // safe deletion through base
    virtual void draw() const = 0;

    virtual std::unique_ptr<Shape> clone() const = 0; // copying polymorphic objects
};

struct Circle : Shape {
    void draw() const override { /* ... */ }

    std::unique_ptr<Shape> clone() const override {
        return std::make_unique<Circle>(*this);
    }
};
```

23.4.4 Tagged union alternative (C.181–C.183)

Prefer `std::variant` for a safe tagged union (keeps track of the active alternative).

```
#include <variant>
#include <string>

using Value = std::variant<int, double, std::string>;

double as_double(const Value& v) {
    if (auto p = std::get_if<int>(&v)) return *p;
    if (auto p = std::get_if<double>(&v)) return *p;
    return 0.0;
}
```

23.5 Adoption checklist for real code bases

- Start by enforcing **encapsulation and invariants** (C.2, C.6, C.9).

- Stabilize **special member functions**: apply rule of zero/five and remove raw owning pointers (C.23, C.61, C.75).
- For hierarchies: ensure **virtual destructor**, use **override**, and stop slicing (C.124, C.127, C.129).
- Replace naked unions/type punning with safe alternatives such as **variant** and explicit conversions (C.181–C.183).

Chapter 24

Class Design

24.1 Introduction

Class design in modern C++ is primarily about:

- expressing **meaning** (what the type represents),
- enforcing **invariants** (what must always be true),
- making **ownership and lifetime** explicit,
- providing **safe, predictable semantics** for copying, moving, and destruction,
- keeping interfaces **easy to use correctly** and **hard to misuse**.

The ISO C++ Core Guidelines provide a coherent set of rules for this (notably `C.*` for classes, `R.*` for resource management, and `F.*` for function interfaces).

24.2 Choose the right kind of type

Before writing any member function, decide which category your type belongs to.

24.2.1 Concrete/value type

A **value type** models a value: copying produces an independent value, moving transfers representation efficiently, and equality/order (if provided) follow the abstraction. Prefer concrete types for most domain concepts.

24.2.2 Resource handle (RAII type)

A **handle** owns a resource (memory, file, socket, mutex, OS handle) and releases it in the destructor (RAII). Copying and moving must be designed intentionally (often move-only for unique ownership).

24.2.3 Polymorphic hierarchy

Use inheritance only when you need **runtime polymorphism** and substitutability. A polymorphic base must be designed for safe destruction through base pointers/references and for evolution.

24.3 Invariants and encapsulation

24.3.1 Use `class` for invariants; `struct` for simple aggregates

Guideline intent (C.2):

- Use `struct` when members can vary independently and public data is reasonable.

- Use `class` when you must enforce an invariant.

24.3.2 Keep representation private

Guideline intent (C.6, C.9): Keep data members private by default; expose operations, not data.

```
#include <stdexcept>

class TemperatureC {
public:
    explicit TemperatureC(double c) : c_(c) {
        if (c_ < -273.15) throw std::out_of_range("below absolute zero");
    }

    double value() const noexcept { return c_; }

    void set(double c) {
        if (c < -273.15) throw std::out_of_range("below absolute zero");
        c_ = c;
    }

private:
    double c_; // invariant: c_ >= -273.15
};
```

24.3.3 Do not expose writable base-class data

Guideline intent (C.36 and hierarchy rules): Do not put representation in `protected` data members. It couples derived classes to internals and makes invariants fragile. Prefer protected member functions if extension points are needed.

24.4 Interface vs. implementation

24.4.1 Separate interface and representation

Guideline intent (C.3): Use a class to represent the distinction between interface and implementation.

24.4.2 Use Pimpl when you need ABI stability or heavy implementation hiding

Pimpl trades an extra indirection for reduced compile-time coupling and ABI insulation.

```
#include <memory>
#include <string>

class Widget {
public:
    Widget();
    ~Widget();                    // out-of-line (needs complete Impl)
    Widget(Widget&&) noexcept;
    Widget& operator=(Widget&&) noexcept;

    Widget(const Widget&) = delete;
    Widget& operator=(const Widget&) = delete;

    void set_name(std::string);
    const std::string& name() const;

private:
    struct Impl;
    std::unique_ptr<Impl> p_;     // owning implementation
};
```

24.5 Member vs. non-member functions

24.5.1 Make a function a member only if it needs representation access

Guideline intent (C.4): Prefer non-member functions when possible to reduce coupling and keep the class smaller.

24.5.2 Place helper functions in the same namespace

Guideline intent (C.5): Keep helper functions near the type without making them members.

```
namespace gfx {

class Point {
public:
    Point(int x, int y) : x_(x), y_(y) {}
    int x() const noexcept { return x_; }
    int y() const noexcept { return y_; }

private:
    int x_, y_;
};

// Non-member: symmetric, does not need private access
inline bool operator==(const Point& a, const Point& b) noexcept {
    return a.x() == b.x() && a.y() == b.y();
}

} // namespace gfx
```

24.6 Construction, invariants, and safety

24.6.1 Constructors establish invariants

A well-designed class ensures:

- after construction, the object is usable and valid,
- every public operation preserves the invariant,
- invalid states are unrepresentable or unreachable through the public API.

24.6.2 Prefer initialization over assignment

Use member initializer lists; do not default-construct and then assign inside the constructor body.

```
#include <string>
#include <utility>

class User {
public:
    User(std::string name, int id)
        : name_(std::move(name)), id_(id) {} // initialization, not assignment

private:
    std::string name_;
    int id_;
};
```

24.6.3 Never call virtual functions from constructors or destructors

Virtual dispatch does not behave as many expect during construction/destruction; it can violate invariants and lead to surprising behavior.

24.7 Default operations: rule of zero and rule of five

24.7.1 Prefer the rule of zero

Guideline intent: If your members already manage resources correctly (standard library types, smart pointers, RAII wrappers), do not write custom destructor/copy/move members.

```
#include <string>
#include <vector>

struct Record {
    std::string name;
    std::vector<int> values;
    // Rule of zero: compiler-generated special members are correct
};
```

24.7.2 If you manage a resource, design all special members intentionally

For unique ownership, make the type move-only. For shared ownership, use shared ownership types only when sharing is real and required.

```
#include <memory>
#include <cstddef>

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_; }
    unsigned char* data() noexcept { return p_.get(); }
    const unsigned char* data() const noexcept { return p_.get(); }

private:
    std::size_t n_{0};
    std::unique_ptr<unsigned char[]> p_;
};
```

24.8 Const correctness and access design

24.8.1 Make read-only operations `const`

Const correctness is a core part of robust class design: it protects invariants and improves usability.

24.8.2 Provide const and non-const accessors when returning references

```
#include <vector>
#include <cstddef>

class Bytes {
public:
    unsigned char& operator[](std::size_t i) { return v_[i]; }
```

```
    const unsigned char& operator[](std::size_t i) const { return v_[i]; }

private:
    std::vector<unsigned char> v_;
};
```

24.9 Operator overloading and conversions

24.9.1 Overload only when meaning is conventional

Operator overloading is part of interface design. Only define operators when they match standard expectations for the abstraction; otherwise prefer named functions.

24.9.2 Avoid surprising implicit conversions

Single-argument constructors and conversion operators should be `explicit` unless implicit conversion is clearly intended.

```
class Seconds {
public:
    explicit Seconds(int v) : v_(v) {}
    int count() const noexcept { return v_; }

private:
    int v_;
};
```

24.10 Class hierarchies: safe and evolvable polymorphism

24.10.1 Only use inheritance for polymorphism

Prefer composition for reuse. Use inheritance when the base truly models a stable interface for substitutable derived types.

24.10.2 Polymorphic base checklist

A robust polymorphic base typically has:

- a **virtual destructor** (or a protected non-virtual destructor for non-deletable bases),
- **no public data members**,
- **override** on overrides,
- copying disabled or replaced with a virtual `clone()` if copying polymorphic objects is required,
- non-slicing interfaces (pass by pointer/reference, not by value).

```
#include <memory>

struct Shape {
    virtual ~Shape() = default;
    virtual void draw() const = 0;

    virtual std::unique_ptr<Shape> clone() const = 0;
};

struct Circle : Shape {
    void draw() const override { /* ... */ }

    std::unique_ptr<Shape> clone() const override {
        return std::make_unique<Circle>(*this);
```

```
    }
};
```

24.11 Unions, variants, and representing alternatives

24.11.1 Avoid naked unions

Unions require strict manual tracking of the active member and are easy to misuse. Prefer tagged-union techniques; in modern C++, `std::variant` is the standard tagged union.

```
#include <variant>
#include <string>
#include <vector>

using Value = std::variant<int, double, std::string>;

bool is_text(const Value& v) {
    return std::holds_alternative<std::string>(v);
}
```

24.11.2 Do not use unions for type punning

Using unions for type punning is not a correct or portable C++ technique; use well-defined alternatives (e.g., `std::memcpy` into a suitably typed object) when you truly need bit-level reinterpretation.

24.12 Practical class design workflow

24.12.1 Step 1: Write the invariant

Write, in one sentence, what must always be true. If you cannot, your type is probably not well-formed yet.

24.12.2 Step 2: Decide ownership

Clarify whether the class:

- owns resources (then design move/copy and destructor intentionally),
- or is a pure value over other RAII members (then rule of zero).

Make ownership explicit using RAII and standard resource managers.

24.12.3 Step 3: Design the public API to prevent invalid states

Prefer:

- constructors/factories that validate,
- operations that preserve invariants,
- narrow interfaces (cohesive, small surface area),
- clear parameter passing and const correctness.

24.12.4 Step 4: Minimize coupling

Prefer:

- non-member helper functions when they do not need representation access (C.4, C.5),
- composition instead of inheritance for reuse,
- Pimpl only when you need the trade-off.

24.13 Checklist

Before finalizing a class, verify:

- Does the class have a clear invariant, and is it enforced by constructors and preserved by operations?
- Are data members private by default, with operations exposing meaning instead of representation?
- Are special members correct and minimal (rule of zero when possible; otherwise deliberate copy/move/destruction)?
- Is ownership explicit (no raw owning pointers as members)?
- Are read-only member functions `const`, and are reference-returning accessors const-correct?
- If the type is polymorphic: is destruction safe, slicing avoided, and overriding explicit?
- Are operators and conversions conventional and not surprising?
- Are unions avoided unless truly necessary, and is type punning avoided?

Chapter 25

Invariants

25.1 Introduction

An **invariant** is a condition that must always hold for an object whenever that object is *observable* by its users. In class design, the most important invariant is the **class invariant**: a logical condition on an object's state that:

- is established by a constructor (or factory),
- is assumed by all public member functions,
- is re-established on exit from every public member function (including when exceptions occur).

Invariants are the foundation of **encapsulation**, **exception safety**, and **correctness by construction**. A type that clearly defines and preserves its invariants is easier to use correctly and harder to misuse.

25.2 Why invariants matter in C++ class design

25.2.1 Encapsulation is mainly about preserving invariants

If a class has an invariant, users must not be able to put the object into an invalid state. That is why the Core Guidelines recommend **private data by default** and exposing operations rather than representation.

25.2.2 Invariants enable reliable reasoning

Once an invariant is established:

- member functions can assume it without re-checking everything,
- clients can rely on consistent behavior,
- testing becomes focused: validate the invariant once, then verify operations preserve it.

25.2.3 Invariants and exception safety are linked

If an operation fails (e.g., throws), the class must still obey its invariant. A common design goal is:

- **basic guarantee:** the object remains valid (invariant holds),
- **strong guarantee (when needed):** operation either succeeds fully or has no observable effect.

The invariant is non-negotiable: even the basic guarantee requires it.

25.3 Core Guidelines view of invariants (Classes section)

25.3.1 Use `class` when an invariant exists

Guideline intent (C.2): Use `class` if the type has an invariant; use `struct` if the data members can vary independently.

25.3.2 Keep representation private to enforce the invariant

Guideline intent (C.6, C.9): Make members private by default and minimize exposure of data members; expose operations that preserve meaning.

25.3.3 Invariants are established by constructors

A key note in the Guidelines states:

- the class invariant is established by a constructor,
- and must be re-established upon exit by every member function callable from outside the class.

25.4 Defining an invariant

25.4.1 Write the invariant explicitly

For each class, write a short statement (in a comment, design doc, or contract-like annotation) describing what must always be true.
Examples:

- `0 <= percent <= 100`

- `file_handle is valid or equals invalid_handle`
- `vector is sorted using <`
- `size_ <= capacity_`

25.4.2 Prefer making invalid states unrepresentable

The most robust approach is to choose a representation that cannot represent invalid states (or makes them hard to create). This can be done through:

- private data members,
- strong types (distinct types for distinct meanings),
- validated construction only (constructors/factories),
- limiting mutating operations to those that preserve the invariant.

25.5 Establishing invariants

25.5.1 Constructors must establish the invariant or fail

If the invariant cannot be established, object construction must not succeed. In normal C++ class design, failure is typically reported via exceptions (or by a factory returning an error type).

```
#include <stdexcept>

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
};
```

25.5.2 Use factories when construction needs multi-step validation

If construction is complex, a factory can centralize validation and still guarantee that only valid objects are produced.

```
#include <optional>
#include <string>

class UserName {
public:
    static std::optional<UserName> create(std::string s) {
        if (s.empty()) return std::nullopt;
        if (s.size() > 32) return std::nullopt;
        return UserName(std::move(s));
    }

    const std::string& get() const noexcept { return s_; }

private:
    explicit UserName(std::string s) : s_(std::move(s)) {}
    std::string s_; // invariant: non-empty, length <= 32
};
```

25.6 Preserving invariants in member functions

25.6.1 Public member functions must restore the invariant

Every public operation must ensure that:

- on normal return, the invariant holds,
- if it throws, the object is still valid (invariant holds).

25.6.2 Keep mutating operations narrow and checked

Prefer small mutating functions that do one logical update and validate that the result stays within the invariant.

```
#include <stdexcept>

class BoundedCounter {
public:
    explicit BoundedCounter(int max) : max_(max), value_(0) {
        if (max_ < 0) throw std::out_of_range("max");
    }

    int value() const noexcept { return value_; }

    void increment() {
        if (value_ == max_) throw std::out_of_range("overflow");
        ++value_; // invariant preserved: 0 <= value_ <= max_
    }

private:
    int max_;
    int value_; // invariant: 0 <= value_ <= max_
};
```

25.6.3 Const correctness protects invariants

Mark member functions `const` when they do not modify the observable state. This prevents accidental mutation and makes interfaces clearer.

25.7 Preconditions, postconditions, and invariants

25.7.1 How they relate

- **Preconditions:** what must be true before a function is called.
- **Postconditions:** what the function guarantees on return.
- **Class invariant:** what must be true for the object at all observable times.

Most member functions can assume the class invariant holds on entry, and must ensure it holds again on exit.

25.7.2 Document requirements clearly

The Core Guidelines encourage stating requirements either formally (e.g., `Expects(...)`) or informally via clear English comments.

```
#include <vector>
#include <algorithm>

// Invariant for SortedInts: v_ is always sorted ascending.
class SortedInts {
public:
    SortedInts() = default;

    void insert(int x) {
```

```
        auto it = std::lower_bound(v_.begin(), v_.end(), x);
        v_.insert(it, x);
        // invariant preserved: v_ remains sorted
    }

    const std::vector<int>& data() const noexcept { return v_; }

private:
    std::vector<int> v_;
};
```

25.8 Techniques for checking invariants

25.8.1 Runtime checking in debug builds

Invariants can be checked with assertions in debug builds:

- check at the end of constructors,
- check at the end of mutating public functions,
- optionally check on entry to public functions during development.

```
#include <cassert>
#include <vector>
#include <algorithm>

class SortedInts {
public:
    void insert(int x) {
        check_invariant();
        auto it = std::lower_bound(v_.begin(), v_.end(), x);
        v_.insert(it, x);
```

```
        check_invariant();
    }

private:
    void check_invariant() const {
        assert(std::is_sorted(v_.begin(), v_.end()));
    }

    std::vector<int> v_;
};
```

25.8.2 RAII guard to check on exit even with exceptions

A small RAII guard can help ensure the invariant is validated on function exit in debug builds. This pattern is compatible with exception-based error handling because destructors run during stack unwinding.

```
#include <cassert>

template<class T>
class InvariantGuard {
public:
    explicit InvariantGuard(const T& obj) : p_(&obj) {
        p_->check_invariant();
    }
    ~InvariantGuard() noexcept {
        p_->check_invariant();
    }
private:
    const T* p_;
};

class Range {
```

```
public:
    Range(int lo, int hi) : lo_(lo), hi_(hi) {
        check_invariant();
    }

    void shift(int d) {
        InvariantGuard<Range> g(*this);
        lo_ += d;
        hi_ += d;
    }

private:
    void check_invariant() const {
        assert(lo_ <= hi_);
    }

    int lo_;
    int hi_; // invariant: lo_ <= hi_
};
```

25.9 Invariants and class hierarchies

25.9.1 Base-class invariants must be simple and stable

In hierarchies, derived classes depend on base class rules. Therefore:

- base-class invariants should be minimal and clearly documented,
- base classes should avoid exposing representation through `protected` data,
- customization points should be protected virtual functions (or non-virtual interface pattern) rather than direct data access.

25.9.2 Do not call virtual functions from constructors or destructors

Doing so can violate invariants because the dynamic type is not the most-derived type during construction/destruction, and virtual dispatch rules differ in these phases.

25.10 Invariants and ownership

25.10.1 Ownership rules are invariants

If a class owns a resource, a core invariant is typically:

- the resource is either valid and owned,
- or it is in a well-defined “empty” state (after move),
- and it will be released exactly once.

25.10.2 Move operations must preserve validity

After a move, both objects must remain valid (though the moved-from object may be in a restricted but valid state). This is an invariant requirement for move-enabled types and is relied upon widely by generic code.

```
#include <memory>
#include <cstddef>

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_; }
    const unsigned char* data() const noexcept { return p_.get(); }

private:
    std::size_t n_{0};
    std::unique_ptr<unsigned char[]> p_;
    // invariant: if n_ > 0 then p_ != nullptr (for objects created by this constructor)
    // after move: moved-from object remains valid (n_ and p_ in a valid state per
    ↪  defaults)
};
```

25.11 Common invariant failures and fixes

25.11.1 Public data members in an invariant-bearing type

Problem: users can violate the invariant. **Fix:** make data private and expose operations that maintain validity.

25.11.2 Two-phase initialization

Problem: object exists in a partially valid state. **Fix:** establish the invariant in the constructor or use a factory that returns only valid objects.

25.11.3 Complex functions that modify many fields

Problem: invariant breaks in the middle and is hard to restore on all paths. **Fix:** refactor into smaller operations, use local temporaries, and commit changes only once the final state is valid (helps achieve the strong guarantee).

25.12 Checklist

For each class:

- Is the class invariant stated clearly?
- Do constructors/factories either establish the invariant or fail?
- Are members private by default, so users cannot violate the invariant directly?
- Does every public mutating operation preserve the invariant on success and on failure?
- Is const correctness used to prevent accidental mutation?
- Are debugging checks available (assertions, invariant checker) without affecting release performance?
- In hierarchies: is the base invariant minimal, stable, and independent of derived internals?
- Are ownership/lifetime rules explicit and preserved under move/copy?

Chapter 26

Constructors

26.1 Introduction

A constructor defines how an object begins its life. In modern C++, a well-designed constructor:

- **establishes the class invariant** (the object becomes valid and usable),
- **makes ownership and lifetime clear** (especially for resource-managing types),
- **prevents invalid states** from being observable,
- **avoids surprises** such as unintended implicit conversions or partial initialization.

The C++ Core Guidelines treat constructors as the primary mechanism for creating *valid objects by construction*. In particular, the class invariant should be established by a constructor and preserved by all public operations.

26.2 Primary rule: constructors establish invariants

26.2.1 Class invariant and validity

A **class invariant** is a condition that must always hold for an object whenever it is observable by its users. A constructor must either:

- establish the invariant (construction succeeds), or
- fail such that the object is not created.

```
#include <stdexcept>

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 <= v_ <= 100
};
```

26.2.2 Avoid two-phase initialization

Two-phase initialization (default-construct, then call `init()`) exposes partially valid states. Prefer:

- constructors that fully initialize, or
- factory functions when construction needs validation or can fail without exceptions.

26.3 Prefer initialization over assignment

26.3.1 Member initializer lists

Use the member initializer list to initialize data members directly. This is not just a style preference: it is required for `const` members, references, and many RAII types, and it avoids redundant work.

```
#include <string>
#include <utility>

class User {
public:
    User(std::string name, int id)
        : name_(std::move(name)), id_(id) {} // initialization, not assignment

private:
    std::string name_;
    int id_;
};
```

26.3.2 Initialization order is declaration order

Data members are initialized in the order they are declared in the class, not the order written in the initializer list. Write the initializer list in declaration order to avoid confusion and warnings.

```
class X {
public:
    X(int a, int b)
        : a_(a), b_(b) {} // matches declaration order: a_ then b_

private:
```

```
    int a_;
    int b_;
};
```

26.4 Choosing the right constructor set

26.4.1 Keep constructors minimal and coherent

Provide the smallest set of constructors that makes the type easy to use correctly. Too many overloads often indicate missing abstractions, unclear invariants, or inconsistent semantics.

26.4.2 Prefer non-throwing default construction only when meaningful

A default constructor should create a valid object in a sensible default state. If no meaningful default exists, do not provide one; require construction with the necessary arguments.

```
class Port {
public:
    explicit Port(int p) : p_(p) {
        if (p_ < 1 || p_ > 65535) throw std::out_of_range("Port");
    }
    int value() const noexcept { return p_; }

private:
    int p_;
};
```

26.5 Avoid unintended implicit conversions

26.5.1 Use `explicit` for single-argument constructors

Single-argument constructors can define implicit conversions. Usually, that is not wanted. Prefer `explicit` unless implicit conversion is a deliberate part of the type's design.

```
class Seconds {
public:
    explicit Seconds(int v) : v_(v) {}
    int count() const noexcept { return v_; }
private:
    int v_;
};
```

26.5.2 `explicit(bool)` (conditional explicitness)

C++ supports `explicit(bool)` to make a constructor explicit only under specific conditions. Use this sparingly and only when it improves correctness and generic usability.

```
#include <type_traits>

template<class T>
class Box {
public:
    // Implicit only if T is implicitly convertible from U
    template<class U>
    explicit(!std::is_convertible_v<U, T>)
    Box(U&& u) : value_(static_cast<T>(u)) {}

private:
    T value_;
};
```

26.6 Delegating and inheriting constructors

26.6.1 Delegating constructors

Delegating constructors allow one constructor to call another constructor of the same class to avoid duplication and keep initialization consistent.

```
#include <string>
#include <utility>

class Path {
public:
    Path() : Path(".") {} // delegate to primary constructor

    explicit Path(std::string s)
        : text_(std::move(s)) {
        // establish invariant here if needed
    }

    const std::string& str() const noexcept { return text_; }

private:
    std::string text_;
};
```

26.6.2 Inheriting constructors

A derived class can inherit constructors from a base using `using Base::Base;`. Use this when the derived class does not require additional initialization beyond the base.

```
#include <string>
#include <utility>
```

```
class Base {
public:
    explicit Base(std::string s) : s_(std::move(s)) {}
protected:
    std::string s_;
};

class Derived : public Base {
public:
    using Base::Base; // inherit Base constructors
};
```

26.7 Factories for “virtual behavior” during initialization

26.7.1 Do not rely on virtual dispatch in constructors

Calling virtual functions from constructors (or destructors) does not provide the usual dynamic dispatch behavior and often breaks invariants.
If you need initialization that depends on the most-derived type, use a factory function (or another post-construction mechanism that operates on a fully constructed object).

```
#include <memory>

struct Shape {
    virtual ~Shape() = default;
    virtual void draw() const = 0;

    // Factory: virtual behavior occurs after construction of the concrete object
    static std::unique_ptr<Shape> make_circle();
};
```

```
struct Circle : Shape {
    void draw() const override {}
};

std::unique_ptr<Shape> Shape::make_circle() {
    return std::make_unique<Circle>();
}
```

26.8 Exception safety in constructors

26.8.1 If construction fails, no object exists

If a constructor throws, the object is not considered constructed. Already-constructed subobjects (base classes and members) are destroyed as stack unwinding proceeds.

26.8.2 RAII members simplify correctness

Prefer RAII members (`std::string`, `std::vector`, smart pointers, lock guards). Then, if an exception occurs during construction, cleanup happens automatically. This supports the rule of zero and reduces error-prone manual cleanup.

```
#include <memory>
#include <vector>

class Image {
public:
    explicit Image(std::size_t n)
        : pixels_(std::make_unique<unsigned char[]>(n)), meta_(n) {
        // if meta_ construction throws, pixels_ is released automatically
    }
```

```
private:
    std::unique_ptr<unsigned char[]> pixels_;
    std::vector<int> meta_;
};
```

26.9 Copy, move, and special member constructors

26.9.1 Rule of zero (preferred)

If your class has no custom resource management beyond RAII members, do not write copy/move constructors or destructors. Let the compiler generate them.

```
#include <string>
#include <vector>

struct Record {
    std::string name;
    std::vector<int> values;
    // compiler-generated copy/move are correct
};
```

26.9.2 Resource owners: design copying and moving intentionally

If your type owns a resource, decide whether it is:

- **move-only** (unique ownership), or
- **copyable** (deep copy or shared ownership by explicit design).

Use `=delete`, `=default`, and `noexcept` intentionally.

```
#include <memory>
#include <cstddef>
```

```
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;

private:
    std::size_t n_{0};
    std::unique_ptr<unsigned char[]> p_;
};
```

26.10 Performance and correctness notes

26.10.1 Avoid heavy work in constructors when it harms usability

Constructors should primarily establish invariants. If construction requires expensive operations, consider:

- a factory that can report failure explicitly,
- lazy initialization (with careful thread-safety),
- separating “create” from “load” when it improves clarity.

Do not compromise invariant establishment, but keep interfaces usable.

26.10.2 Prefer `noexcept` where non-throwing is part of the design

Use `noexcept` only when you can guarantee the constructor will not throw. Marking a throwing constructor `noexcept` is incorrect: if it throws, the program calls `std::terminate`.

26.10.3 `constexpr` constructors

C++ allows constructors to be usable in constant evaluation when constraints are met. Use `constexpr` to express that the type is intended to work at compile time, when meaningful for the abstraction.

```
class Vec2 {
public:
    constexpr Vec2(int x, int y) : x_(x), y_(y) {}
    constexpr int x() const noexcept { return x_; }
    constexpr int y() const noexcept { return y_; }
private:
    int x_;
    int y_;
};
```

26.11 Common constructor pitfalls

26.11.1 Leaving members uninitialized

Always initialize all members (directly or indirectly). Prefer member initializers and RAII types.

26.11.2 Narrowing conversions

Avoid constructors that silently narrow values (e.g., `double` to `int`). Prefer explicit conversions and validation as part of invariant establishment.

26.11.3 Ambiguity with `initializer_list`

Overloads involving `std::initializer_list` can be preferred by overload resolution and change which constructor is called. Design initializer-list constructors deliberately and test call sites for ambiguity.

26.11.4 Calling virtual functions in constructors/destructors

Do not do this as a design technique. Use factories or post-construction hooks that operate on a fully constructed object.

26.12 Checklist

Before finalizing constructors for a class, verify:

- Does every constructor either establish the invariant or fail (no partially valid objects)?
- Are all members initialized using the member initializer list (not assigned later)?
- Is the initialization order aligned with declaration order?
- Are single-argument constructors `explicit` unless implicit conversion is intended?
- Is a default constructor provided only when it yields a meaningful valid state?
- If the type owns resources, are copy/move constructors and assignments intentionally designed (rule of zero or deliberate rule of five)?

- Are factories used when you need validation, complex creation, or “virtual behavior” during initialization?
- Is `noexcept` used only when you can truly guarantee non-throwing behavior?

Chapter 27

Destructors

27.1 Introduction

A destructor defines how an object ends its lifetime. In modern C++, destructors are primarily a tool for **RAII**: a class that owns a resource releases it automatically when the object is destroyed. This is the foundation of exception-safe cleanup and deterministic resource management.
A well-designed destructor:

- releases owned resources exactly once,
- never throws,
- works correctly during stack unwinding (exceptions),
- interacts correctly with inheritance and deletion through base pointers,
- leaves the program in a safe state even when cleanup is triggered by failure paths.

27.2 Destructor fundamentals (what the language guarantees)

27.2.1 When destructors run

In C++, destructors are invoked:

- at end of scope for automatic objects,
- when `delete` is applied to a pointer to an object,
- when a temporary ends its full-expression lifetime,
- when a container or owning object destroys its elements/subobjects,
- during stack unwinding when an exception leaves a scope.

27.2.2 Destruction order

For a most-derived object:

- the derived destructor body runs first,
- then members are destroyed in reverse order of their declaration,
- then base-class destructors run.

27.2.3 A destructor is implicitly `noexcept`

By the C++ language rules, a destructor is implicitly non-throwing (effectively `noexcept(true)`) unless explicitly specified otherwise. If an exception exits a destructor during stack unwinding, the program calls `std::terminate`. This is why the Core Guidelines state: **destructors must not throw**.

27.3 Core Guidelines: the destructor rules

27.3.1 Do not throw from destructors

Guideline intent (C.60): A destructor must not fail. Do not let exceptions escape.
Reason:

- Destructors often run during exception unwinding.
- Throwing during unwinding can cause immediate termination.
- Even outside unwinding, exceptions from destructors create fragile control flow and make code hard to reason about.

27.3.2 Use RAII and prefer the rule of zero

Guideline intent (C.23): Prefer the *rule of zero*. If members manage resources (standard containers, smart pointers), you should not define a destructor at all.

```
#include <string>
#include <vector>

struct Record {
    std::string name;
    std::vector<int> values;
    // No destructor needed. Rule of zero.
};
```

27.3.3 If you define a destructor, review all special members

Defining a destructor can affect implicit generation of move operations. If you write a destructor, you must consider copy/move constructors and assignments (rule of five mindset).

27.4 Designing a correct destructor

27.4.1 Destructor responsibilities

A destructor should typically do only:

- release owned resources,
- perform minimal, non-throwing cleanup.

It should not:

- perform complex business logic,
- throw exceptions,
- call virtual functions expecting derived overrides.

27.4.2 Keep cleanup non-throwing

Prefer resource types whose cleanup is non-throwing by design:

- `std::unique_ptr` and standard containers,
- RAII wrappers for OS handles that never throw in `close`/`release`.

If cleanup can fail (e.g., flush-to-disk), do not perform the failing action in the destructor as the primary error-reporting path. Provide an explicit `close()` / `commit()` that reports errors, and let the destructor do a best-effort non-throwing cleanup.

```
#include <cstdio>
#include <stdexcept>

class FileWriter {
```

```
public:
    explicit FileWriter(const char* path) {
        f_ = std::fopen(path, "wb");
        if (!f_) throw std::runtime_error("open failed");
    }

    // explicit error reporting
    void close() {
        if (!f_) return;
        if (std::fclose(f_) != 0) {
            f_ = nullptr;
            throw std::runtime_error("close failed");
        }
        f_ = nullptr;
    }

    ~FileWriter() noexcept {
        // Best-effort: never throw.
        if (f_) std::fclose(f_);
    }

private:
    std::FILE* f_{nullptr};
};
```

27.5 Destructors and ownership

27.5.1 RAII makes ownership explicit

If a class owns a resource, its destructor defines the release policy. The Core Guidelines strongly recommend avoiding raw owning pointers and using resource-managing types.

27.5.2 Prefer `unique_ptr` for exclusive ownership

`std::unique_ptr` expresses unique ownership and ensures deletion in the destructor of the owning object. This often eliminates the need for a user-written destructor.

```
#include <memory>

struct Impl {};

class Widget {
public:
    Widget() : p_(std::make_unique<Impl>()) {}
    // No destructor needed: unique_ptr cleans up.
private:
    std::unique_ptr<Impl> p_;
};
```

27.5.3 "After move" validity is part of the invariant

Move operations must leave moved-from objects valid (though possibly empty). Destructors must therefore tolerate the moved-from state without special cases.

27.6 Destructors and inheritance

27.6.1 Virtual destructor for polymorphic base classes

Guideline intent (C.124, C.85): A class with virtual functions intended for polymorphic use must have a virtual destructor, so deleting through a base pointer is safe.

```
struct Base {
    virtual ~Base() = default; // required for polymorphic deletion
    virtual void f() = 0;
```

```
};

struct Derived : Base {
    ~Derived() override = default;
    void f() override {}
};
```

27.6.2 When a base class should *not* be deletable

Sometimes a base is not meant to be deleted through a base pointer. In such designs, the base destructor can be `protected` and non-virtual, preventing deletion through `Base*`. This is an advanced technique and must match the ownership model.

```
struct NonDeletableBase {
    virtual void f() = 0;

protected:
    ~NonDeletableBase() = default; // cannot delete via NonDeletableBase*
};
```

27.6.3 Do not call virtual functions from destructors

During destruction, the dynamic type is effectively stepping down the hierarchy. Virtual calls from a base destructor do not dispatch to already-destroyed derived parts and can violate invariants.

27.7 Custom destructors and the special member set

27.7.1 Destructor presence affects implicit move generation

If you declare a destructor (even `=default`), the rules for implicit move operations can change. Therefore, when you write a destructor, re-evaluate whether you also need to declare move/copy operations explicitly.

27.7.2 Prefer `=default` when you need a declared destructor for form

Sometimes you want a declared destructor for readability, to make a type polymorphic, or to place it out-of-line. If the default behavior is correct, use `=default`.

```
struct Poly {
    virtual ~Poly() = default; // defaulted and virtual
};
```

27.8 Best practices for “cleanup that can fail”

27.8.1 Provide explicit operations for error reporting

If release can fail in a way that matters (flush, commit, close with error), provide:

- an explicit `close()` / `commit()` that can report failure,
- a destructor that performs best-effort cleanup and never throws.

27.8.2 Never rely on destructors for business-critical error reporting

Because destructors cannot reliably report failures (especially during unwinding), they must not be the only mechanism that detects and reports critical cleanup errors.

27.9 Common destructor mistakes

27.9.1 Throwing exceptions

This is the most severe error: it can terminate the program during unwinding.

27.9.2 Manual `delete` on owning raw pointers

This is error-prone and often leads to leaks, double deletion, and unclear ownership. Use smart pointers or RAII wrappers.

27.9.3 Polymorphic base without virtual destructor

Deleting a derived object through a base pointer with a non-virtual base destructor is undefined behavior.

27.9.4 Doing too much work in destructors

Destructors should be simple and reliable. Put complex logic into explicit functions with error handling.

27.10 Checklist

Before finalizing a destructor design, verify:

- Does the type actually need a destructor, or can it follow the rule of zero?
- If a destructor exists, is it guaranteed not to throw (directly or indirectly)?
- Does the destructor release owned resources exactly once and tolerate moved-from states?

- If the class is polymorphic, is the destructor virtual (or otherwise designed to prevent base deletion)?
- Are any operations that can fail (flush/commit/close) exposed as explicit functions rather than destructor-only behavior?
- If you declared a destructor, did you re-check copy/move operations and invariants (rule of five mindset)?

Chapter 28

Rule of 0 / 3 / 5

28.1 Introduction

The **Rule of 0**, **Rule of 3**, and **Rule of 5** are practical design rules for C++ classes that manage resources or define custom object lifetime behavior.
They address one question:

> **If I define one special member function, which other special member functions must I consider?**

In modern C++, the C++ Core Guidelines recommend designing types so that you usually follow the **Rule of 0**, and only define special member functions when your type truly needs custom resource management or special semantics.

28.2 Special member functions (the set you must understand)

A class can have the following **special member functions** (SMFs):

- destructor: `T()`

- copy constructor: `T(const T&)`
- copy assignment: `T& operator=(const T&)`
- move constructor: `T(T&&)`
- move assignment: `T& operator=(T&&)`

These functions are often **implicitly generated** by the compiler depending on your class members and what you declare. Modern C++ also supports explicitly controlling them with `=default` and `=delete`.

28.3 Rule of 0

28.3.1 Definition

Rule of 0: If your class does not directly manage a resource, define **none** of the special member functions. Let the compiler generate them.

28.3.2 Why the Core Guidelines prefer it

The Core Guidelines emphasize RAII and using standard library types and smart pointers for resource management. If your members already manage their own resources correctly, your class becomes automatically correct for copying/moving/destruction, and you avoid subtle lifetime bugs.

28.3.3 Example: value type with RAII members

```
#include <string>
#include <vector>
```

```
class Person {
public:
    Person(std::string name, int age)
        : name_(std::move(name)), age_(age) {}

    const std::string& name() const noexcept { return name_; }
    int age() const noexcept { return age_; }

private:
    std::string name_;
    int age_{0};
    std::vector<int> history_;
    // Rule of 0: no destructor/copy/move needed
};
```

28.3.4 What makes Rule of 0 possible

Rule of 0 becomes natural when you:

- store resources in standard types (`std::vector`, `std::string`, etc.),
- use smart pointers (`std::unique_ptr`, `std::shared_ptr`) instead of owning raw pointers,
- wrap OS handles in RAII types so cleanup is automatic.

28.4 Rule of 3

28.4.1 Definition (historical and still relevant)

Rule of 3 (pre-C++11): If you define any of:

- destructor,

- copy constructor,
- copy assignment,

you usually need to define all three. This rule arose because user-defined ownership (e.g., raw `new`/`delete`) requires consistent deep-copy and cleanup behavior.

28.4.2 Why it still matters today

Even in modern C++:

- legacy code may still use manual resource management,
- some low-level wrappers may intentionally manage resources without standard helpers,
- you may need custom copy semantics for performance or policy reasons.

When you do, you must ensure copying and destruction are consistent and safe.

28.4.3 Example: deep-copying resource owner (Rule of 3)

```
#include <algorithm>
#include <cstddef>
#include <stdexcept>

class IntBuffer {
public:
    explicit IntBuffer(std::size_t n)
        : n_(n), p_(n ? new int[n] : nullptr) {}

    ~IntBuffer() noexcept {
        delete[] p_;
    }
```

```
    IntBuffer(const IntBuffer& other)
        : n_(other.n_), p_(other.n_ ? new int[other.n_] : nullptr) {
        std::copy(other.p_, other.p_ + n_, p_);
    }

    IntBuffer& operator=(const IntBuffer& other) {
        if (this == &other) return *this;

        int* newp = other.n_ ? new int[other.n_] : nullptr;
        std::copy(other.p_, other.p_ + other.n_, newp);

        delete[] p_;
        p_ = newp;
        n_ = other.n_;
        return *this;
    }

    std::size_t size() const noexcept { return n_; }
    const int* data() const noexcept { return p_; }

private:
    std::size_t n_{0};
    int* p_{nullptr};
};
```

Note: This example is correct but intentionally demonstrates manual management. In modern code, prefer `std::vector<int>` or `std::unique_ptr<int[]>` so you can use the Rule of 0.

28.5 Rule of 5

28.5.1 Definition

Rule of 5 (C++11 and later): If you declare any of:

- destructor,
- copy constructor,
- copy assignment,
- move constructor,
- move assignment,

you likely need to declare all five (or explicitly default/delete those that should not exist).

28.5.2 Why it exists

Move operations were added to support efficient transfer of resources. If you manage a resource and define custom copy/destruction, you must decide what moving means and implement it correctly.

28.5.3 Move-only types (unique ownership)

For unique ownership, copying is typically disabled and moving is enabled.

```
#include <memory>
#include <cstddef>

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;

    std::size_t size() const noexcept { return n_; }

private:
    std::size_t n_{0};
    std::unique_ptr<unsigned char[]> p_;
};
```

This is a modern, safe design:

- destructor is defaulted (resource release handled by `unique_ptr`),
- copying is deleted (no accidental sharing),
- moving is defaulted and marked `noexcept` as provided by the member types.

28.5.4 Copyable resource owners (deep copy + move)

If the abstraction requires copying, implement deep copy and also define moves (often `noexcept`) for performance.

```
#include <algorithm>
#include <cstddef>
#include <utility>

class IntBuffer2 {
```

```
public:
    explicit IntBuffer2(std::size_t n)
        : n_(n), p_(n ? new int[n] : nullptr) {}

    ~IntBuffer2() noexcept { delete[] p_; }

    IntBuffer2(const IntBuffer2& other)
        : n_(other.n_), p_(other.n_ ? new int[other.n_] : nullptr) {
        std::copy(other.p_, other.p_ + n_, p_);
    }

    IntBuffer2& operator=(const IntBuffer2& other) {
        if (this == &other) return *this;

        int* newp = other.n_ ? new int[other.n_] : nullptr;
        std::copy(other.p_, other.p_ + other.n_, newp);

        delete[] p_;
        p_ = newp;
        n_ = other.n_;
        return *this;
    }

    IntBuffer2(IntBuffer2&& other) noexcept
        : n_(other.n_), p_(other.p_) {
        other.n_ = 0;
        other.p_ = nullptr;
    }

    IntBuffer2& operator=(IntBuffer2&& other) noexcept {
        if (this == &other) return *this;
        delete[] p_;
```

```
        n_ = other.n_;
        p_ = other.p_;
        other.n_ = 0;
        other.p_ = nullptr;
        return *this;
    }

    std::size_t size() const noexcept { return n_; }

private:
    std::size_t n_{0};
    int* p_{nullptr};
};
```

28.6 Rule of 0 vs. Rule of 5: how to choose

28.6.1 Prefer Rule of 0 whenever possible

If you can represent your resources withII purely using RAII members, do it. This improves correctness, testability, and maintainability.

28.6.2 When Rule of 5 is justified

Write special members only when:

- you directly manage a resource that cannot be expressed via standard RAII members, or
- you need special copy/move semantics that differ from member-wise semantics, or
- you must enforce a specific invariant across copying/moving not expressible otherwise.

28.7 Important modern details (that cause real bugs)

28.7.1 A user-declared destructor can inhibit implicit moves

In modern C++, declaring a destructor (even `=default`) can change which move operations are implicitly generated. Therefore, declaring any special member requires you to re-evaluate the others.

28.7.2 Use `=default` and `=delete` to express intent

Modern C++ encourages explicitly defaulting or deleting special members to document the intended semantics and to keep overload sets correct.

```
class NonCopyable {
public:
    NonCopyable() = default;
    ~NonCopyable() = default;

    NonCopyable(const NonCopyable&) = delete;
    NonCopyable& operator=(const NonCopyable&) = delete;

    NonCopyable(NonCopyable&&) noexcept = default;
    NonCopyable& operator=(NonCopyable&&) noexcept = default;
};
```

28.7.3 Move operations should often be `noexcept`

Many standard library optimizations (e.g., moving elements during reallocation) assume move operations are non-throwing. When correct, prefer `noexcept` moves for performance and stronger guarantees.

28.7.4 Destructors must not throw

A throwing destructor can terminate the program during exception unwinding. Design cleanup so errors are reported through explicit operations, not destructors.

28.8 A practical decision checklist

- Can the class be built from RAII members only?
  - Yes → follow **Rule of 0**.
  - No → continue.
- Does the abstraction require copying?
  - No → make it move-only (delete copy, default/define move).
  - Yes → implement correct deep copy and also implement move (Rule of 5).
- If you declare any special member, did you explicitly decide the behavior of all others?
- Are invariants preserved after copy and after move (moved-from object valid)?
- Are moves `noexcept` when correct?
- Do destructors never throw?

Chapter 29

Copy and Move Semantics

29.1 Introduction

Copy and move semantics define what it means to duplicate or transfer an object's state. In modern C++, correct copy/move design is essential for:

- **value types** (predictable behavior like `int`, `std::string`),
- **resource-managing types** (ownership, RAII, exception safety),
- **performance** (efficient transfers instead of deep copies),
- **generic programming** (containers and algorithms rely on well-defined semantics).

The Core Guidelines emphasize that a type should be either:

- a **value type** with regular semantics,
- a **unique owner** (move-only),
- or a **shared owner** (explicit shared semantics), and that the chosen model must be consistent and visible in the interface.

29.2 The five special member functions

Copy and move behavior is defined by the special member functions (SMFs).

- destructor: `T()`
- copy constructor: `T(const T&)`
- copy assignment: `T& operator=(const T&)`
- move constructor: `T(T&&)`
- move assignment: `T& operator=(T&&)`

A class may get these implicitly from the compiler or you may:

- request the default behavior with `=default`,
- forbid an operation with `=delete`,
- implement a custom operation with a user-defined body.

29.3 Design decision: value, unique ownership, or shared ownership

29.3.1 Value semantics (copy means independent value)

A value type behaves like a mathematical value: copying creates an independent object with the same meaning; changes to one do not affect the other.
Examples: `std::string`, `std::vector`, domain types like `Money`, `Date`.

29.3.2 Unique ownership (move-only)

A unique owner represents exclusive ownership of a resource. Copying is typically disabled; moving transfers ownership. Examples: `std::unique_ptr`, file handles, sockets.

29.3.3 Shared ownership (explicit sharing)

Shared ownership means multiple objects jointly own a resource and the resource is released when the last owner is gone. In C++, this is typically expressed using `std::shared_ptr`. The Core Guidelines advise using shared ownership only when it is truly required.

29.4 Rule of zero first

29.4.1 Prefer the rule of zero

If your type can be built from members that already manage resources correctly (RAII types), do not write any special member functions. Let the compiler generate correct copy/move behavior.

```
#include <string>
#include <vector>

class Person {
public:
    Person(std::string name, int age)
        : name_(std::move(name)), age_(age) {}

private:
    std::string name_;
    int age_{0};
    std::vector<int> history_;
```

```
    // Rule of zero: correct copy/move/destruction automatically
};
```

29.5 Copy semantics

29.5.1 Copy constructor: create a new object from an existing one

A copy constructor should:

- produce an independent object with equivalent value (for value types),
- preserve invariants,
- avoid resource leaks and double deletion,
- provide the appropriate exception safety guarantees.

29.5.2 Copy assignment: replace this object's value

Copy assignment must also handle:

- self-assignment correctness,
- exception safety (often at least the basic guarantee: object remains valid),
- resource management consistency.

29.5.3 Deep copy for resource owners

If a type owns a resource and is copyable, copying must be defined (deep copy, copy-on-write is generally discouraged in modern C++ standard types, and sharing must be explicit).

```
#include <algorithm>
#include <cstddef>

class IntBuffer {
public:
    explicit IntBuffer(std::size_t n)
        : n_(n), p_(n ? new int[n] : nullptr) {}

    ~IntBuffer() noexcept { delete[] p_; }

    IntBuffer(const IntBuffer& other)
        : n_(other.n_), p_(other.n_ ? new int[other.n_] : nullptr) {
        std::copy(other.p_, other.p_ + n_, p_);
    }

    IntBuffer& operator=(const IntBuffer& other) {
        if (this == &other) return *this;

        int* newp = other.n_ ? new int[other.n_] : nullptr;
        std::copy(other.p_, other.p_ + other.n_, newp);

        delete[] p_;
        p_ = newp;
        n_ = other.n_;
        return *this;
    }

private:
    std::size_t n_{0};
    int* p_{nullptr};
};
```

Modern recommendation: Prefer `std::vector<int>` or `std::unique_ptr<int[]>` to avoid

manual management and enable rule-of-zero designs.

29.6 Move semantics

29.6.1 Move constructor and move assignment

Move operations transfer resources from a source object to a destination object. Key requirements:

- **No leaks, no double ownership.**
- The moved-from object must remain **valid** and destructible (but may be empty).

29.6.2 Move-only type (unique ownership) example

```
#include <memory>
#include <cstddef>

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_;
};
```

29.6.3 Why `noexcept` on moves matters

Standard containers may use move operations during reallocation only if moving is known not to throw (otherwise they may copy to preserve strong exception safety). Therefore, when moving is truly non-throwing, declare move operations `noexcept`.

29.7 Defaulted and deleted operations

29.7.1 Express intent with `=default` and `=delete`

Modern C++ encourages explicit defaulting/deleting to document semantics and avoid accidental copies or moves.

```
class NonCopyable {
public:
    NonCopyable() = default;
    ~NonCopyable() = default;

    NonCopyable(const NonCopyable&) = delete;
    NonCopyable& operator=(const NonCopyable&) = delete;

    NonCopyable(NonCopyable&&) noexcept = default;
    NonCopyable& operator=(NonCopyable&&) noexcept = default;
};
```

29.7.2 Be aware of implicit generation rules

The compiler's generation of copy/move operations depends on what you declare and on member types. Declaring one special member can suppress or change implicit generation of others. Therefore, when you declare any special member, re-check the full set.

29.8 Exception safety and strong invariants

29.8.1 Copy assignment and the strong guarantee

When appropriate, you can implement copy assignment using the copy-and-swap idiom to provide a strong exception guarantee: either the assignment succeeds fully, or the original object remains unchanged.

```
#include <utility>

class X {
public:
    X() = default;
    X(const X&) = default;
    X& operator=(X other) noexcept { // copy (or move) into 'other'
        swap(other);
        return *this;
    }

    void swap(X& other) noexcept {
        using std::swap;
        swap(a_, other.a_);
        swap(b_, other.b_);
    }

private:
```

```
    int a_{0};
    int b_{0};
};
```

29.8.2 Destructors must not throw

A destructor must not allow exceptions to escape, especially during stack unwinding. This influences design: operations that can fail should typically be explicit functions, not hidden in destructors.

29.9 Copy and move in class hierarchies

29.9.1 Avoid slicing

Copying polymorphic objects by value through a base type slices derived parts, breaking semantics. Pass polymorphic objects by reference/pointer, and if copying is needed, provide a virtual `clone()`.

```
#include <memory>

struct Shape {
    virtual ~Shape() = default;
    virtual std::unique_ptr<Shape> clone() const = 0;
};

struct Circle : Shape {
    std::unique_ptr<Shape> clone() const override {
        return std::make_unique<Circle>(*this);
    }
};
```

29.9.2 Consider deleting base copy operations

If copying a base by value would be wrong, delete copy operations in the base to prevent accidental slicing.

```
struct Base {
    Base() = default;
    Base(const Base&) = delete;
    Base& operator=(const Base&) = delete;
    virtual ~Base() = default;
};
```

29.10 Practical guidance for real code bases

29.10.1 Prefer standard RAII members

If you still own raw pointers, replace them with:

- `std::unique_ptr<T>` or `std::unique_ptr<T[]>` for unique ownership,
- `std::shared_ptr<T>` for shared ownership when required,
- standard containers for dynamic arrays.

29.10.2 Make semantics obvious at the interface

A user should be able to infer ownership and copying behavior from the type. Examples:

- move-only types should not appear copyable in APIs,
- shared ownership should be explicit (`shared_ptr` in the type),
- value types should behave predictably under copy and move.

29.11 Checklist

- Is the type a value type, unique owner, or shared owner? Is that visible in the interface?
- Can you follow the rule of zero (no custom SMFs)?
- If you declare any special member, did you decide the behavior of all five?
- For copyable types: does copy create an independent value and preserve invariants?
- For movable types: does move transfer ownership and leave the source valid?
- Are move operations `noexcept` when correct?
- Are destructors non-throwing and cleanup reliable?
- For hierarchies: is slicing prevented and copying provided via `clone()` if needed?

Chapter 30

Inheritance

30.1 Introduction

Inheritance is a mechanism for building **class hierarchies** where a derived class is related to a base class. In modern C++, inheritance is primarily justified for **runtime polymorphism**: selecting behavior through a base-class interface at run time.
However, inheritance is also one of the easiest ways to create:

- fragile designs (tight coupling to base internals),
- broken invariants across derived classes,
- undefined behavior (wrong destruction through base pointers),
- object slicing,
- hard-to-maintain dependency structures.

The ISO C++ Core Guidelines therefore recommend:

- **prefer composition** for reuse,
- use inheritance **only when it models substitutability** (“is-a”),
- design polymorphic bases carefully (virtual destructor, stable interface, no data exposure),
- avoid patterns that depend on downcasts and fragile base internals.

30.2 When inheritance is the right tool

30.2.1 Public inheritance for substitutability (“is-a”)

Public inheritance should represent a semantic relationship:

> If `D` publicly derives from `B`, then a `D` object should be usable wherever a `B` is expected, without breaking correctness.

This idea underlies the Guidelines rules for hierarchies: use inheritance only when you truly need runtime polymorphism and substitutability.

30.2.2 Prefer composition for reuse

If you only want code reuse (implementation reuse), prefer composition:

- a member object,
- a helper class,
- a free function or algorithm,
- templates (compile-time composition).

Composition avoids exposing a base representation and typically produces a design that is easier to evolve.

```
#include <string>

class Logger {
public:
    void info(const std::string& msg) { /* ... */ }
};

class Service {
public:
    explicit Service(Logger& log) : log_(log) {}

    void run() { log_.info("run"); /* ... */ }

private:
    Logger& log_; // composition: reuse without inheritance
};
```

30.3 The safe polymorphic base: required design elements

Polymorphic hierarchies are where inheritance is most often used. A **polymorphic class** is typically a class with at least one virtual function.

30.3.1 Virtual destructor (deletion through base)

If objects will be deleted through a base pointer, the base must have a **virtual destructor**. Otherwise deleting a derived object through `Base*` has undefined behavior.

```
struct Shape {
    virtual ~Shape() = default; // required for polymorphic deletion
```

```
    virtual void draw() const = 0;
};

struct Circle : Shape {
    void draw() const override { /* ... */ }
};
```

30.3.2 No public or protected representation

Do **not** put representation (data members) in `public` or `protected` in a base class. Exposing data makes derived classes depend on base internals and makes invariants hard to enforce. Prefer protected functions if you need extension points.

```
class Base {
public:
    virtual ~Base() = default;
    virtual void step() = 0;

protected:
    void log_step() { /* ... */ }  // protected function is safer than protected data
private:
    int internal_state_{0};        // representation stays private
};
```

30.3.3 Use `override` consistently

Mark overriding virtual functions with `override`. It documents intent and lets the compiler diagnose mismatches.

```
struct B {
    virtual ~B() = default;
    virtual int f(int) const = 0;
};
```

```
struct D : B {
    int f(int x) const override { return x + 1; }
};
```

30.3.4 Use `final` when further derivation is not intended

Use `final` to prohibit overriding/derivation where that is part of the design.

```
struct Task {
    virtual ~Task() = default;
    virtual void run() = 0;
};

struct ConcreteTask final : Task {
    void run() override { /* ... */ }
};
```

30.4 Object slicing and copying in hierarchies

30.4.1 Avoid slicing

Copying a derived object into a base object by value discards the derived part (slicing). Avoid passing polymorphic objects by value; use references or pointers.

```
struct B { virtual ~B() = default; virtual void f() const = 0; };
struct D : B { void f() const override {} };

// Bad: slices D into B
void take_by_value(B b);

// Good: no slicing
```

```
void take_by_ref(const B& b);
void take_by_ptr(const B* b);
```

30.4.2 If copying polymorphic objects is needed, use `clone()`

A common and guideline-aligned approach is a virtual `clone()` that returns an owning smart pointer to a new copy of the most-derived object.

```
#include <memory>

struct Shape {
    virtual ~Shape() = default;
    virtual std::unique_ptr<Shape> clone() const = 0;
    virtual void draw() const = 0;
};

struct Circle : Shape {
    std::unique_ptr<Shape> clone() const override {
        return std::make_unique<Circle>(*this);
    }
    void draw() const override { /* ... */ }
};
```

30.4.3 Consider deleting copy operations in the base

If copying a base by value would be wrong, delete copy operations to prevent accidental slicing-style usage.

```
struct Base {
    Base() = default;
    Base(const Base&) = delete;
    Base& operator=(const Base&) = delete;
    virtual ~Base() = default;
};
```

30.5 Virtual calls during construction and destruction

30.5.1 Do not call virtual functions from constructors or destructors

During construction and destruction, virtual dispatch does not behave as many programmers expect because the object is not yet (or is no longer) the most-derived type. Calling virtual functions from constructors/destructors can violate invariants and cause surprising behavior.

30.5.2 Use the Non-Virtual Interface (NVI) pattern for invariants

If a base must enforce invariants around a customization point, use a non-virtual public function that calls a protected virtual hook.

```
class Processor {
public:
    void process() {
        precheck();        // invariant/validation
        do_process();      // customization point
        postcheck();       // invariant/cleanup
    }

    virtual ~Processor() = default;

protected:
    virtual void do_process() = 0;

private:
    void precheck() { /* ... */ }
    void postcheck() { /* ... */ }
};
```

30.6 Overload sets and name hiding in derived classes

30.6.1 Use `using Base::f` to expose base overloads

In C++, a derived declaration of a function name hides all base overloads with the same name. Use `using Base::f;` to bring the base overload set into scope when intended.

```
struct Base {
    void f(int) {}
    void f(double) {}
};

struct Derived : Base {
    using Base::f; // keep both overloads visible
    void f(const char*) {}
};
```

30.7 Multiple inheritance and virtual inheritance

30.7.1 Multiple inheritance: use primarily for interfaces

Multiple inheritance can be safe when it models multiple **distinct interfaces** (pure abstract bases with no data and virtual destructor).

```
struct Printable {
    virtual ~Printable() = default;
    virtual void print() const = 0;
};

struct Serializable {
    virtual ~Serializable() = default;
    virtual void save() const = 0;
```

```
};

struct Report : Printable, Serializable {
    void print() const override { /* ... */ }
    void save()  const override { /* ... */ }
};
```

30.7.2 Virtual inheritance is for real diamond problems

Virtual inheritance changes object layout and construction rules. Use it only to solve a genuine diamond inheritance ambiguity and keep it rare and deliberate.

30.8 Downcasting and RTTI

30.8.1 Avoid downcasts when possible

If your design needs frequent downcasting, it usually indicates the base interface is wrong or incomplete. Prefer redesigning the interface to avoid type checks.

30.8.2 If you must check dynamic type, be explicit and safe

When runtime type checks are genuinely needed, `dynamic_cast` provides checked downcasts for polymorphic types. Use it sparingly and only when the design truly requires it.

```
struct Base { virtual ~Base() = default; };
struct Derived : Base { void g() {} };

void use(Base& b) {
    if (auto* p = dynamic_cast<Derived*>(&b)) {
        p->g();
    }
}
```

30.9 Public, protected, private inheritance

30.9.1 Prefer public inheritance for substitutability

Public inheritance is the normal choice for polymorphic interfaces.

30.9.2 Prefer composition over private inheritance

Private inheritance expresses an *implemented-in-terms-of* relationship, but often composition is clearer, safer, and more flexible. Use private inheritance only when you need access to protected members or want empty base optimization in controlled designs.

30.10 Practical hierarchy checklist

For a hierarchy intended for runtime polymorphism, verify:

- The base has a **virtual destructor** if deletion through `Base*` is possible.
- The base exposes **no public/protected data**; representation is private.
- Overriding functions are marked `override`.
- You prevent slicing: pass and store polymorphic objects by pointer/reference, not by value.
- If copying is needed, provide `clone()`; consider deleting base copy.
- You do not call virtual functions from constructors/destructors.
- The interface is small and stable; you do not force derived classes to depend on base internals.

- Multiple inheritance is used only for clear interface composition (or with a documented diamond reason).
- Downcasts are rare; if they are frequent, redesign the interface.

Chapter 31

Polymorphism

31.1 Introduction

Polymorphism means "many forms": calling an operation through a common interface can execute different implementations. In modern C++, polymorphism comes in two main forms:

- **Runtime polymorphism** (dynamic polymorphism): implemented with virtual functions and inheritance.
- **Compile-time polymorphism** (static polymorphism): implemented with templates, concepts, and overloading.

This chapter focuses on **runtime polymorphism** because it is the primary meaning of polymorphism in the ISO C++ Core Guidelines section on class hierarchies (`C.*`) and because it has unique correctness hazards (ownership, destruction, slicing).

31.2 When to use runtime polymorphism

31.2.1 Use it only when you need dynamic behavior selection

Runtime polymorphism is justified when:

- you do not know the concrete type at compile time,
- you need a stable interface for many implementations,
- you want to store heterogeneous objects behind a uniform interface,
- you need late binding across components (plugins, drivers, backends).

31.2.2 Prefer alternatives when runtime polymorphism is not required

The Core Guidelines emphasize: prefer simpler mechanisms when possible:

- **composition** for reuse,
- **templates** for static polymorphism (no virtual dispatch cost),
- **std::variant** for a closed set of alternatives (type-safe tagged union),
- **function objects/lambdas** for local customization.

31.3 Core principles of safe runtime polymorphism

31.3.1 Design for substitutability

If `D` derives from `B` and is used through `B`, then:

- base expectations must hold for derived objects,
- overriding must preserve the meaning of the base operation,
- invariants must not be weakened in derived types.

31.3.2 Keep base interfaces small and stable

A stable base class is usually:

- abstract (pure virtual functions),
- minimal (few operations),
- free of representation exposure (no public/protected data),
- designed for extension without breaking old derived classes.

31.3.3 Use inheritance primarily for runtime polymorphism

Inheritance should not be your default tool for reuse; prefer composition. Use inheritance when you truly want a substitutable interface with dynamic dispatch.

31.4 The polymorphic base class: mandatory rules

31.4.1 Virtual destructor when deletion via base is possible

A polymorphic base class must have a **virtual destructor** if an object may be deleted through a base pointer/reference. Otherwise, deleting a derived object through `Base*` is undefined behavior.

```
struct Shape {
    virtual ~Shape() = default;      // enables safe deletion through Shape*
    virtual void draw() const = 0;
};
```

31.4.2 Do not expose representation in a base class

Avoid `public` or `protected` data members in base classes; it tightly couples derived classes to base internals and makes invariants fragile. Use `private` data and provide protected functions if needed.

```
class Device {
public:
    virtual ~Device() = default;
    void reset() { pre_reset(); do_reset(); post_reset(); }

protected:
    virtual void do_reset() = 0;

private:
    void pre_reset()  { /* validate invariant */ }
    void post_reset() { /* re-establish invariant */ }
};
```

31.4.3 Use `override` and `final`

Mark every overriding function with `override`. Use `final` where further overriding/derivation is not part of the design.

```
struct Task {
    virtual ~Task() = default;
    virtual void run() = 0;
};

struct FileTask final : Task {
    void run() override { /* ... */ }
};
```

31.4.4 Do not call virtual functions from constructors/destructors

During construction/destruction, virtual dispatch does not behave as normal dynamic dispatch. Relying on virtual calls there can violate invariants and cause surprising behavior.

31.5 Object slicing and ownership

31.5.1 Avoid object slicing

Passing or storing polymorphic objects by value slices away the derived parts. Therefore, use polymorphic objects through references or pointers.

```
struct B { virtual ~B() = default; virtual void f() const = 0; };
struct D : B { void f() const override {} };

void bad(B b);            // slices
void good(const B& b);    // no slicing
```

31.5.2 Prefer `unique_ptr` for ownership

If a polymorphic object is owned, express ownership with `std::unique_ptr<Base>`. This documents lifetime and prevents leaks.

```
#include <memory>
#include <vector>

struct Shape {
    virtual ~Shape() = default;
    virtual void draw() const = 0;
};

struct Circle : Shape {
```

```
    void draw() const override {}
};

int main() {
    std::vector<std::unique_ptr<Shape>> v;
    v.push_back(std::make_unique<Circle>());
}
```

31.5.3 Copying polymorphic objects: use `clone()`

Copying polymorphic objects by value is usually wrong. If polymorphic copying is required, provide a virtual `clone()` that returns an owning smart pointer to a new object of the most-derived type.

```
#include <memory>

struct Shape {
    virtual ~Shape() = default;
    virtual std::unique_ptr<Shape> clone() const = 0;
    virtual void draw() const = 0;
};

struct Circle : Shape {
    std::unique_ptr<Shape> clone() const override {
        return std::make_unique<Circle>(*this);
    }
    void draw() const override {}
};
```

31.6 Runtime polymorphism patterns

31.6.1 Abstract base class (interface)

An abstract base class typically:

- has at least one pure virtual function,
- has a virtual destructor,
- has no public data members,
- defines only the essential interface.

31.6.2 Non-Virtual Interface (NVI)

NVI uses a non-virtual public function to enforce invariants and call a protected virtual function as a customization point. This helps maintain base invariants while allowing derived customization.

```
class Parser {
public:
    void parse() {
        begin();     // invariant checks
        do_parse();  // customization point
        end();       // invariant restoration
    }
    virtual ~Parser() = default;

protected:
    virtual void do_parse() = 0;

private:
    void begin() { /* ... */ }
    void end()   { /* ... */ }
};
```

31.6.3 Type erasure as an alternative (`std::function` and custom wrappers)

When you want runtime polymorphism without inheritance, type erasure can be used. The standard library provides type-erased call wrappers such as `std::function`. Use type erasure when you want to hide concrete types and avoid exposing a base class, but be aware of overhead (indirection and possible allocation).

31.7 Downcasting and RTTI

31.7.1 Avoid downcasting when possible

Frequent downcasts often indicate the base interface is not expressing the required operations. Prefer redesigning the interface rather than checking dynamic types.

31.7.2 If you must, use `dynamic_cast` deliberately

When runtime type checks are genuinely required, `dynamic_cast` provides checked downcasting for polymorphic types. Use it sparingly.

```
struct Base { virtual ~Base() = default; };
struct Derived : Base { void g() {} };

void use(Base& b) {
    if (auto* p = dynamic_cast<Derived*>(&b)) {
        p->g();
    }
}
```

31.8 Performance characteristics

31.8.1 What virtual dispatch costs

Runtime polymorphism typically involves:

- an extra indirection (vtable lookup),
- inhibition of some inlining and optimization across the call boundary,
- larger object size for vptr (implementation detail, but common),
- potential branch prediction effects in hot loops.

These costs are often acceptable when polymorphism provides a clean architecture, but in hot paths a static alternative (templates, variants) may be better.

31.9 Multiple inheritance: safe use

Multiple inheritance is safest when used to combine **pure interfaces** (abstract bases without data).

```
struct Printable {
    virtual ~Printable() = default;
    virtual void print() const = 0;
};

struct Serializable {
    virtual ~Serializable() = default;
    virtual void save() const = 0;
};
```

```
struct Report : Printable, Serializable {
    void print() const override {}
    void save()  const override {}
};
```

Virtual inheritance is a specialized tool to resolve real diamond base duplication and changes layout and construction rules; use it only when necessary and documented.

31.10 Checklist for safe polymorphic design

- Do you truly need runtime polymorphism, or would templates/variant/composition work better?
- Is the base interface minimal, coherent, and stable?
- Does the base have a virtual destructor if deletion through base is possible?
- Are data members not exposed in the base (no public/protected representation)?
- Are overrides marked `override` (and `final` used intentionally)?
- Are polymorphic objects handled by reference/pointer to avoid slicing?
- Is ownership explicit (typically `std::unique_ptr<Base>`)?
- If copying is needed, is it provided via `clone()` rather than copying by value?
- Are virtual functions not called from constructors/destructors?
- Are downcasts rare and justified (otherwise redesign)?

Chapter 32

`final` / `override`

32.1 Introduction

In modern C++, `override` and `final` are **contextual keywords** used with **virtual functions** and **inheritance**. They are not about performance tricks; they are about **correctness**, **maintainability**, and making class hierarchies safer.
The ISO C++ Core Guidelines recommend using these specifiers systematically:

- `override` to state intent and prevent accidental non-overrides,
- `final` to state design constraints and prevent unintended extension.

32.2 What `override` means

32.2.1 Definition

When a member function is declared with `override`, it must **override** a virtual function from a base class. If it does not, the program is ill-formed and the compiler must diagnose it.

32.2.2 Why `override` matters

Without `override`, a small mismatch (const-qualification, ref-qualification, parameter types, return type, or name hiding) can silently create a new function instead of overriding the base virtual function. This often leads to bugs that are difficult to detect.

32.2.3 Example: a silent bug without `override`

```
#include <string>

struct Base {
    virtual std::string name() const { return "Base"; }
};

struct Derived : Base {
    // Bug: missing 'const' => does not override Base::name()
    std::string name() { return "Derived"; }
};

void f(const Base& b) {
    // Calls Base::name() if Derived::name() did not override
    (void)b.name();
}
```

32.2.4 The same code fixed with `override`

```
#include <string>

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

```
struct Derived : Base {
    // Compiler enforces correctness:
    std::string name() const override { return "Derived"; }
};
```

32.2.5 Guideline practice: mark every override

The recommended practice is:

> **Always** write `override` on any function intended to override a base virtual function.

32.3 What `final` means

32.3.1 Definition (two forms)

`final` can be applied to:

- a **virtual function** to prevent further overriding,
- a **class** to prevent inheritance from that class.

32.3.2 Function `final`

A virtual function declared `final` cannot be overridden in derived classes.

```
struct Base {
    virtual void run() = 0;
    virtual ~Base() = default;
};

struct Mid : Base {
```

```
    void run() final { /* fixed behavior */ }
};

struct Derived : Mid {
    // void run() override {} // error: cannot override a final function
};
```

32.3.3 Class `final`

A class declared `final` cannot be used as a base class.

```
struct Engine final {
    void start() {}
};

// struct FancyEngine : Engine {}; // error: Engine is final
```

32.3.4 Why `final` matters

Use `final` to express and enforce design intent:

- prevent unintended extension that could break invariants,
- stabilize an interface for security-critical or correctness-critical components,
- document “this is a closed type” or “this function’s behavior must not change via overriding”.

32.4 How `override` and `final` interact with class design

32.4.1 Polymorphic base classes and safe overriding

For runtime polymorphism, base classes are designed to be overridden. In such designs:

- derived overrides should be explicit with `override`,
- certain customization points can be sealed with `final` if the base must enforce behavior.

32.4.2 Non-Virtual Interface (NVI) with `final`

A common robust pattern is:

- a public non-virtual function that enforces invariants,
- a protected virtual hook to customize,
- optionally `final` on the public function to prevent derived classes from bypassing invariants.

```
class Processor {
public:
    void process() final {
        precheck();
        do_process();
        postcheck();
    }

    virtual ~Processor() = default;

protected:
    virtual void do_process() = 0;

private:
    void precheck()  { /* validate invariant */ }
    void postcheck() { /* restore invariant */ }
};
```

32.5 Common pitfalls and how the specifiers prevent them

32.5.1 Name hiding and overload sets

If a derived class declares a function with the same name as in the base, it hides all base overloads with that name. This is separate from `override`, but it often appears in the same code. Use `using Base::f;` when you want to keep base overloads visible.

```
struct Base {
    virtual void f(int) {}
    virtual void f(double) {}
    virtual ~Base() = default;
};

struct Derived : Base {
    using Base::f; // keep overload set visible
    void f(int) override {} // overrides Base::f(int)
};
```

32.5.2 Signature mismatches

`override` catches mismatches in:

- `const` qualification,
- ref-qualification (&, &&),
- parameter types and cv qualifiers,
- noexcept specification as part of the function type in modern C++ rules,
- return type (except allowed covariant returns).

32.5.3 Accidental overriding vs. deliberate sealing

- Use `override` when overriding is intended.
- Use `final` when overriding must be forbidden for correctness or design reasons.

32.6 Performance notes (secondary to correctness)

Many compilers can devirtualize calls when they can prove the dynamic type is known. Marking a class or function `final` can sometimes help optimization by making the set of overrides closed. However, treat any optimization benefit as secondary; the primary purpose of `final`/`override` is correctness and design clarity.

32.7 Recommended style rules

32.7.1 A disciplined, guideline-aligned policy

- Mark **every** intended override with `override`.
- Consider marking leaf derived classes `final` when they are not meant to be extended.
- Consider marking key non-virtual-interface entry points `final` when derived classes must not bypass invariants.
- Keep base-class virtual interfaces minimal and stable.

32.7.2 Where to place the specifiers

Both `override` and `final` appear at the end of a virtual function declaration:

```
struct B {
    virtual void f() = 0;
    virtual ~B() = default;
};

struct D : B {
    void f() override final {}
};
```

32.8 Checklist

- Are all overriding functions marked `override`?
- Are any leaf classes that should not be extended marked `final`?
- Are there functions whose behavior must remain fixed (invariants/security) marked `final`?
- Are base overload sets preserved intentionally (using `using Base::name;`)?
- Are you avoiding calling virtual functions from constructors/destructors?

Part V

Enumerations (Enum)

Chapter 33

Enum.1 – Enum.10: Enumerations (Complete Numbered Set)

33.1 Scope and numbering note

This chapter lists **all guideline numbers** from **Enum.1** through **Enum.10**. In the current published C++ Core Guidelines revision dated **Jul 8, 2025**, the **Enumerations** section contains rules **Enum.1** through **Enum.8** only, and **Enum.9** and **Enum.10 are not assigned** in that revision.

33.2 Enum.1: Prefer enumerations over macros

Rule

Prefer `enum` / `enum class` to macros that define integral constants.

Reason

Macros do not respect scope or type rules and disappear during preprocessing, making debugging and tooling worse.

Example

```
// Bad: macros collide across headers and scopes
#define RED   0xFF0000
#define BLUE  0x0000FF

// Good: scoped and typed
enum class Web_color { red = 0xFF0000, blue = 0x0000FF };
enum class Product_color { red = 0, blue = 1 };

Web_color c = Web_color::blue;
// int x = blue; // error: forces you to be specific
```

Practical note

For single constants, also consider `constexpr` or `inline const` variables instead of macros.

33.3 Enum.2: Use enumerations to represent sets of related named constants

Rule

Use an enumeration when you have a group of related named integral values.

Reason

An enumeration documents that the values belong together and can form a distinct type.

Example

```
enum class Status { ok, failed, timeout };

Status s = Status::timeout;
```

Switch completeness

Switching on an enum allows tools/compilers to warn about missing cases when enumerators change (for example, when a new enumerator is added).

```
#include <iostream>

enum class Color { red, purple, blue };

void print(Color c) {
    switch (c) {
    case Color::red:    std::cout << "red"; break;
    case Color::purple: std::cout << "purple"; break;
    case Color::blue:   std::cout << "blue"; break;
    }
}
```

33.4 Enum.3: Prefer `enum class` over "plain" `enum`

Rule

Prefer scoped enumerations (`enum class`) to unscoped (traditional) `enum`.

Reason

Unscoped enums convert too readily to integers and their enumerator names leak into the surrounding scope, increasing the chance of surprises and name collisions. Scoped enums avoid both problems.

Example

```
void print_color(int); // legacy API

enum Web_color { red = 0xFF0000, blue = 0x0000FF };
enum Product_info { purple = 1 };

Web_color w = Web_color::blue;
print_color(w);                 // compiles: implicit conversion to int (surprising)
print_color(Product_info::purple); // compiles too (very likely wrong)

enum class Safe_color { red = 0xFF0000, blue = 0x0000FF };
Safe_color s = Safe_color::blue;
// print_color(s); // error: no implicit conversion
```

33.5 Enum.4: Define operations on enumerations for safe and simple use

Rule

When an enum needs arithmetic, iteration, bitwise operations, or other behavior, define the operations explicitly.

Reason

This improves convenience while avoiding repeated casts and reducing errors.

Example: incrementing a day of week

```
#include <type_traits>

enum class Day { mon, tue, wed, thu, fri, sat, sun };

constexpr Day next_day(Day d) {
    using U = std::underlying_type_t<Day>;
    return (d == Day::sun)
        ? Day::mon
        : static_cast<Day>(static_cast<U>(d) + 1);
}
```

Example: bitmask flags

```
#include <type_traits>

enum class Perm : unsigned {
    none  = 0,
    read  = 1u << 0,
    write = 1u << 1,
    exec  = 1u << 2
};

constexpr Perm operator|(Perm a, Perm b) {
    using U = std::underlying_type_t<Perm>;
    return static_cast<Perm>(static_cast<U>(a) | static_cast<U>(b));
}
```

```
constexpr bool has(Perm p, Perm bit) {
    using U = std::underlying_type_t<Perm>;
    return (static_cast<U>(p) & static_cast<U>(bit)) != 0;
}
```

33.6 Enum.5: Don't use ALL_CAPS for enumerators

Rule

Avoid ALL_CAPS enumerator names.

Reason

ALL_CAPS names are commonly used for macros and can collide with macros from other headers.

Example

```
enum class Color { red, green, blue }; // good
// enum class Color { RED, GREEN, BLUE }; // risky: likely macro collisions
```

33.7 Enum.6: Avoid unnamed enumerations

Rule

Do not use unnamed enums as a container for unrelated constants.

Reason

If you cannot name the enum, the values are often not truly related; it becomes a disguised, unsafe way to define constants.

Alternative

Use `constexpr` variables with appropriate types.

```
// Bad: unrelated values packed into an unnamed enum
// enum { red = 0xFF0000, scale = 4, is_signed = 1 };

// Good: typed constants
constexpr int   red       = 0xFF0000;
constexpr short scale     = 4;
constexpr bool  is_signed = true;
```

33.8 Enum.7: Specify the underlying type only when necessary

Rule

Do not specify an enum's underlying type unless there is a real need.

Reason

The default is simplest and most readable; explicitly writing an underlying type can be redundant noise.

When it is necessary or justified

- **Forward declarations** of enums require specifying the underlying type.
- **ABI/storage constraints** (e.g., compact storage, file/network formats) may justify a fixed underlying type.

Examples

```
#include <cstdint>

// Necessary for forward declaration:
enum class Flags : std::uint8_t;
void f(Flags);

// Later:
enum class Flags : std::uint8_t { a = 1, b = 2 };

// Possibly justified: compact storage
enum class Direction : std::uint8_t { n, s, e, w, ne, nw, se, sw };
```

33.9 Enum.8: Specify enumerator values only when necessary

Rule

Do not assign explicit values to enumerators unless needed.

Reason

Default consecutive values are simplest and help avoid typos such as accidental duplicates. They also fit common `switch` implementations.

When explicit values are necessary

- Conventional external numbering (e.g., months starting at 1).
- Bitmask flags (separate bits).
- Interoperability with protocols/files/hardware registers.

Examples

```
enum class Color1 { red, yellow, blue }; // simplest

enum class Color2 { red = 1, yellow = 2, blue = 2 }; // bug: duplicate value (typo)

enum class Month { jan = 1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };

enum class Base_flag { dec = 1, oct = dec << 1, hex = dec << 2 }; // separate bits
```

33.10 Enum.9: Not assigned in the Jul 8, 2025 revision

Status: The published Enumerations section lists rules **Enum.1** through **Enum.8**; there is no **Enum.9** rule in that revision.

33.11 Enum.10: Not assigned in the Jul 8, 2025 revision

Status: The published Enumerations section lists rules **Enum.1** through **Enum.8**; there is no **Enum.10** rule in that revision.

33.12 Quick checklist (engineering use)

- Use `enum class` by default.
- Use enums for related sets of named values, not macros.
- Keep enumerator names out of `ALL_CAPS`.
- Use named enums; use `constexpr` for unrelated constants.
- Specify underlying type and enumerator values only when needed (forward-decl, ABI, formats, flags).
- Provide explicit operators/helpers for flags and safe iteration.

Chapter 34

`enum class`

34.1 Introduction

A scoped enumeration, written as `enum class`, is the modern C++ form of enumeration introduced in C++11. It provides a *distinct type* whose enumerator names are *scoped* and whose values do *not* implicitly convert to integers.
The ISO C++ Core Guidelines explicitly recommend preferring `enum class` over traditional unscoped `enum` to reduce surprises and bugs caused by implicit conversions and name leakage.

34.2 Why `enum class` is the default choice

34.2.1 Scoped enumerators prevent name pollution

With traditional `enum`, enumerator names enter the surrounding scope, increasing the chance of collisions. With `enum class`, you must qualify enumerators, improving clarity and avoiding collisions.

```
enum Color_old { red, green, blue };      // enumerators leak into enclosing scope
```

```
enum class Color { red, green, blue };    // enumerators are scoped

Color c1 = Color::red; // OK
// Color_old c2 = red; // OK, but 'red' is a global name now
```

34.2.2 No implicit conversion to `int`

Traditional enums convert to integers too readily. Scoped enums do not implicitly convert, so accidental mixing with unrelated integers (or other enums) is rejected by the compiler.

```
#include <cstdint>

enum Web_color { red = 0xFF0000, blue = 0x0000FF };

enum class Safe_color : std::uint32_t { red = 0xFF0000, blue = 0x0000FF };

void paint(std::uint32_t);

void demo() {
    paint(Web_color::blue);           // allowed via implicit conversion (often unwanted)
    // paint(Safe_color::blue);        // error: no implicit conversion
    paint(static_cast<std::uint32_t>(Safe_color::blue)); // explicit and clear
}
```

34.3 Syntax and core rules

34.3.1 Basic declaration

```
enum class Status { ok, failed, timeout };
```

34.3.2 Underlying type (only when needed)

A scoped enum may specify a fixed underlying type:

```
#include <cstdint>
enum class Byte : std::uint8_t { a = 1, b = 2 };
```

Specify the underlying type only when you have a real need, such as:

- ABI / storage constraints (wire formats, file formats, hardware registers),
- forward declaration of an enum (requires fixed underlying type),
- very large enumerator values that do not fit the default underlying type.

34.3.3 Forward declaration

Forward declaring an enumeration is allowed in modern C++, but an incomplete enumeration type must have a fixed underlying type.

```
#include <cstdint>

enum class Error_code : std::uint16_t; // forward declaration (incomplete type)

void handle(Error_code);

enum class Error_code : std::uint16_t { ok = 0, io = 5, bad_data = 9 };
```

34.4 Using `enum class` in control flow

34.4.1 `switch` over a scoped enum

Use `switch` for clear branching on enumerators. When you switch over an enum, keep cases complete and consider tool/compiler warnings for missing cases as your enum evolves.

```
#include <string>

enum class Mode { fast, safe, debug };

std::string to_string(Mode m) {
    switch (m) {
    case Mode::fast:  return "fast";
    case Mode::safe:  return "safe";
    case Mode::debug: return "debug";
    }
    return "unknown"; // defensive fallback if new enumerators are added later
}
```

34.5 Converting to the underlying integer safely

34.5.1 Prefer `std::to_underlying` in C++23 and later

When you need the underlying integer value, C++23 provides `std::to_underlying`. It converts an enum value to its underlying type without guessing or narrowing.

```
#include <cstdint>
#include <utility> // std::to_underlying (C++23)

enum class Flags : std::uint8_t { a = 1, b = 2 };

std::uint8_t raw(Flags f) {
    return std::to_underlying(f); // returns std::uint8_t
}
```

34.5.2 Pre-C++23 approach: `std::underlying_type`

If `std::to_underlying` is not available, use `std::underlying_type` and a cast.

```
#include <type_traits>

template<class E>
constexpr auto to_u(E e) noexcept -> std::underlying_type_t<E> {
    return static_cast<std::underlying_type_t<E>>(e);
}
```

34.6 Flags and bitmasks with `enum class`

34.6.1 Do not rely on implicit bitwise operations

Scoped enums do not implicitly behave like integers, so bitwise operations are not available unless you define them. This is usually good: it prevents accidental misuse.

34.6.2 Define explicit operators for flag enums

For an enum that represents a set of independent bits, define the required operators explicitly.

```
#include <type_traits>

enum class Perm : unsigned {
    none  = 0u,
    read  = 1u << 0,
    write = 1u << 1,
    exec  = 1u << 2
};

constexpr Perm operator|(Perm a, Perm b) noexcept {
    using U = std::underlying_type_t<Perm>;
    return static_cast<Perm>(static_cast<U>(a) | static_cast<U>(b));
}
```

```
constexpr Perm operator&(Perm a, Perm b) noexcept {
    using U = std::underlying_type_t<Perm>;
    return static_cast<Perm>(static_cast<U>(a) & static_cast<U>(b));
}

constexpr bool has(Perm set, Perm bit) noexcept {
    return (set & bit) != Perm::none;
}
```

34.7 Modern convenience: `using enum` (C++20)

C++20 introduced `using enum` to bring enumerator names into the current scope deliberately. It can improve readability in small, local scopes while preserving the stronger typing of `enum class`.

```
enum class Color { red, green, blue };

int main() {
    using enum Color; // introduces red, green, blue into this scope
    Color c = red;    // OK here, still type-safe
    (void)c;
}
```

Use `using enum` sparingly and locally to avoid reintroducing large-scale name pollution.

34.8 Design guidance aligned with the Core Guidelines

34.8.1 Prefer `enum class` by default

This corresponds to the Core Guidelines rule to prefer scoped enums over plain enums to minimize surprises.

34.8.2 Avoid `ALL_CAPS` enumerators

Enumerator names in `ALL_CAPS` are more likely to conflict with macros from legacy headers and third-party libraries. Prefer normal identifiers.

34.8.3 Assign explicit values only when necessary

Avoid specifying enumerator values unless you have a reason (external protocol, bit flags, conventional numbering). Default consecutive values are simpler and reduce mistakes like accidental duplicates.

34.8.4 Document meaning and keep enums cohesive

Use an enum to represent a cohesive set of related named constants, not as a container for unrelated integers. If you cannot name the enum meaningfully, prefer typed `constexpr` constants.

34.9 Common mistakes and how `enum class` helps

34.9.1 Accidental mixing of unrelated enums

Two unscoped enums may both convert to `int`, enabling unintended comparisons and arithmetic. Scoped enums prevent this unless you explicitly convert.

34.9.2 Overusing casts

If you find yourself writing many casts from an enum to integers, reconsider the API:

- keep operations in the enum domain (write helper functions),
- return/accept the enum type where possible,

- convert at boundaries only (serialization, logging, protocol I/O).

34.10 Checklist

- Use `enum class` by default.
- Specify an underlying type only when needed (forward declaration, ABI/storage, protocols).
- Do not assign explicit enumerator values unless necessary.
- Define explicit operators for flag-style enums.
- Prefer `std::to_underlying` (C++23) when you need the underlying integer.
- Use `using enum` (C++20) only locally when it improves clarity.
- Avoid `ALL_CAPS` enumerators to reduce macro-collision risk.

Chapter 35

Avoiding Unsafe Enums

35.1 Introduction

Enumerations are one of the best tools in C++ for giving names to related integral values. However, several common enum usage patterns are **unsafe** because they:

- allow accidental implicit conversions to integers,
- allow mixing values from unrelated enums,
- leak enumerator names into surrounding scopes,
- encourage “magic numbers” and macro-like usage,
- hide preconditions (e.g., which values are valid),
- make bitmask usage error-prone.

The ISO C++ Core Guidelines address these risks directly in the **Enum** rules, especially by recommending `enum class`, minimizing explicit values, and avoiding naming patterns that

collide with macros. This chapter shows how unsafe enum patterns arise and how to replace them with safer designs grounded in standard C++ semantics.

35.2 What makes an enum “unsafe” in practice

35.2.1 Unscoped enums leak names and convert implicitly

Traditional (unscoped) enums place enumerator names in the enclosing scope and can implicitly convert to integer types. This combination is responsible for many real bugs: accidental overload selection, wrong comparisons, and unintended arithmetic.

```
#include <cstdint>

void set_color(std::uint32_t);

enum Web_color { red = 0xFF0000, blue = 0x0000FF };
enum Product_color { red_product = 1, blue_product = 2 };

void demo() {
    set_color(Web_color::blue); // implicit conversion to integer: often unintended
    // set_color(blue); // 'blue' is a global name (name leakage risk)
}
```

35.2.2 Enums can carry invalid values

Even if an enum has named enumerators, variables of that enum type can still hold other values (typically via casts, I/O, or unvalidated input). Unsafe code assumes “if it is of enum type, it must be valid”. Safer code validates at boundaries.

35.2.3 Flags are often modeled incorrectly

A frequent unsafe pattern is using an enum for bit flags without defining explicit bitwise operations and without controlling conversions. This leads to repeated casts, unclear intent, and accidental mixing with integers.

35.3 Core Guidelines strategy for safety

The Core Guidelines' enumerations rules can be read as a safety strategy:

- **Prefer** `enum class` (scoped, strongly typed) (Enum.3).
- **Prefer enums over macros** (Enum.1) and avoid `ALL_CAPS` enumerators (Enum.5).
- **Avoid unnamed enums** used as ad-hoc constant buckets (Enum.6).
- **Specify underlying types and explicit enumerator values only when necessary** (Enum.7, Enum.8).
- **Define needed operations explicitly** (Enum.4) rather than relying on implicit integer behavior.

35.4 Safer alternative #1: use `enum class` by default

35.4.1 Scoped enumerators prevent name collisions

`enum class` keeps enumerators scoped and forces qualification.

```
enum class Color { red, green, blue };

Color c = Color::red; // explicit and unambiguous
```

35.4.2 No implicit conversion to `int`

`enum class` prevents accidental integer conversions and accidental mixing of unrelated domains.

```
#include <cstdint>

enum class Web_color : std::uint32_t { red = 0xFF0000, blue = 0x0000FF };

void set_color(std::uint32_t);

void demo() {
    // set_color(Web_color::blue); // error: no implicit conversion
    set_color(static_cast<std::uint32_t>(Web_color::blue)); // explicit boundary conversion
}
```

35.5 Safer alternative #2: keep enums cohesive and named

35.5.1 Avoid unnamed enums for unrelated constants

Unnamed enums are often used as a macro replacement for unrelated constants, which defeats the purpose of having a meaningful type. Use named enums for cohesive sets, and use typed `constexpr` variables for unrelated constants.

```
// Bad: unrelated values packed into an unnamed enum
// enum { buffer_size = 4096, max_retry = 3, red = 0xFF0000 };

// Good:
constexpr int buffer_size = 4096;
constexpr int max_retry   = 3;

enum class Color { red, green, blue };
```

35.5.2 Avoid `ALL_CAPS` enumerators

`ALL_CAPS` names commonly collide with macros in legacy headers and third-party libraries. Use normal identifiers.

```
enum class Mode { fast, safe, debug }; // good
```

35.6 Safer alternative #3: convert explicitly at boundaries

35.6.1 Prefer `std::to_underlying` in C++23

When you must serialize, log, store, or interoperate with C APIs, convert explicitly at the boundary. C++23 provides `std::to_underlying`.

```
#include <cstdint>
#include <utility> // std::to_underlying

enum class Level : std::uint8_t { low = 1, high = 2 };

std::uint8_t encode(Level L) {
    return std::to_underlying(L);
}
```

35.6.2 Pre-C++23: use `std::underlying_type`

```
#include <type_traits>

template<class E>
constexpr auto to_u(E e) noexcept -> std::underlying_type_t<E> {
    return static_cast<std::underlying_type_t<E>>(e);
}
```

35.7 Safer alternative #4: validate enum values from external input

35.7.1 Use a “try-parse” pattern

If enum values come from I/O, network, files, or untrusted sources, validate before constructing or using the enum. This prevents invalid values from propagating.

```
#include <optional>
#include <cstdint>

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;
    }
}
```

35.7.2 Prefer mapping functions over casts

Casts accept any value; mapping functions define the allowed set explicitly.

35.8 Safer alternative #5: design flags as flags (and define operations)

35.8.1 Define explicit bitwise operators

If an enum represents independent bits, define the allowed bitwise operations explicitly (Enum.4). Keep the underlying type unsigned and avoid accidental integer mixing.

```
#include <type_traits>

enum class Perm : unsigned {
    none  = 0u,
    read  = 1u << 0,
    write = 1u << 1,
    exec  = 1u << 2
};

constexpr Perm operator|(Perm a, Perm b) noexcept {
    using U = std::underlying_type_t<Perm>;
    return static_cast<Perm>(static_cast<U>(a) | static_cast<U>(b));
}

constexpr Perm operator&(Perm a, Perm b) noexcept {
    using U = std::underlying_type_t<Perm>;
    return static_cast<Perm>(static_cast<U>(a) & static_cast<U>(b));
}

constexpr bool has(Perm set, Perm bit) noexcept {
    return (set & bit) != Perm::none;
}
```

35.8.2 Keep explicit enumerator values only when needed

Flags are one of the legitimate cases for explicit enumerator values (Enum.8).

35.9 Safer alternative #6: keep underlying types and explicit values minimal

35.9.1 Do not specify underlying type unless necessary

Specifying underlying type is justified for ABI/storage constraints and for forward declarations (Enum.7).

```
#include <cstdint>

// Forward declaration requires underlying type:
enum class Error_code : std::uint16_t;

enum class Error_code : std::uint16_t { ok = 0, io = 5 };
```

35.9.2 Do not assign explicit values unless necessary

Avoid explicit values unless you need them (protocols, flags, conventional numbering). This reduces duplicates and maintenance errors (Enum.8).

```
enum class Color { red, yellow, blue }; // simplest and safest
```

35.10 Safer alternative #7: write safe API surfaces

35.10.1 Prefer enums in public APIs instead of integers

If a function expects one of a fixed set of choices, use an enum (prefer `enum class`) rather than `int`. This makes illegal calls harder to write.

```
enum class Format { text, binary };

void save(Format f); // better than void save(int);
```

35.10.2 Avoid “sentinel” values like `-1` when possible

Instead of inventing out-of-range sentinel integers, use:

- `std::optional<Enum>` for “no value”,
- a separate state type,
- or a dedicated enumerator only if it is semantically meaningful.

35.11 Safe usage patterns in real code

35.11.1 Provide conversion to text via total mapping

A total mapping avoids undefined behavior and improves maintainability.

```
#include <string_view>

enum class Mode { fast, safe, debug };

constexpr std::string_view to_string(Mode m) noexcept {
    switch (m) {
    case Mode::fast:  return "fast";
    case Mode::safe:  return "safe";
    case Mode::debug: return "debug";
    }
    return "unknown"; // defensive for future enum extensions
}
```

35.11.2 Use `using enum` locally when it improves clarity (C++20)

`using enum` can reduce verbosity without losing type safety, but use it only in small scopes to avoid new name pollution.

```
enum class Color { red, green, blue };

void paint(Color c) {
    using enum Color; // local only
    if (c == red) { /* ... */ }
}
```

35.12 Checklist: how to avoid unsafe enums

- Use `enum class` by default (scoped, no implicit int conversion).
- Use enums for cohesive, named sets of related values; avoid unnamed enums for constants.
- Avoid `ALL_CAPS` enumerators to reduce macro collision risk.
- Convert to integers only at boundaries (serialization, logging, C APIs), preferably with `std::to_underlying` (C++23).
- Validate enum values that come from untrusted sources; avoid "just cast it".
- For flag enums, define explicit bitwise operators and helpers; do not rely on implicit integer behavior.
- Specify underlying type and explicit enumerator values only when necessary.

Chapter 36

Explicit Conversions

36.1 Introduction

An **explicit conversion** is any conversion that the programmer must request directly (typically with a cast, explicit construction, or an explicit conversion function). In modern C++, explicit conversions are essential for writing safe code because they:

- make intent visible at the call site,
- prevent accidental loss of information,
- prevent accidental mixing of unrelated domains (especially with `enum class`),
- make interfaces more robust under maintenance and refactoring.

The ISO C++ Core Guidelines recommend using **strong typing** (notably `enum class`) and **explicit operations** to reduce surprises, especially for enumerations.

36.2 Why explicit conversions matter for enumerations

36.2.1 The core problem: integers are too permissive

Integers happily accept many kinds of values. If your API expects a closed set of choices, an integer does not communicate that. Traditional unscoped enums implicitly convert to integers, so they can silently participate in arithmetic, overload resolution, and comparisons.

36.2.2 `enum class` enforces explicit conversions

A scoped enum (`enum class`) does *not* implicitly convert to an integer type. This forces explicit conversion at boundaries (serialization, logging, C APIs), and prevents accidental mixing.

```
#include <cstdint>

void write_u32(std::uint32_t);

enum Web_color { red = 0xFF0000, blue = 0x0000FF }; // unscoped
enum class Safe_color : std::uint32_t { red = 0xFF0000, blue = 0x0000FF }; // scoped

void demo() {
    write_u32(Web_color::blue); // OK (implicit), but may hide intent

    // write_u32(Safe_color::blue); // error: no implicit conversion
    write_u32(static_cast<std::uint32_t>(Safe_color::blue)); // explicit boundary
    ↪  conversion
}
```

36.3 Forms of explicit conversions in C++

36.3.1 Explicit construction

A value can be explicitly converted by constructing the destination type. This is especially important when constructors are marked `explicit`.

```
#include <string>

struct Name {
    explicit Name(std::string s) : s_(std::move(s)) {}
    std::string s_;
};

void f() {
    // Name n = "x";          // error (would require implicit conversion)
    Name n{std::string{"x"}}; // explicit construction
    (void)n;
}
```

36.3.2 `static_cast`

`static_cast<T>(expr)` performs well-defined, compile-time checked conversions that the language permits. For enums, it is the fundamental cast to convert between an enum and its underlying integer type (or vice versa), but it must be used carefully because it can also create invalid enum values.

```
#include <cstdint>

enum class Status : std::uint8_t { ok = 0, failed = 1, timeout = 2 };

std::uint8_t encode(Status s) {
    return static_cast<std::uint8_t>(s);
}
```

```
Status unsafe_decode(std::uint8_t raw) {
    return static_cast<Status>(raw); // may create invalid Status values
}
```

36.3.3 Library helpers for enum conversion

36.3.3.1 `std::to_underlying` (C++23)

C++23 provides `std::to_underlying(e)` to convert an enum value to its underlying type without repeating the type name and without risking conversion to the *wrong* integer type.

```
#include <cstdint>
#include <utility> // std::to_underlying

enum class Flags : std::uint8_t { a = 1, b = 2 };

std::uint8_t raw(Flags f) {
    return std::to_underlying(f); // returns std::uint8_t
}
```

36.3.3.2 `std::underlying_type` (pre-C++23 helper)

If `std::to_underlying` is not available, `std::underlying_type_t<E>` gives the underlying type of an enum `E`. This supports writing a safe, generic helper.

```
#include <type_traits>

template<class E>
constexpr auto to_u(E e) noexcept -> std::underlying_type_t<E> {
    return static_cast<std::underlying_type_t<E>>(e);
}
```

36.3.4 User-defined explicit conversion operators

Classes may provide `explicit operator T()` to require a cast at call sites. This is useful for potentially lossy or conceptually non-trivial conversions.

```
#include <cstdint>

class Port {
public:
    explicit Port(std::uint16_t v) : v_(v) {}
    explicit operator std::uint16_t() const noexcept { return v_; }
private:
    std::uint16_t v_;
};

void use_u16(std::uint16_t);

void demo() {
    Port p{443};
    // use_u16(p);                       // error
    use_u16(static_cast<std::uint16_t>(p)); // explicit
}
```

36.4 Guideline-aligned rules for explicit conversions with enums

36.4.1 Prefer `enum class` to force explicit conversions

The Core Guidelines recommend scoped enums to avoid implicit conversion to integers and name leakage.

36.4.2 Define operations instead of relying on integer casts

If an enum needs operations (increment, comparison helpers, bitwise flags), define them as functions/operators. This avoids scattered casts and makes intent explicit and safe.

```
#include <type_traits>

enum class Day { mon, tue, wed, thu, fri, sat, sun };

constexpr Day next(Day d) {
    using U = std::underlying_type_t<Day>;
    return (d == Day::sun)
        ? Day::mon
        : static_cast<Day>(static_cast<U>(d) + 1);
}
```

36.4.3 Convert only at boundaries

Treat integer conversion as an **I/O boundary** concern:

- encoding/decoding to file/network formats,
- interfacing with C or OS APIs,
- logging and telemetry.

Inside your domain logic, keep values typed as enums and operate in the enum domain.

36.5 The unsafe part: explicit casts can still be wrong

36.5.1 Explicit does not guarantee correctness

A cast can silence the type system even when the result is meaningless. Two particularly important cases for enums:

36.5.2 Casting integers to enums can create invalid values

The standard permits conversions that result in an enum object that does not correspond to any named enumerator. That value may still exist in the program and must be handled safely.

36.5.3 Preferred approach: validate during decoding

When decoding from an integer, prefer mapping/validation rather than blindly casting. Use a `switch` or table to accept only valid values.

```
#include <cstdint>
#include <optional>

enum class Status : std::uint8_t { ok = 0, failed = 1, timeout = 2 };

std::optional<Status> try_decode(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;
    }
}
```

36.6 Explicit conversions and narrowing

36.6.1 Narrowing conversions are a common source of defects

Converting from a larger range to a smaller one (e.g., `int` to `std::uint8_t`) can lose information. Prefer designs that avoid narrowing or that validate before narrowing.

```
#include <cstdint>
```

```cpp
#include <stdexcept>

std::uint8_t to_u8_checked(int x) {
    if (x < 0 || x > 255) throw std::out_of_range("not a byte");
    return static_cast<std::uint8_t>(x);
}
```

36.7 Choosing the right explicit conversion technique

36.7.1 For enums to integers

- Prefer `std::to_underlying` (C++23) for clarity and correctness of the target type.
- Otherwise use a small helper based on `std::underlying_type`.
- Avoid scattering raw casts across the codebase; centralize conversions at boundaries.

36.7.2 For integers to enums

- Prefer validation/mapping (“try-parse”) over direct casting.
- Use `std::optional` (or an error type) to represent invalid inputs.

36.7.3 For domain types to primitives

- Use `explicit` constructors and `explicit operator T()` to prevent accidental conversions.
- Provide named functions (e.g., `to_int()`, `value()`) when readability is better than a cast.

36.8 Checklist

- Use `enum class` to force explicit conversions and prevent name leakage.
- Convert enums to integers only at boundaries; prefer `std::to_underlying` (C++23).
- Do not decode integers to enums using unchecked casts; validate inputs.
- Define enum operations explicitly (helpers/operators) rather than scattering casts.
- Use `explicit` constructors and explicit conversion operators to prevent accidental conversions.

Chapter 37

Type Compatibility

37.1 Introduction

In the context of enumerations, **type compatibility** means:

- when two values are considered to have the same type and can be used interchangeably without conversion,
- when the language permits implicit conversions between an enum and other types (especially integers),
- how overload resolution and comparisons behave when enums interact with other types,
- how to design APIs so that incompatible values cannot be mixed accidentally.

The ISO C++ Core Guidelines treat type compatibility for enums as a safety problem: they recommend `enum class` to prevent implicit conversions and accidental mixing across domains. This chapter focuses on enums, but the principles apply broadly to designing type-safe interfaces.

37.2 Compatibility rules: unscoped `enum` vs. scoped `enum class`

37.2.1 Unscoped `enum`: compatible with integers in many contexts

Traditional (unscoped) enums:

- inject enumerator names into the enclosing scope,
- can convert implicitly to an integer type,
- may participate in arithmetic and comparisons with integers,
- may accidentally select integer overloads.

This creates **over-compatibility**: too many operations are permitted.

```
#include <cstdint>

void f(int);
void f(double);

enum Color { red = 1, blue = 2 };

void demo() {
    Color c = red;
    f(c);        // calls f(int) via implicit conversion (often unintended)
    int x = c;   // allowed (may hide intent)
    (void)x;
}
```

37.2.2 Scoped `enum class`: incompatible with integers by default

`enum class` values:

- do not implicitly convert to integers,
- do not implicitly convert to other enums,
- require explicit conversions at boundaries.

This prevents accidental mixing and makes compatibility a deliberate choice.

```
#include <cstdint>

void f(int);

enum class Color : std::uint8_t { red = 1, blue = 2 };

void demo() {
    Color c = Color::red;
    // f(c);        // error: no implicit conversion to int
    f(static_cast<int>(c)); // explicit boundary conversion
}
```

37.3 Compatibility across different enums

37.3.1 Different enum types are not interchangeable

Even if two enums share the same underlying type and enumerator values, they represent different domains. Treating them as compatible is a common source of bugs.

```
enum class Web_color  : unsigned { red = 0xFF0000, blue = 0x0000FF };
enum class Led_state  : unsigned { red = 1,        blue = 2        };
```

```
void paint(Web_color);

void demo() {
    // paint(Led_state::red); // error: incompatible types (good)
}
```

With unscoped enums, accidental mixing is easier because both can convert to integers, defeating domain separation.

37.4 Underlying types and representation compatibility

37.4.1 Underlying type basics

Every enum has an **underlying integer type**. For scoped enums, you may specify it explicitly; otherwise it is chosen by the implementation subject to the standard's constraints.

37.4.2 When underlying type matters for compatibility

Underlying type affects:

- storage size and ABI (important for binary I/O, networking, file formats),
- interop with C APIs and OS APIs,
- forward declarations (an incomplete enum type must have a fixed underlying type),
- conversion safety (narrowing and sign).

```
#include <cstdint>

// Forward declaration requires explicit underlying type:
```

```
enum class Error_code : std::uint16_t;
void handle(Error_code);

enum class Error_code : std::uint16_t { ok = 0, io = 5, bad_data = 9 };
```

37.4.3 Do not treat “same underlying type” as semantic compatibility

Two enums having the same underlying type does not make them semantically interchangeable. Maintain type separation and convert explicitly only where the meaning is correct.

37.5 Compatibility with integers and explicit conversions

37.5.1 Prefer explicit conversions at boundaries

When you must interface with integer-based protocols or legacy APIs:

- convert explicitly and locally,
- avoid sprinkling casts across the codebase,
- validate when converting from integer to enum (decoding).

37.5.2 Use `std::to_underlying` (C++23) for encoding

C++23 provides `std::to_underlying` to convert an enum to its underlying type reliably.

```
#include <cstdint>
#include <utility> // std::to_underlying

enum class Mode : std::uint8_t { fast = 1, safe = 2 };

std::uint8_t encode(Mode m) {
```

```
    return std::to_underlying(m);
}
```

37.5.3 Validate when decoding

Casting integers to an enum can create values that are not one of the named enumerators. Instead, validate input and reject unknown values.

```
#include <cstdint>
#include <optional>

enum class Mode : std::uint8_t { fast = 1, safe = 2 };

std::optional<Mode> try_decode(std::uint8_t raw) {
    switch (raw) {
    case 1: return Mode::fast;
    case 2: return Mode::safe;
    default: return std::nullopt;
    }
}
```

37.6 Compatibility in expressions and comparisons

37.6.1 Comparisons are type-safe with `enum class`

Scoped enums support comparisons with values of the *same* enum type. They do not compare with integers without explicit conversion.

```
enum class Color { red, blue };

bool ok(Color c) {
    return c == Color::red; // fine
```

```
    // return c == 0;        // error
}
```

37.6.2 Arithmetic is not defined for `enum class` unless you define it

This is usually desirable. If you need iteration or flag operations, define explicit helpers or operators (Core Guidelines Enum.4).

```
#include <type_traits>

enum class Day { mon, tue, wed, thu, fri, sat, sun };

constexpr Day next_day(Day d) {
    using U = std::underlying_type_t<Day>;
    return (d == Day::sun)
        ? Day::mon
        : static_cast<Day>(static_cast<U>(d) + 1);
}
```

37.7 Overload resolution and API compatibility

37.7.1 Problem: unscoped enums can bind to unintended overloads

Because unscoped enums can convert to integers, they can unexpectedly call integer overloads.

```
void log(int);
void log(const char*);

enum Level { low = 0, high = 1 };

void demo() {
    log(high); // calls log(int) (works but may hide intent)
}
```

37.7.2 Solution: use `enum class` and take the enum type in APIs

This prevents accidental binding and communicates the valid domain.

```
enum class Level { low, high };

void log(Level);

void demo() {
    log(Level::high); // explicit, type-safe
}
```

37.8 Interoperability patterns (when compatibility is required)

37.8.1 Define explicit mapping functions

If you must map between an enum and protocol integers, provide dedicated encode/decode functions. This centralizes compatibility and validation.

```
#include <cstdint>
#include <optional>
#include <utility>

enum class Status : std::uint8_t { ok = 0, failed = 1, timeout = 2 };

constexpr std::uint8_t encode(Status s) noexcept {
    return std::to_underlying(s); // C++23
}

std::optional<Status> decode(std::uint8_t raw) noexcept {
    switch (raw) {
```

```
    case 0: return Status::ok;
    case 1: return Status::failed;
    case 2: return Status::timeout;
    default: return std::nullopt;
    }
}
```

37.8.2 Avoid reintroducing implicit compatibility via helper macros

Avoid macros that "cast away" enum typing or that define global constants. Prefer typed functions and explicit conversions (Core Guidelines Enum.1).

37.9 Design rules for compatibility that stays safe

37.9.1 Use `enum class` for domain separation

Treat each enum as a separate domain type. This prevents accidental compatibility.

37.9.2 Keep explicit values and underlying types minimal

Specify underlying type only when necessary (forward declarations, ABI, protocols) and enumerator values only when necessary (flags, protocols).

37.9.3 Prefer return/parameter types that preserve domain meaning

Avoid APIs that take raw integers where an enum expresses the domain. Use the enum type in the public interface, and convert at the edges.

37.10 Checklist

- Prefer `enum class` to prevent accidental compatibility with integers and other enums.
- Do not treat "same underlying type" as semantic compatibility.
- Convert explicitly at boundaries; validate when decoding from integers.
- Use `std::to_underlying` (C++23) or a safe helper for encoding.
- Define explicit operations (iteration, flags) rather than relying on implicit integer behavior.
- Avoid macro-based constants; prefer enums or typed constants.

Part VI

Resource Management (R)

Chapter 38

Resource Management (R.1–R.17)

38.1 Purpose and scope

A **resource** is anything that must be acquired and then released (explicitly or implicitly): dynamic memory, file descriptors/handles, sockets, locks, GUI handles, database connections, GPU buffers, etc. Resource rules aim to:

- prevent leaks (never losing the ability to release a resource),
- prevent double-release (releasing the same resource twice),
- make ownership explicit (who is responsible for release),
- make cleanup correct on *all* paths, especially when exceptions occur.

The core technique is **RAII** (Resource Acquisition Is Initialization): tie resource lifetime to object lifetime so constructors acquire and destructors release.

38.2 Rule map for this chapter (R.1–R.17)

The ISO C++ Core Guidelines numbering for resource rules contains intentional gaps. The rules that exist in the current guideline set and fall within **R.1–R.15** are covered explicitly below:

- R.1–R.6: general resource management and ownership expression.
- R.10–R.15: allocation/deallocation and safe array-view interfaces.

About R.7–R.9 and R.16–R.17

In the current guideline set, these labels are not assigned to resource rules. This book chapter keeps the requested chapter title range (R.1–R.17) and uses the missing labels as **reserved slots** for future guideline expansions or for project-specific addenda (if you choose to add them later).

38.3 R.1: Manage resources automatically using RAII

38.3.1 Rule

Manage resources automatically using resource handles and RAII (Resource Acquisition Is Initialization).

38.3.2 Why

Manual acquire/release is fragile. Every early return, exception, and conditional path becomes a leak or double-free risk. RAII uses the language rule that destructors run automatically when objects leave scope.

38.3.3 What to do

- Wrap each acquire/release pair in a type whose destructor releases the resource.
- Prefer standard RAII types first:
  - memory: `std::vector`, `std::string`, `std::unique_ptr`, `std::shared_ptr`
  - locks: `std::lock_guard`, `std::unique_lock`
  - file streams: `std::ifstream`, `std::ofstream`, `std::fstream`
- For non-standard resources (e.g., OS handles), build a small handle wrapper with a destructor.

38.3.4 Example: manual cleanup (bad) vs RAII (good)

```
// BAD: fragile manual cleanup
void send_raw(int* p, int n) {
    auto h = open_port();
    my_mutex.lock();

    // ...
    transmit(h, p, n); // may throw
    // ...

    my_mutex.unlock();
    close_port(h);
    delete[] p;
}
```

```
// GOOD: RAII makes cleanup correct on all paths
class Port {
    PortHandle h_;
public:
```

```
    Port() : h_{open_port()} {}
    ~Port() { close_port(h_); }

    Port(const Port&) = delete;
    Port& operator=(const Port&) = delete;

    PortHandle get() const noexcept { return h_; }
};

void send_raii(std::unique_ptr<int[]> p, int n) {
    Port port;
    std::lock_guard<std::mutex> lock{my_mutex};

    // ...
    transmit(port.get(), p.get(), n); // may throw
    // ...
} // lock released, port closed, memory freed automatically
```

38.3.5 Practical guidance

- Keep destructors **non-throwing**. If cleanup can fail (e.g., close errors), decide on a policy (log, store status elsewhere, fail fast) rather than throwing from a destructor.
- Prefer small, local scopes: create and release resources as close as possible to their use.

38.4 R.2: In interfaces, use raw pointers to denote individual objects (only)

38.4.1 Rule

In interfaces, use raw pointers to denote *individual objects* (only), not arrays.

38.4.2 Why

A raw pointer carries no size information; using it for arrays invites out-of-bounds access. Arrays are better represented by containers (owning) or views (non-owning).

38.4.3 What to do

- Use `T*` (or `T&`) only when you mean a single object.
- Use `std::span<T>` for a non-owning view of a contiguous sequence.
- Use `std::vector<T>` (or `std::array<T,N>`) for ownership.

38.4.4 Examples

```
// BAD: pointer + count is easy to misuse
void f(int* p, int n) {
    p[2] = 7; // no guarantee p has 3 elements
}

// GOOD: span carries size
#include <span>

void g(std::span<int> s) {
    if (s.size() > 2) {
        s[2] = 7;
    }
}
```

38.4.5 Notes

- Nullability: if `nullptr` is a valid value, a pointer communicates that possibility.

- C-style strings are a special case: if you rely on zero-termination, use a dedicated type or convention (e.g., `zstring` in the Guidelines Support Library context).

38.5 R.3: A raw pointer (`T*`) is non-owning

38.5.1 Rule

A raw pointer (`T*`) is non-owning.

38.5.2 Why

Most raw pointers in modern C++ code are observers. If ownership is hidden in raw pointers, tools and humans cannot reliably determine who must delete.

38.5.3 What to do

- Treat `T*` as an observer by default.
- Express ownership via RAII owners: `std::unique_ptr<T>`, `std::shared_ptr<T>`, or an owning container.
- If you must use an owning raw pointer in legacy or ABI-bound code, document it strongly and isolate it.
- In guideline-oriented code bases, consider tagging ownership explicitly (e.g., `owner<T*>` as a documentation/type-annotation mechanism where available).

38.5.4 Example

```
// BAD: owning raw pointer (unclear and leak-prone)
int* p = new int{7};
```

```
// GOOD: explicit ownership
auto q = std::make_unique<int>(7);
```

38.6 R.4: A raw reference (T&) is non-owning

38.6.1 Rule

A raw reference (T&) is non-owning.

38.6.2 Why

References do not express ownership transfer; they express access to an existing object with an assumed lifetime managed elsewhere.

38.6.3 What to do

- Use T& (or const T&) to express a required, non-null object.
- Use T* when optionality (nullptr) is meaningful.
- Do not return references or pointers to objects that will not outlive the reference (avoid dangling).

38.6.4 Example

```
struct Config { int level; };

void apply(const Config& cfg); // non-owning, cannot be null

void maybe_apply(const Config* cfg) { // optional
```

```
    if (cfg) apply(*cfg);
}
```

38.7 R.5: Prefer scoped objects; do not heap-allocate unnecessarily

38.7.1 Rule

Prefer scoped objects; do not heap-allocate unnecessarily.

38.7.2 Why

Automatic storage (local variables) is simpler, faster to allocate, and easier to reason about. Heap allocation increases failure modes, ownership complexity, and fragmentation concerns.

38.7.3 What to do

- Prefer values and local objects.
- Use dynamic allocation when you truly need:
  - dynamic lifetime (must outlive the current scope),
  - polymorphic object lifetime management,
  - large objects that must not live on the stack (rare, and design-specific),
  - shared ownership semantics (also rare; prefer single ownership).
- If you do allocate, allocate into an owner immediately (see R.12).

38.7.4 Example

```
// BAD: needless heap allocation
auto p = std::make_unique<std::vector<int>>(1000);

// GOOD: just a local value
std::vector<int> v(1000);
```

38.8 R.6: Avoid non-const global variables

38.8.1 Rule

Avoid non-const global variables.

38.8.2 Why

Non-const globals are a form of shared mutable state: they complicate reasoning, create initialization-order hazards, and make lifetime and ownership unclear (especially across translation units). They also inhibit testability and can introduce hidden coupling.

38.8.3 What to do

- Prefer function parameters, returned values, or objects with well-defined lifetimes.
- If a single shared instance is required, prefer controlled access (e.g., create it in `main` and pass references).
- If you need global constants, make them `constexpr` or otherwise immutable.

38.8.4 Example

```
// BAD: global mutable state
std::vector<int> cache;

// BETTER: controlled lifetime in main
int main() {
    std::vector<int> cache;
    run(cache);
}
```

38.9 R.10: Avoid `malloc()` and `free()`

38.9.1 Rule

Avoid `malloc()` and `free()`.

38.9.2 Why

`malloc` does not call constructors and `free` does not call destructors. Mixing `malloc/free` with C++ object lifetimes leads to undefined behavior for non-trivial types and makes exception safety harder.

38.9.3 What to do

- Prefer `new` via RAII owners, or better: containers and factory functions.
- For dynamic arrays, prefer `std::vector<T>` or `std::unique_ptr<T[]>`.
- When interfacing with C libraries that require `malloc/free`, wrap the allocation in a RAII handle with a custom deleter.

38.9.4 Example: wrapping `malloc` with RAII

```
#include <cstdlib>
#include <memory>

struct FreeDeleter {
    void operator()(void* p) const noexcept { std::free(p); }
};

using malloc_ptr = std::unique_ptr<void, FreeDeleter>;

malloc_ptr make_buffer(std::size_t n) {
    if (void* p = std::malloc(n)) return malloc_ptr{p};
    throw std::bad_alloc{};
}
```

38.10 R.11: Avoid calling `new` and `delete` explicitly

38.10.1 Rule

Avoid calling `new` and `delete` explicitly.

38.10.2 Why

Manual `new`/`delete` is a common source of leaks, double deletes, and exception-safety bugs. Using owners and factories makes ownership explicit and cleanup automatic.

38.10.3 What to do

- Prefer:
  - `std::make_unique<T>(...)` instead of `new T(...)`

  - `std::make_shared<T>(...)` instead of `shared_ptr<T>(new T(...))`

- Prefer owning containers (e.g., `std::vector`) when managing sequences.

38.10.4 Example

```
// BAD
auto p = std::unique_ptr<Foo>(new Foo(1, 2, 3));

// GOOD
auto q = std::make_unique<Foo>(1, 2, 3);
```

38.11 R.12: Immediately give the result of an explicit allocation to a manager object

38.11.1 Rule

Immediately give the result of an explicit resource allocation to a manager object.

38.11.2 Why

A raw pointer returned from allocation is vulnerable: any exception thrown before it is stored in an owner leaks the resource.

38.11.3 What to do

- Construct owners directly from allocation results.

- Prefer factory functions (`make_unique`, `make_shared`) that never expose the raw pointer at all.

- For non-memory resources, wrap immediately in a handle type.

38.11.4 Example

```
// Risky if something throws before ownership is established
T* raw = new T{};
do_something(); // may throw
std::unique_ptr<T> p{raw}; // too late if exception happened

// Better: establish ownership immediately
auto p2 = std::make_unique<T>();
do_something(); // safe: p2 will clean up on exception
```

38.12 R.13: Perform at most one explicit resource allocation in a single expression

38.12.1 Rule

Perform at most one explicit resource allocation in a single expression statement.

38.12.2 Why

Complex expressions with multiple allocations can create subtle exception-safety problems due to evaluation order and intermediate temporaries. Even when modern C++ sequencing rules help, the guideline favors clarity and robust patterns.

38.12.3 What to do

- Split complex expressions.

- Prefer `make_unique`/`make_shared` and local variables.
- Make evaluation order obvious.

38.12.4 Example

```
// LESS CLEAR: multiple allocations/operations entangled
f(std::shared_ptr<X>(new X), std::shared_ptr<Y>(new Y));

// CLEARER: separate and use factories
auto x = std::make_shared<X>();
auto y = std::make_shared<Y>();
f(x, y);
```

38.13 R.14: Avoid `[]` parameters; prefer `span`

38.13.1 Rule

Avoid `[]` parameters; prefer `std::span`.

38.13.2 Why

Array parameters decay to pointers and lose size information. A span is a non-owning view that carries bounds and supports safer iteration.

38.13.3 What to do

- Prefer `std::span<T>` or `std::span<const T>`.
- For fixed-size arrays, accept `std::array<T,N>&` or a span.
- For ownership, use containers (e.g., `std::vector`).

38.13.4 Example

```
#include <span>
#include <cstddef>

void sum_to(std::span<const int> in, std::span<int> out) {
    int acc = 0;
    for (std::size_t i = 0; i < in.size(); ++i) {
        acc += in[i];
        out[i] = acc;
    }
}
```

38.14 R.15: Always overload matched allocation/deallocation pairs

38.14.1 Rule

Always overload matched allocation/deallocation pairs.

38.14.2 Why

If you provide a custom allocation function, you must provide the corresponding deallocation function so that the runtime can correctly free memory allocated by the matching allocator. Mismatches are undefined behavior and can corrupt heaps.

38.14.3 What to do

- If you overload `operator new`, also overload the matching `operator delete`.
- If you provide `operator new[]`, also provide `operator delete[]`.

- Include *sized delete* overloads where appropriate for your targeted language modes and platforms.

38.14.4 Example

```
#include <cstddef>
#include <new>

struct PoolAllocated {
    static void* operator new(std::size_t n);
    static void operator delete(void* p) noexcept;

    static void* operator new[](std::size_t n);
    static void operator delete[](void* p) noexcept;
};
```

38.15 R.16 (reserved): Unassigned in current guideline set

This label is currently **unassigned** in the Resource Management section of the ISO C++ Core Guidelines. In this book, it is reserved for future guideline updates or for an optional project-specific rule you may add later (for example, a dedicated rule about OS-handle wrappers or resource-return APIs).

38.16 R.17 (reserved): Unassigned in current guideline set

This label is currently **unassigned** in the Resource Management section of the ISO C++ Core Guidelines. In this book, it is reserved for future guideline updates or for an optional project-specific rule you may add later.

38.17 Common patterns and checklists (practical summary)

38.17.1 Ownership and lifetimes checklist

- **Default:** raw pointers/references are *non-owning observers*.
- **Single owner:** use `std::unique_ptr<T>` or an owning container.
- **Shared ownership:** use `std::shared_ptr<T>` only when multiple independent owners are required.
- **No arrays via raw pointers:** use `std::span` for views and `std::vector`/`std::array` for ownership.
- **No manual delete:** allocate into owners immediately; prefer factory functions.

38.17.2 Exception safety checklist

- Ensure every resource is owned by an object whose destructor releases it.
- Make destructors non-throwing; handle cleanup failure via policy (log, store state, fail fast).
- Avoid complex expressions that intertwine multiple allocations; split for clarity.

Chapter 39

RAII (Resource Acquisition Is Initialization)

39.1 What RAII means in modern C++

39.1.1 Definition

RAII (Resource Acquisition Is Initialization) is a design technique where a resource is:

- **acquired** and made ready for use during the **construction** of an object, and
- **released** during the **destruction** of that object.

The key idea is that C++ guarantees that destructors for fully constructed objects are invoked automatically when those objects leave scope (including during stack unwinding caused by exceptions). This makes cleanup reliable on every control-flow path.

39.1.2 Resources covered by RAII

RAII is not only about dynamic memory. It applies to any limited or reversible capability:

- Dynamic memory (heap allocations)
- File handles and file descriptors
- Sockets and network connections
- Mutexes, locks, semaphores, and other synchronization primitives
- Database connections and transactions
- OS objects (Windows `HANDLE`, POSIX file descriptors, shared memory)
- GPU resources (buffers, textures, command queues) when wrapped properly

39.1.3 Relationship to the Core Guidelines (Resource Management)

RAII is the central mechanism behind the Core Guidelines resource rules, especially:

- **R.1**: Manage resources automatically using resource handles and RAII.
- **R.10–R.14**: Avoid raw allocation interfaces; prefer safe owners and views.

39.2 Why RAII is the default resource-management model

39.2.1 The problem with manual cleanup

Manual acquire/release code is fragile:

- Every `return`, `break`, `continue`, and exception path must release resources.

- Any forgotten release becomes a leak; any repeated release becomes undefined behavior (or worse).

- Error paths tend to grow, becoming hard to test and maintain.

39.2.2 RAII advantages

RAII provides:

- **Leak freedom by construction** (when used consistently).

- **Exception safety**: cleanup occurs automatically during unwinding.

- **Local reasoning**: resource lifetime matches object lifetime.

- **Composability**: multiple resources can be acquired in multiple objects; destruction happens in reverse construction order.

39.3 The RAII contract: invariants and destructor rules

39.3.1 Constructor establishes the invariant

A good RAII type makes a strong promise:

- After construction completes successfully, the object is in a valid state and owns (or manages) its resource.

- If construction fails (throws), no resource is leaked.

39.3.2 Destructor releases the resource and must not throw

In modern C++ practice, destructors should **not allow exceptions to escape**. If a destructor throws while another exception is already being processed, the program will typically terminate.
Practical policy:

- Make destructors `noexcept` (explicitly or implicitly).
- If cleanup can fail (e.g., closing a file handle with an error code), handle it inside the destructor without throwing:
  - log it,
  - store status in an external error channel,
  - provide an explicit `close()` member that can report failure, while the destructor performs best-effort cleanup.

39.4 A minimal RAII wrapper for non-memory resources

39.4.1 Design goals

A resource-handle wrapper should:

- clearly own exactly one resource (or represent “no resource”),
- release it exactly once,
- be **non-copyable** (to avoid double-release) and **movable** (to allow transfer of ownership),
- expose safe access to the underlying handle when needed.

39.4.2 Example: generic handle wrapper pattern

```
#include <utility>

template<class Handle, class Invalid, class Close>
class unique_handle {
public:
    unique_handle() noexcept : h_(Invalid::value()) {}
    explicit unique_handle(Handle h) noexcept : h_(h) {}

    ~unique_handle() noexcept { reset(); }

    unique_handle(const unique_handle&) = delete;
    unique_handle& operator=(const unique_handle&) = delete;

    unique_handle(unique_handle&& other) noexcept : h_(other.release()) {}
    unique_handle& operator=(unique_handle&& other) noexcept {
        if (this != &other) reset(other.release());
        return *this;
    }

    Handle get() const noexcept { return h_; }
    explicit operator bool() const noexcept { return h_ != Invalid::value(); }

    Handle release() noexcept {
        Handle tmp = h_;
        h_ = Invalid::value();
        return tmp;
    }

    void reset(Handle new_h = Invalid::value()) noexcept {
        if (h_ != Invalid::value()) Close{}(h_);
        h_ = new_h;
```

```
    }

private:
    Handle h_;
};
```

39.4.3 Example: POSIX file descriptor

```
#include <unistd.h>   // close
#include <fcntl.h>    // open
#include <stdexcept>

struct fd_invalid { static int value() noexcept { return -1; } };
struct fd_close   { void operator()(int fd) const noexcept { ::close(fd); } };

using unique_fd = unique_handle<int, fd_invalid, fd_close>;

unique_fd open_readonly(const char* path) {
    int fd = ::open(path, O_RDONLY);
    if (fd < 0) throw std::runtime_error("open failed");
    return unique_fd{fd};
}
```

39.5 RAII for memory: prefer standard owners and containers

39.5.1 The preferred owning tools

For memory and object lifetime, prefer standard library owners:

- `std::vector<T>` for owning dynamic sequences.

- `std::string` for owning text.
- `std::unique_ptr<T>` for exclusive ownership of a single object.
- `std::unique_ptr<T[]>` for exclusive ownership of a dynamically allocated array (when `vector` is not suitable).
- `std::shared_ptr<T>` for shared ownership (use only when truly needed).

39.5.2 Prefer factory functions

Factory functions reduce error-prone explicit allocation and make ownership immediate:

- `std::make_unique<T>(...)`
- `std::make_shared<T>(...)`

```
#include <memory>
#include <vector>

struct Node { int v; };

auto p = std::make_unique<Node>(Node{42});     // exclusive ownership
auto s = std::make_shared<Node>(Node{7});      // shared ownership (when justified)

std::vector<int> data(1000);                   // owning sequence
```

39.6 RAII for synchronization: scoped locking

39.6.1 The standard pattern

Synchronization primitives are resources: a locked mutex must be unlocked. C++ provides lock wrappers that implement RAII and guarantee unlocking when leaving scope.

39.6.2 Example: `std::lock_guard`

```cpp
#include <mutex>

std::mutex m;
int shared_state = 0;

void update() {
    std::lock_guard<std::mutex> lock(m); // locks here
    ++shared_state;
} // unlocks here automatically (even if exceptions occur)
```

39.6.3 When to prefer `std::unique_lock`

Use `std::unique_lock` when you need:

- deferred locking,
- manual unlock/lock within a scope,
- ownership transfer (move) of the lock,
- condition variables (common pairing).

39.7 RAII for files and streams

39.7.1 Streams are RAII types

Standard file streams acquire OS resources (file handles) and release them in their destructors.

39.7.2 Example: safe file reading with exceptions enabled

```cpp
#include <fstream>
```

```cpp
#include <string>
#include <stdexcept>

std::string read_all(const char* path) {
    std::ifstream in(path, std::ios::binary);
    if (!in) throw std::runtime_error("cannot open file");

    in.exceptions(std::ios::badbit); // choose a policy (example)

    std::string content((std::istreambuf_iterator<char>(in)),
                        std::istreambuf_iterator<char>());
    return content; // file closes automatically when 'in' is destroyed
}
```

39.7.3 Policy note

Some codebases prefer not to use stream exceptions and instead check status flags. RAII still applies either way: the stream object owns the file handle and releases it automatically.

39.8 Views vs owners: `std::span` completes the RAII picture

RAII expresses ownership and cleanup. Many functions should **not** own; they should observe safely.

39.8.1 Preferred interfaces

- Owning: `std::vector<T>`, `std::array<T,N>`
- Non-owning view with size: `std::span<T>`, `std::span<const T>`
- Single object, non-owning: `T&` (required), `T*` (optional)

39.8.2 Example: API that avoids pointer + length

```
#include <span>
#include <cstdint>

std::uint32_t checksum(std::span<const std::byte> bytes);

void process(std::span<int> values) {
    for (int& x : values) x *= 2;
}
```

39.9 Move semantics: the engine of unique ownership

39.9.1 Why move matters for RAII

Many RAII types represent unique ownership (a single file descriptor, a single mutex lock, a single OS handle). Copying would duplicate ownership and cause double release. Moving transfers ownership safely.

39.9.2 Rule of zero, rule of five, and RAII

- **Rule of zero**: Prefer using standard RAII members so you do not write custom destructor/copy/move at all.
- **Rule of five**: If your type directly manages a resource, you typically define or delete: destructor, copy constructor, copy assignment, move constructor, move assignment.

39.9.3 Example: move-only RAII type

```
class Socket {
public:
```

```
    Socket();               // acquires resource
    ~Socket() noexcept;     // releases resource

    Socket(const Socket&) = delete;
    Socket& operator=(const Socket&) = delete;

    Socket(Socket&&) noexcept;
    Socket& operator=(Socket&&) noexcept;

    void send(const void* data, std::size_t n);

private:
    int fd_ = -1;
};
```

39.10 Custom deleters: RAII for “foreign” resources

Sometimes a library requires a specific release function (not `delete`). Smart pointers support this via custom deleters.

39.10.1 Example: RAII with custom deleter

```
#include <memory>

// Suppose a C library returns a handle that must be freed by lib_free().
extern "C" {
    struct lib_obj;
    lib_obj* lib_create();
    void lib_free(lib_obj*);
}

struct lib_deleter {
```

```
    void operator()(lib_obj* p) const noexcept { lib_free(p); }
};

using lib_ptr = std::unique_ptr<lib_obj, lib_deleter>;

lib_ptr make_lib_obj() {
    lib_obj* raw = lib_create();
    if (!raw) throw std::bad_alloc{};
    return lib_ptr(raw); // ownership is immediate
}
```

39.11 Scope guards: RAII for “do this when leaving scope”

Sometimes cleanup is not tied to a single resource object but to an action (rollback, restore state, unlock a non-standard lock, etc.). A **scope guard** is a small RAII object that runs a callable in its destructor.

39.11.1 Example: minimal scope guard

```
#include <utility>

template<class F>
class scope_exit {
public:
    explicit scope_exit(F f) noexcept : f_(std::move(f)), active_(true) {}
    ~scope_exit() noexcept { if (active_) f_(); }

    scope_exit(const scope_exit&) = delete;
    scope_exit& operator=(const scope_exit&) = delete;

    scope_exit(scope_exit&& other) noexcept
        : f_(std::move(other.f_)), active_(other.active_) {
```

```
        other.active_ = false;
    }

    void release() noexcept { active_ = false; }

private:
    F f_;
    bool active_;
};

template<class F>
scope_exit<F> make_scope_exit(F f) noexcept { return scope_exit<F>(std::move(f)); }
```

39.11.2 Example: transactional rollback

```
void update_record(Database& db) {
    db.begin();
    auto rollback = make_scope_exit([&] { db.rollback(); });

    db.write_changes();
    db.commit();
    rollback.release(); // commit succeeded, do not rollback
}
```

39.12 Common RAII pitfalls and how to avoid them

39.12.1 Pitfall 1: throwing destructors

Rule: Destructors should not throw. If an error must be reported, provide an explicit operation that can report failure, and keep the destructor best-effort and non-throwing.

39.12.2 Pitfall 2: shared ownership when you really have a single owner

`std::shared_ptr` is useful but easy to overuse. Prefer `unique_ptr` when there is a single owner and pass raw pointers/references or spans to observers.

39.12.3 Pitfall 3: reference cycles with `shared_ptr`

Cycles prevent cleanup. Break cycles with `std::weak_ptr` when using shared ownership graphs.

39.12.4 Pitfall 4: returning views to objects that will not outlive the view

Never return `std::span`, `T*`, or `T&` referencing local objects. Ensure the referenced storage outlives the view.

39.12.5 Pitfall 5: confusing "scope-based" lifetime with "resource" lifetime

RAII ties cleanup to object lifetime. If you place an RAII owner in a longer-lived object (e.g., heap, static storage), the resource lifetime is longer too. This is correct, but it must be intentional and visible in design.

39.13 Practical design checklist for RAII types

1. **State** clearly whether the type owns a resource.
2. Acquire in the constructor (or factory) and release in the destructor.
3. Make destruction **non-throwing**.
4. Delete copying if copying would duplicate ownership.

5. Provide move operations to support ownership transfer.
6. Expose the underlying handle carefully (`get()`), and provide `release()` only when needed.
7. Prefer using standard RAII members so you can follow the **rule of zero**.

39.14 Summary

RAII is the fundamental C++ technique for correct, maintainable, exception-safe resource management. It works because C++ provides deterministic destruction and well-defined object lifetime rules. In guideline-oriented code, RAII is the default: express ownership with RAII owners, express non-ownership with safe views, and design interfaces so resource cleanup is automatic and unavoidable.

Chapter 40

Resource Ownership

40.1 What “ownership” means

40.1.1 Definition

A program **owns** a resource when it is responsible for eventually releasing it exactly once. Ownership answers one question clearly:

> *Who must perform the cleanup, and when?*

In C++, correct ownership is primarily expressed by **RAII** types: the owner acquires the resource in construction (or a factory) and releases it in its destructor. This ensures cleanup happens on every path, including exceptions.

40.1.2 Resources in scope

Ownership rules apply to more than dynamic memory:

- memory (objects, arrays, buffers),

- OS handles (file descriptors, Windows `HANDLE`, sockets),
- locks (mutexes, semaphores),
- transactions (database, file-system operations),
- any acquire/release pair that must be balanced.

40.2 Owner vs observer (the central distinction)

40.2.1 Owners

An **owner** is a value that controls a resource lifetime and performs cleanup automatically:

- `std::unique_ptr<T>` (single owner)
- `std::shared_ptr<T>` (shared owners, reference counted)
- `std::vector<T>`, `std::string`, `std::array<T,N>` (owning containers)
- custom RAII handle types for non-standard resources

40.2.2 Observers

An **observer** refers to a resource owned elsewhere and does not release it:

- `T&` (non-null observer of a single object)
- `T*` (nullable observer of a single object)
- `std::span<T>` (non-owning view of a contiguous sequence with size)
- `std::string_view` (non-owning view of text)
- iterators, references, and pointers into containers (while valid)

40.2.3 Core Guidelines rule summary

The Core Guidelines state, in the Resource Management section:

- **R.3:** A raw pointer (`T*`) is **non-owning**.
- **R.4:** A raw reference (`T&`) is **non-owning**.

This is a practical convention that makes code review and tooling reliable: if a raw pointer meant ownership sometimes, you could not detect leaks and double deletes systematically.

40.3 Three ownership models you must choose from

40.3.1 Model 1: Single ownership

Exactly one entity is responsible for cleanup.

- Prefer this model by default.
- Implementation tools: `std::unique_ptr<T>` and owning containers.
- Ownership transfer is done by **moving** the owner.

40.3.2 Model 2: Shared ownership

Multiple entities are owners; resource is released when the last owner is destroyed.

- Use only when independent lifetimes require it.
- Implementation tool: `std::shared_ptr<T>`.
- Beware of cycles; break cycles with `std::weak_ptr`.

40.3.3 Model 3: No ownership in the interface (borrowed use)

The interface receives access without taking responsibility for cleanup.

- This is the most common and most scalable model for APIs.
- Use references/pointers/spans/views to express borrowing.

40.4 How to express ownership in APIs (recommended patterns)

40.4.1 Passing a single object

40.4.1.1 Borrowing without ownership transfer

```
struct Image { /* ... */ };

void draw(const Image& img);     // required, non-owning, not null
void maybe_draw(const Image* p); // optional, non-owning, may be null
```

40.4.1.2 Transferring single ownership

Pass an owner type **by value** and move it:

```
#include <memory>

struct Widget { /* ... */ };

void set_widget(std::unique_ptr<Widget> w); // takes ownership

void caller() {
    auto w = std::make_unique<Widget>();
```

```
    set_widget(std::move(w)); // ownership transferred
} // w is now empty
```

40.4.1.3 Sharing ownership

Only when the callee must keep ownership beyond the call and multiple parties must also own:

```
#include <memory>

struct Session { /* ... */ };

void register_session(std::shared_ptr<Session> s); // shares ownership

void use(const std::shared_ptr<Session>& s); // sharing already exists; avoid increments
```

Practical note If the function merely *uses* the object during the call, prefer `T&` or `T*`, not `shared_ptr`. This keeps ownership out of most interfaces and reduces coupling.

40.4.2 Passing sequences (arrays, buffers)

40.4.2.1 Borrowing a sequence

Prefer `std::span` for contiguous ranges:

```
#include <span>
#include <cstddef>

void scale(std::span<float> v, float k) {
    for (float& x : v) x *= k;
}
```

40.4.2.2 Owning a sequence

Prefer containers:

```cpp
#include <vector>

std::vector<int> make_data(std::size_t n) {
    return std::vector<int>(n); // ownership returned by value
}
```

40.4.3 Returning ownership

40.4.3.1 Return by value (preferred)

For most types, returning by value is correct, efficient, and clear:

```cpp
#include <string>

std::string read_name();
```

40.4.3.2 Return an owner

Return `unique_ptr` when the result is polymorphic or must be dynamically allocated:

```cpp
#include <memory>

struct Base { virtual ~Base() = default; };
struct Derived : Base { /* ... */ };

std::unique_ptr<Base> make_polymorphic() {
    return std::make_unique<Derived>();
}
```

40.4.3.3 Return a view only when lifetime is guaranteed

Returning a view is safe only if the referenced storage outlives the view:

```cpp
#include <string_view>
```

```cpp
#include <string>

class Table {
public:
    std::string_view name() const noexcept { return name_; } // view into member
private:
    std::string name_;
};
```

40.5 Ownership in class design

40.5.1 Rule of zero: let members own

The best ownership design is often: **store owning RAII members** and avoid writing custom special members.

```cpp
#include <vector>
#include <string>

class Log {
public:
    void add(std::string s) { entries_.push_back(std::move(s)); }
private:
    std::vector<std::string> entries_; // owns its memory
}; // no destructor/copy/move needed (rule of zero)
```

40.5.2 When you manage a raw handle directly (move-only RAII)

For non-standard resources, make the type:

- non-copyable (avoid double-close),
- movable (allow transfer),

- destructible without throwing (best-effort cleanup).

```
#include <utility>

struct HandleOps {
    static int invalid() noexcept { return -1; }
    static void close(int) noexcept; // provided elsewhere
};

class UniqueHandle {
public:
    UniqueHandle() noexcept : h_(HandleOps::invalid()) {}
    explicit UniqueHandle(int h) noexcept : h_(h) {}
    ~UniqueHandle() noexcept { reset(); }

    UniqueHandle(const UniqueHandle&) = delete;
    UniqueHandle& operator=(const UniqueHandle&) = delete;

    UniqueHandle(UniqueHandle&& other) noexcept : h_(other.release()) {}
    UniqueHandle& operator=(UniqueHandle&& other) noexcept {
        if (this != &other) reset(other.release());
        return *this;
    }

    int get() const noexcept { return h_; }
    explicit operator bool() const noexcept { return h_ != HandleOps::invalid(); }

    int release() noexcept {
        int tmp = h_;
        h_ = HandleOps::invalid();
        return tmp;
    }

    void reset(int nh = HandleOps::invalid()) noexcept {
```

```
        if (h_ != HandleOps::invalid()) HandleOps::close(h_);
        h_ = nh;
    }

private:
    int h_;
};
```

40.6 Avoiding dangerous ownership patterns

40.6.1 Never transfer ownership by a raw pointer

Transferring ownership by `T*` is ambiguous and commonly leads to leaks and double deletes. If ownership must transfer, use `std::unique_ptr<T>` (or an owning container) in the interface.

40.6.2 Avoid owning raw pointers and owning references

Owning references are especially confusing because a reference cannot be reseated, yet someone must delete the object:

```
// BAD: raw owning reference
int& r = *new int{7};
// ...
delete &r; // fragile, non-idiomatic, error-prone
```

40.6.3 Prefer factories to explicit `new`

Factories both document ownership and provide immediate safety:

```
#include <memory>

// GOOD: no raw pointer escapes
auto p = std::make_unique<int>(42);
```

40.6.4 Shared ownership pitfalls

- **Overuse:** `shared_ptr` spreads ownership decisions everywhere.
- **Cycles:** reference cycles leak by design; break with `weak_ptr`.
- **Hidden costs:** reference counting updates and control-block allocations matter in performance-critical code.

```
#include <memory>

struct Node {
    std::shared_ptr<Node> next; // can form cycles (leak)
    std::weak_ptr<Node>  prev;  // breaks cycles
};
```

40.7 Ownership annotations and the Guidelines Support Library (GSL)

40.7.1 Why annotations exist

In large code bases and when interfacing with legacy APIs, you may encounter pointers that *really are* owners. The Guidelines Support Library provides vocabulary types to make intent explicit for humans and tools (not to change runtime behavior).

40.7.2 Typical vocabulary

Commonly used vocabulary in guideline-oriented projects includes:

- `gsl::owner<T*>` to annotate an owning raw pointer (legacy boundary).
- `gsl::span<T>` as a bounds-safe view (often an alias or close relative of `std::span`).

- `gsl::not_null<T*>` to express non-null pointers explicitly (when used in a project).

```
#include <gsl/gsl>

gsl::owner<int*> legacy_make(); // ownership is explicit in the type

void take(gsl::not_null<int*> p); // cannot be null (by contract)
```

Boundary rule Use `gsl::owner<T*>` only at boundaries where you cannot change the signature to a safer owner (e.g., ABI constraints). Inside modern C++ code, prefer `unique_ptr` and containers.

40.8 A practical checklist for resource ownership

1. Decide the ownership model: **single**, **shared**, or **borrowed**.
2. Express owners using RAII:
    - `unique_ptr` or containers for single ownership,
    - `shared_ptr/weak_ptr` only when required.
3. Express borrowing using:
    - `T&` for required objects,
    - `T*` for optional objects,
    - `std::span` for sequences.
4. Never transfer ownership by raw pointer or raw reference.
5. Establish ownership **immediately** after allocation; prefer `make_unique/make_shared`.
6. Keep destructors non-throwing; handle cleanup failure by policy.

7. Document lifetimes for all views (`span`, `string_view`, pointers) and avoid returning dangling views.

40.9 Summary

Resource ownership is the foundation of safe C++: an owner must release exactly once, and non-owners must never release. The Core Guidelines default convention is that raw pointers and references are **non-owning observers**; ownership is made explicit via RAII owners (containers, `unique_ptr`, handle wrappers) and shared ownership is used only when necessary. Modern interfaces stay stable and safe by keeping ownership out of most function parameters and using views like `std::span` for sequences.

Chapter 41

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

41.1 Purpose of this chapter

Dynamic allocation exists in C++, but modern C++ engineering treats explicit `new` and `delete` as **low-level tools of last resort**. This chapter explains:

- what `new`/`delete` actually do (language rules and failure modes),
- why the Core Guidelines recommend avoiding them in most code,
- the safe, modern replacements (`make_unique`, containers, RAII handles),
- the rare cases where direct allocation is appropriate, and how to do it correctly.

41.2 Core Guidelines viewpoint (Resource Management)

In the ISO C++ Core Guidelines, the key rules for this topic are:

- **R.11:** Avoid calling `new` and `delete` explicitly.

- **R.12:** Immediately give the result of an explicit allocation to a manager object.
- **R.13:** Perform at most one explicit resource allocation in a single expression statement.
- **R.10:** Avoid `malloc`/`free` (related, but important at boundaries).

The design goal is simple: **make ownership explicit and cleanup automatic**.

41.3 What `new` really does

41.3.1 Two operations: allocation + construction

A `new` expression is conceptually:

1. **Allocate raw storage** (typically by calling an `operator new` function).
2. **Construct an object** in that storage (run the constructor).

```
// Conceptual model (not literal implementation)
T* p = static_cast<T*>(::operator new(sizeof(T))); // allocate bytes
try {
    ::new (p) T(args...);                          // construct
} catch (...) {
    ::operator delete(p);                          // deallocate on failure
    throw;
}
```

41.3.2 Failure behavior

- **Normal** `new` throws `std::bad_alloc` on allocation failure.
- `new(std::nothrow)` returns `nullptr` on failure (no exception), but still constructs if allocation succeeds.

```
#include <new>      // std::nothrow
#include <iostream> // example only

int* p = new int(7);                // throws std::bad_alloc on failure
int* q = new (std::nothrow) int(9);  // returns nullptr on failure
```

41.3.3 Array `new`

`new T[n]` allocates enough storage for `n` objects and then constructs `n` objects. If element construction throws partway through, already-constructed elements are destroyed, and the storage is released.

41.4 What `delete` really does

41.4.1 Two operations: destruction + deallocation

A `delete` expression is conceptually:

1. **Destroy the object** (run the destructor).
2. **Deallocate the storage** (typically by calling an `operator delete` function).

```
// Conceptual model (not literal implementation)
p->~T();                // destroy
::operator delete(p);    // deallocate bytes
```

41.4.2 `delete` on null

`delete nullptr;` is a no-op. This is convenient for conditional cleanup, but it is not a substitute for correct ownership design.

41.4.3 Array delete must match array new

`new T[n]` must be paired with `delete[]`, and `new T` must be paired with `delete`. A mismatch is undefined behavior.

```
// BAD: mismatch => undefined behavior
int* a = new int[10];
delete a;        // wrong

// GOOD: correct pairing
int* b = new int[10];
delete[] b;
```

41.4.4 Deleting through a base pointer

If you delete an object through a pointer-to-base, the base type must have a **virtual destructor**, otherwise behavior is undefined.

```
struct Base {
    virtual ~Base() = default; // required for polymorphic deletion
};

struct Derived : Base {
    ~Derived() = default;
};

Base* p = new Derived{};
delete p; // OK
```

41.5 Why explicit `new`/`delete` are discouraged

Explicit `new`/`delete` are common sources of:

- **leaks** (missing `delete` on some path),
- **double deletes** (two owners),
- **mismatched delete** (`delete` vs `delete[]`),
- **dangling pointers** (use-after-free),
- **exception safety bugs** (leak when something throws between allocation and ownership establishment),
- **unclear ownership in interfaces** (reviewers and tools cannot tell who deletes).

The Core Guidelines therefore recommend: express ownership with RAII owners and keep most interfaces **non-owning**.

41.6 Modern replacements (preferred patterns)

41.6.1 Prefer owning containers

If you need a dynamic sequence, `std::vector<T>` is the default owner.

```
#include <vector>

std::vector<int> v(1000); // owns 1000 ints
```

41.6.2 Prefer `std::make_unique` for single objects

```
#include <memory>

struct X { X(int, int) {} };

auto p = std::make_unique<X>(1, 2); // ownership is explicit and immediate
```

41.6.3 Prefer `std::make_shared` when you truly need shared ownership

```
#include <memory>

struct Session {};

auto s = std::make_shared<Session>();
```

41.6.4 Prefer `std::unique_ptr<T[]>` only when you really need a raw array

A dynamic array with unique ownership can be represented as `unique_ptr<T[]>`, but a `vector` is usually better because it carries size and supports richer operations.

```
#include <memory>

auto buf = std::make_unique<unsigned char[]>(4096); // owns a byte array
```

41.7 R.12 and R.13 in practice: exception safety and clarity

41.7.1 R.12: Immediately bind to an owner

```
// BAD: leak if something throws before delete
int* p = new int(7);
do_something();          // may throw
delete p;

// GOOD: establish ownership immediately
#include <memory>

auto p2 = std::make_unique<int>(7);
do_something();          // safe, p2 cleans up automatically
```

41.7.2 R.13: One explicit allocation per statement

Keep allocation separate from other complex work so that ownership and sequencing are obvious.

```
#include <memory>

// LESS CLEAR (and often unnecessary)
f(std::shared_ptr<int>(new int(1)), std::shared_ptr<int>(new int(2)));

// CLEARER
auto a = std::make_shared<int>(1);
auto b = std::make_shared<int>(2);
f(a, b);
```

41.8 When explicit `new`/`delete` can be justified

Explicit allocation is sometimes appropriate:

- **Low-level infrastructure**: custom allocators, memory pools, and frameworks implementing containers.
- **Polymorphic factories** returning `unique_ptr<Base>` where you cannot avoid dynamic allocation (still: prefer `make_unique`).
- **Placement-new scenarios**: constructing objects into pre-allocated storage (e.g., arenas, shared memory).
- **ABI or C-API boundaries**: interacting with libraries that allocate and require custom release functions (wrap with RAII).

Even in these cases, the guideline intent remains: **do not leak raw owning pointers into general code**.

41.9 Placement `new`: construction without allocation

41.9.1 What it is

Placement `new` constructs an object at a given address. It does **not** allocate memory.

```
#include <new>
#include <cstddef>

alignas(int) unsigned char storage[sizeof(int)];
int* p = ::new (static_cast<void*>(storage)) int(42); // construct in place
p->~int(); // manual destruction required
```

41.9.2 Correct use cases

- constructing objects in arenas and custom allocators,
- reconstructing an object in the same storage (with careful lifetime rules),
- interfacing with shared memory or hardware-mapped memory (rare, high-risk).

41.9.3 Pitfall: forgetting explicit destruction

With placement `new`, you must usually call the destructor explicitly (or manage it via a scope guard/RAII wrapper).

41.10 Overloading allocation and deallocation functions

41.10.1 Class-specific `operator new`/`operator delete`

A type can define custom allocation behavior. If you do this, you must provide matching deallocation functions.

```
#include <cstddef>
#include <new>

struct PoolAllocated {
    static void* operator new(std::size_t n);
    static void  operator delete(void* p) noexcept;

    static void* operator new[](std::size_t n);
    static void  operator delete[](void* p) noexcept;
};
```

41.10.2 Sized deallocation and alignment-aware allocation (modern C++)

Modern C++ includes additional forms that an implementation may use:

- **sized delete** overloads: `operator delete(void*, std::size_t)` and `operator delete[](void*, std::size_t)`
- **aligned new/delete** overloads for over-aligned types: `operator new(std::size_t, std::align_val_t)` and corresponding deletes

If you provide custom allocation, consider providing the matching sized and aligned forms when appropriate for your platforms and language modes.

41.11 The `new/delete` taxonomy (quick reference)

41.11.1 `new`

- `new T(args...)`: allocate + construct; throws `std::bad_alloc` on failure
- `new T[n]`: allocate + construct `n` elements

- `new (std::nothrow) T`: returns `nullptr` on allocation failure
- `new (ptr) T`: placement-new; constructs at `ptr`; no allocation

41.11.2 `delete`

- `delete p`: destroy + deallocate a single object
- `delete[] p`: destroy + deallocate an array created by `new[]`
- `delete nullptr`: no-op

41.12 Common correctness rules (non-negotiable)

1. Always match `new` with `delete`, and `new[]` with `delete[]`.
2. Never `delete` the same object twice.
3. Never `delete` memory you did not obtain from a matching `new` (or matching allocation API).
4. If deleting through a base pointer, ensure the base has a virtual destructor.
5. Do not let raw owning pointers escape into general code; wrap immediately in RAII owners.

41.13 How to design code so you almost never write `delete`

41.13.1 Rule of zero

If a class stores resources only in standard RAII members, it typically needs no custom destructor or copy/move operations.

```cpp
#include <vector>
#include <string>

class Report {
public:
    void add(std::string s) { lines_.push_back(std::move(s)); }
private:
    std::vector<std::string> lines_; // owns memory
}; // no manual new/delete anywhere
```

41.13.2 Return-by-value is ownership transfer

Returning an object by value is the simplest ownership model in C++: the caller receives a fully formed value.

```cpp
#include <vector>

std::vector<int> make_data() {
    std::vector<int> v = {1, 2, 3, 4};
    return v; // returned by value; no manual delete
}
```

41.14 Performance note: allocation is expensive, but design comes first

Heap allocation can be slow and can cause fragmentation. If performance matters:

- reduce allocations (reuse buffers, reserve capacity, prefer value semantics),
- use appropriate containers and allocators,

- consider `std::pmr` (polymorphic memory resources) for configurable allocation strategies in allocation-heavy code,
- measure with real profilers before making low-level changes.

Avoid jumping to custom `operator new` unless measurements justify it and the design is stable.

41.15 Summary

`new` and `delete` are correct tools, but they are sharp and easy to misuse. The modern C++ rule is:

- Prefer RAII owners (`vector`, `string`, `unique_ptr`, `shared_ptr`) and factory functions (`make_unique`, `make_shared`).
- If you must allocate explicitly, immediately bind the result to an owner (R.12), keep expressions simple (R.13), and ensure all matching delete rules.
- Treat raw pointers and references as non-owning observers by default (R.3, R.4).

Chapter 42

Avoiding Leaks

42.1 What is a leak (precise definition)

A **leak** occurs when a program *loses the ability to release a resource* that it previously acquired. A leak is not limited to memory. Any resource that must be released can leak, including:

- heap allocations,
- file descriptors / OS handles,
- sockets,
- locks (a lock "leak" becomes a deadlock risk),
- threads that are started but never joined or detached properly,
- GPU resources and other device handles.

42.1.1 Why leaks matter

- Long-running processes (servers, services, GUI applications) accumulate leaks and eventually fail.
- Leaks can degrade performance (more allocations, paging, fragmentation).
- Some leaks are also correctness and security problems (exhausting file descriptors, exhausting handles, or leaving privileged resources open).

42.2 Core Guidelines principles behind leak prevention

The ISO C++ Core Guidelines leak-avoidance strategy is built on a small set of enforceable principles:

- **R.1:** Manage resources automatically using RAII.
- **R.3–R.4:** Raw pointers/references are non-owning (avoid hidden owners).
- **R.10–R.12:** Avoid `malloc/free` and explicit `new/delete` in most code; if you allocate explicitly, bind immediately to a manager.
- **R.14:** Avoid `[]` parameters; prefer `std::span` for non-owning sequences.

The essential idea is:

> *Make ownership explicit and automatic, so cleanup is inevitable.*

42.3 Leak classes (what actually goes wrong)

42.3.1 Class 1: missing cleanup on one control-flow path

Typical causes:

- early `return`,
- error-handling branches,
- exceptions thrown between acquisition and release.

```
// BAD: easy to forget cleanup on a path
void process() {
    int* p = new int[1000];

    if (!ready()) return; // leak

    work(p);
    delete[] p;
}
```

42.3.2 Class 2: unclear ownership (two owners or no owner)

When ownership is unclear, one of two disasters happens:

- nobody deletes (leak),
- two places delete (double-free / undefined behavior).

```
// BAD: raw owning pointer escapes, ownership unclear
int* make() { return new int(7); } // who deletes?

void use() {
    int* p = make();
    // ...
} // likely leak
```

42.3.3 Class 3: mismatched acquire/release APIs

Examples:

- `new[]` paired with `delete`,
- `malloc` paired with `delete`,
- OS handle acquired by one API released incorrectly (or not at all).

```
// BAD: mismatch => undefined behavior
int* p = new int[10];
delete p; // must be delete[]
```

42.3.4 Class 4: cycles in shared ownership

`std::shared_ptr` uses reference counting. Reference cycles do not reach zero, so the objects never get destroyed.

```
#include <memory>

struct Node {
    std::shared_ptr<Node> next; // cycle risk
    std::weak_ptr<Node> prev;   // break the cycle
};
```

42.3.5 Class 5: lifetime errors of views (not a leak, but often related)

Dangling pointers, dangling references, and dangling `std::span`/`std::string_view` are not leaks, but they commonly appear in the same code that mishandles ownership. Clean ownership design eliminates both families of bugs.

42.4 Primary cure: RAII everywhere

42.4.1 Rule: every resource must have an owner object

The most reliable leak prevention technique is to enforce that:

> *Every acquired resource is immediately owned by an RAII object whose destructor releases it.*

42.4.2 Memory resources: use containers and smart pointers

```
#include <memory>
#include <vector>

void good1() {
    std::vector<int> v(1000);           // owns memory
    // ...
} // freed automatically

void good2() {
    auto p = std::make_unique<int[]>(1000); // owns array
    // ...
} // freed automatically
```

42.4.3 Non-memory resources: create a small handle wrapper

```
#include <utility>

// Example pattern: move-only RAII handle
struct HandleOps {
    static int invalid() noexcept { return -1; }
    static void close(int) noexcept; // implemented elsewhere
```

```
};

class UniqueHandle {
public:
    UniqueHandle() noexcept : h_(HandleOps::invalid()) {}
    explicit UniqueHandle(int h) noexcept : h_(h) {}
    ~UniqueHandle() noexcept { reset(); }

    UniqueHandle(const UniqueHandle&) = delete;
    UniqueHandle& operator=(const UniqueHandle&) = delete;

    UniqueHandle(UniqueHandle&& other) noexcept : h_(other.release()) {}
    UniqueHandle& operator=(UniqueHandle&& other) noexcept {
        if (this != &other) reset(other.release());
        return *this;
    }

    int get() const noexcept { return h_; }
    explicit operator bool() const noexcept { return h_ != HandleOps::invalid(); }

    int release() noexcept {
        int tmp = h_;
        h_ = HandleOps::invalid();
        return tmp;
    }

    void reset(int nh = HandleOps::invalid()) noexcept {
        if (h_ != HandleOps::invalid()) HandleOps::close(h_);
        h_ = nh;
    }

private:
    int h_;
```

```
};
```

42.5 Avoid explicit `new`/`delete` in application code

42.5.1 Prefer factories

```
// BAD
auto p = std::unique_ptr<int>(new int(42));

// GOOD
auto q = std::make_unique<int>(42);
```

42.5.2 R.12: if you must allocate explicitly, bind immediately

```
// BAD: leak if g() throws
T* raw = new T{};
g();
delete raw;

// GOOD: ownership established immediately
#include <memory>
auto p = std::make_unique<T>();
g(); // safe
```

42.6 Make ownership explicit in interfaces

42.6.1 Raw pointers are non-owning observers

```
struct Image {};

void render(const Image& img);     // required, non-owning
```

```
void maybe_render(const Image* p); // optional, non-owning
```

42.6.2 Transfer ownership with `unique_ptr`

```
#include <memory>

struct Widget {};

void set_widget(std::unique_ptr<Widget> w); // takes ownership

void caller() {
    auto w = std::make_unique<Widget>();
    set_widget(std::move(w)); // no leak possible
}
```

42.6.3 Use `std::span` for sequences instead of pointer+size

```
#include <span>

void fill(std::span<int> s) {
    for (int& x : s) x = 0;
}
```

42.7 Exception safety: preventing leaks under failure

42.7.1 RAII makes basic exception safety automatic

When resources are owned by stack objects, exceptions cannot bypass cleanup because destruction happens during stack unwinding.

42.7.2 Do not let destructors throw

Destructors should not allow exceptions to escape. If cleanup can fail (e.g., `close()`), choose a policy:

- provide an explicit `close()` or `commit()` that can report failure,
- keep the destructor best-effort and non-throwing.

```
class File {
public:
    void close();             // can report failure (error code or exception)
    ~File() noexcept {        // destructor never throws
        try { close(); } catch (...) { /* log or swallow by policy */ }
    }
};
```

42.7.3 One allocation per statement (clarity and robustness)

Avoid complex expressions that hide allocation and ownership transfer inside argument lists:

```
#include <memory>

// LESS CLEAR
f(std::shared_ptr<X>(new X), std::shared_ptr<Y>(new Y));

// CLEARER
auto x = std::make_shared<X>();
auto y = std::make_shared<Y>();
f(x, y);
```

42.8 Shared ownership: avoiding reference-count cycles

42.8.1 The rule

If objects can form graphs, ensure at least one direction is non-owning (use `std::weak_ptr`).

42.8.2 Example

```
#include <memory>

struct Node {
    std::shared_ptr<Node> next; // owning
    std::weak_ptr<Node>  prev;  // non-owning to break cycles
};
```

42.9 Thread and lock “leaks”

42.9.1 Lock leaks

A lock that is not released can deadlock the program. Use scoped locks:

```
#include <mutex>

std::mutex m;

void safe_update() {
    std::lock_guard<std::mutex> lock(m);
    // ...
} // unlocks automatically
```

42.9.2 Thread management leaks

Threads are resources: a `std::thread` must be joined or detached before destruction, otherwise the program terminates. Prefer RAII joining patterns:

- use `std::jthread` (C++20) for automatic joining and cooperative stopping,
- or wrap `std::thread` in a joining guard if `jthread` is not available.

```
#include <thread>

void run() {
    std::jthread t([]{ /* work */ }); // joins automatically at scope exit
}
```

42.10 Detecting leaks: build configuration and tooling

Leak avoidance is primarily a design task, but detection is essential:

- Enable compiler warnings and use static analysis where available.
- Use runtime sanitizers and leak detectors in test builds when supported.
- Prefer guideline-friendly code so analysis tools can prove ownership.

42.10.1 Design makes detection easier

When ownership is explicit (`unique_ptr`, containers, RAII handles), many leaks become impossible to express, and both reviewers and tools can identify remaining risks with high confidence.

42.11 A practical “no leaks” checklist

1. Every resource is acquired into an RAII owner immediately.
2. Raw pointers/references are treated as non-owning observers.

3. Ownership transfer uses move-only owners (`unique_ptr`) or return-by-value.
4. Sequences use `std::vector`/`std::array` for ownership and `std::span` for views.
5. Avoid explicit `new`/`delete`; use `make_unique`/containers.
6. Avoid `shared_ptr` unless shared ownership is required; break cycles with `weak_ptr`.
7. Locks are scoped; threads are joined automatically (`jthread`) or guarded.
8. Destructors do not throw; failures are handled by explicit operations or policies.

42.12 Summary

Leaks occur when a program loses the ability to release a resource. The Core Guidelines prevent leaks by requiring that resources are owned by RAII objects and that ownership is explicit in types and interfaces. In modern C++, well-designed code rarely contains explicit `delete`, and most resource cleanup is automatic, exception-safe, and locally understandable.

Chapter 43

Non-Memory Resources

43.1 Introduction

In C++, resources extend beyond dynamic memory. Non-memory resources are entities that must be acquired and released in a balanced way, such as:

- file descriptors and file handles,
- network sockets,
- mutex locks and synchronization primitives,
- database connections and transactions,
- hardware or system handles,
- operating system objects.

Unlike memory, these resources are often limited by the operating system or environment. Leaking them can cause immediate program failure or system-wide degradation. The ISO C++

Core Guidelines treat all resources uniformly: they must be managed safely with clear ownership and automatic cleanup.

43.2 Core Guidelines principles for non-memory resources

The Core Guidelines do not differentiate memory from other resources. The same fundamental rules apply:

- **R.1:** Manage resources automatically using RAII.
- **R.3:** A raw pointer (`T*`) is non-owning.
- **R.4:** A raw reference (`T&`) is non-owning.
- **R.12:** Immediately give the result of an explicit allocation or acquisition to a manager object.
- Use specialized RAII types or small wrappers to encapsulate non-memory resources safely.

The guiding principle is:

> All resources must be owned by objects whose destructors unconditionally release the resource.

43.3 Resource Acquisition Is Initialization (RAII) for non-memory resources

43.3.1 RAII overview

RAII ties a resource's lifetime to the lifetime of an object. The constructor acquires the resource; the destructor releases it. This ensures that resources are released on all control-flow paths, including those involving exceptions.

43.3.2 RAII guarantees

- Deterministic cleanup: resources are released as soon as the owning object goes out of scope.
- Exception safety: cleanup occurs automatically during stack unwinding.
- Ownership clarity: object interfaces express ownership transfer explicitly through types and signatures.

43.4 Common non-memory resources in C++ systems

43.4.1 File handles

File handles represent open files at the OS level. They must be closed explicitly to release operating system resources.

43.4.2 Network sockets

Sockets represent network endpoints. Closing a socket releases bandwidth slots, port assignments, and internal buffers.

43.4.3 Mutexes and locks

Mutexes and other synchronization primitives control concurrent access. Lock leaks commonly result in deadlock or resource contention.

43.4.4 Threads and thread handles

A started thread is a resource: threads must be either joined or detached to allow the runtime to reclaim associated resources safely.

43.4.5 Database connections and transactions

Database resources are external and often limited; proper closure and transaction completion are essential to maintain consistency and availability.

43.5 Designing RAII wrappers for non-memory resources

When the standard library does not provide a suitable RAII wrapper, you must create one that:

- acquires the resource in its constructor or a factory function,
- releases the resource in its destructor,
- is non-copyable (to avoid double release),
- is movable (to allow transfer of ownership),
- exposes safe access to the underlying handle when needed.

43.5.1 Essential wrapper properties

- **Non-copyable:** prevent unintended resource sharing and double release.
- **Movable:** allow ownership transfer without duplicating the resource.
- **Non-throwing destructor:** destructors must not emit exceptions; handle errors inside or via specific operations.
- **Safe accessors:** provide `get()` or conversion operators only when necessary, and make them explicit.

43.6 RAII example: file handle

43.6.1 POSIX file descriptor wrapper

```
#include <unistd.h>
#include <fcntl.h>
#include <stdexcept>

struct FDInvalid {
    static int value() noexcept { return -1; }
};

struct FDClose {
    void operator()(int fd) const noexcept {
        if (fd != FDInvalid::value()) ::close(fd);
    }
};

class UniqueFD {
public:
```

```
    UniqueFD() noexcept : fd_(FDInvalid::value()) {}

    explicit UniqueFD(int fd) noexcept : fd_(fd) {}

    ~UniqueFD() noexcept {
        FDClose{}(fd_);
    }

    UniqueFD(const UniqueFD&) = delete;
    UniqueFD& operator=(const UniqueFD&) = delete;

    UniqueFD(UniqueFD&& other) noexcept : fd_(other.release()) {}

    UniqueFD& operator=(UniqueFD&& other) noexcept {
        if (this != &other) {
            reset(other.release());
        }
        return *this;
    }

    int get() const noexcept { return fd_; }
    int release() noexcept {
        int result = fd_;
        fd_ = FDInvalid::value();
        return result;
    }
    void reset(int new_fd = FDInvalid::value()) noexcept {
        FDClose{}(fd_);
        fd_ = new_fd;
    }

private:
    int fd_;
```

```
};

UniqueFD open_readonly(const char* path) {
    int fd = ::open(path, O_RDONLY);
    if (fd < 0) throw std::runtime_error("open failed");
    return UniqueFD(fd);
}
```

43.6.2 Usage pattern with RAII

```
void process_file(const char* path) {
    UniqueFD file = open_readonly(path);
    // work with file.get()
} // file closed automatically here
```

43.7 RAII example: network socket

43.7.1 Socket wrapper

Below is a conceptual RAII wrapper for a socket handle:

```
#include <sys/socket.h>
#include <unistd.h>
#include <stdexcept>

struct SocketOps {
    static int invalid() noexcept { return -1; }
    static void close(int s) noexcept {
        if (s != invalid()) ::close(s);
    }
};
```

```
class UniqueSocket {
public:
    UniqueSocket() noexcept : sock_(SocketOps::invalid()) {}

    explicit UniqueSocket(int s) noexcept : sock_(s) {}

    ~UniqueSocket() noexcept {
        SocketOps::close(sock_);
    }

    UniqueSocket(const UniqueSocket&) = delete;
    UniqueSocket& operator=(const UniqueSocket&) = delete;

    UniqueSocket(UniqueSocket&& other) noexcept
        : sock_(other.release()) {}

    UniqueSocket& operator=(UniqueSocket&& other) noexcept {
        if (this != &other) {
            reset(other.release());
        }
        return *this;
    }

    int get() const noexcept { return sock_; }
    int release() noexcept {
        int tmp = sock_;
        sock_ = SocketOps::invalid();
        return tmp;
    }
    void reset(int s = SocketOps::invalid()) noexcept {
        SocketOps::close(sock_);
        sock_ = s;
    }
```

```
private:
    int sock_;
};

UniqueSocket create_tcp_client() {
    int s = ::socket(AF_INET, SOCK_STREAM, 0);
    if (s == SocketOps::invalid()) throw std::runtime_error("socket failed");
    return UniqueSocket(s);
}
```

43.7.2 Usage pattern

```
void connect_and_process() {
    UniqueSocket client = create_tcp_client();
    // use client.get()
} // socket closed automatically at scope exit
```

43.8 RAII for synchronization primitives

43.8.1 Standard library support

The standard library provides RAII types for common synchronization resources:

- `std::lock_guard<Mutex>` for simple locking,
- `std::unique_lock<Mutex>` for more flexible locking (defer, unlock/lock, condition variables).

43.8.2 Example: scoped lock

```
#include <mutex>
```

```
std::mutex m;

void safe_increment(int& counter) {
    std::lock_guard<std::mutex> lock(m);
    ++counter;
} // lock released here
```

43.9 Thread resources

43.9.1 Thread as a resource

A running thread is a resource. A thread must be either:

- **joined** (waited on) to ensure completion and cleanup, or
- **detached** to allow it to run independently.

43.9.2 RAII thread management

If a thread object is destroyed while joinable and not detached, the program terminates. Modern C++ provides safer patterns for thread cleanup.

```
#include <thread>

void start_work() {
    std::jthread worker([]{ /* work */ }); // joins automatically on destruction
}
```

43.10 Database connections and transactions

43.10.1 Connection handle RAII

Database connections are typically limited and expensive resources. Wrap them so that the destructor ensures disconnection.

43.10.2 Transaction scope guard

A transaction should be committed or rolled back automatically:

```
class Transaction {
public:
    Transaction(DB& db) : db_(db) { db_.begin(); }
    ~Transaction() noexcept {
        if (!committed_) db_.rollback();
    }
    void commit() { db_.commit(); committed_ = true; }
private:
    DB& db_;
    bool committed_ = false;
};
```

43.11 Guidelines for non-memory resource APIs

43.11.1 Make ownership explicit

Types and function signatures must clearly express:

- whether ownership transfers,
- who is responsible for cleanup,
- whether a resource is borrowed (non-owning) or owned.

43.11.2 Prefer RAII handles in public interfaces

```
void save_file(UniqueFD file); // ownership transferred
void read_file(const UniqueFD& file); // non-owning use
```

43.11.3 Do not expose raw handles if not necessary

Raw handles (file descriptors, socket integers) are non-owning and should appear only when ownership is already expressed externally.

43.12 Avoiding common pitfalls

43.12.1 Pitfall: missing cleanup on error paths

RAII ensures cleanup even if exceptions are thrown or early returns occur.

43.12.2 Pitfall: unintended copy of resources

Make resource wrappers non-copyable if copying would duplicate ownership or lead to double release.

43.12.3 Pitfall: forgetting to join or detach threads

Use scoped thread types or guard objects that automatically join at destruction.

43.12.4 Pitfall: inconsistent lock release

Always use scoped locking types; avoid manual lock/unlock pairs.

43.13 Checking correctness and leaks

43.13.1 Static analysis

Tools that understand RAII patterns can verify correct ownership and resource release paths when code adheres to guideline conventions.

43.13.2 Dynamic leak detection

Even for non-memory resources, runtime tools can detect open handles, unjoined threads, or unclosed connections in test builds.

43.14 Summary

Non-memory resources are ubiquitous and often limited by the environment. The ISO C++ Core Guidelines treat them no differently than memory: ownership must be explicit, and cleanup must be automatic. RAII is the cornerstone of safe resource management. Where standard RAII types do not exist, small custom handle wrappers are the recommended solution. Use standard scoped primitives for synchronization and thread management. Design APIs so that ownership transfer and borrowing are clearly communicated and enforced by types.

Part VII

Pointers and References (P)

Chapter 44

P.1 – P.12: Pointers and References (Guiding Principles)

44.1 Scope and intent

This chapter explains the Core Guidelines principles P.1–P.12 and applies them *specifically* to pointer and reference usage. Pointers and references are powerful, but most hard C++ bugs involve:

- unclear ownership,
- out-of-bounds access,
- dereferencing `nullptr`,
- dangling pointers/references,
- hidden mutation and data races,
- "messy" low-level pointer arithmetic leaking into high-level logic.

The goal is not to ban pointers and references. The goal is to use them so that correctness, performance, and maintainability follow naturally.

44.2 P.1: Express ideas directly in code

44.2.1 Rule

Make pointer/reference intent obvious in the type and the signature, not in comments.

44.2.2 What this means for pointers and references

- Use `T&` when you require a valid object and `nullptr` is not meaningful.
- Use `T*` when "no object" is a valid state (nullable).
- Use `std::span<T>` for non-owning sequences (pointer + length expressed in one value).
- Use `std::string_view` for non-owning text views (when lifetime is guaranteed).
- Use owning types (e.g., `std::unique_ptr<T>`, `std::vector<T>`) to express ownership.

44.2.3 Examples

```
#include <span>

struct Image { /*...*/ };

void draw(const Image& img);        // required object, non-owning
void maybe_draw(const Image* img);  // optional object, non-owning (nullable)

void normalize(std::span<float> v); // non-owning sequence with size
```

44.3 P.2: Write in ISO Standard C++

44.3.1 Rule

Prefer standard language/library facilities for pointer and lifetime safety.

44.3.2 What this means for pointers and references

- Prefer `std::unique_ptr`, `std::shared_ptr`, containers, `std::span`, `std::string_view` over ad-hoc non-standard wrappers.
- If you must use platform APIs (POSIX/Win32 handles, custom allocators), encapsulate them behind a standard C++ interface (RAII handle type).
- Localize unavoidable extensions and keep them out of general APIs.

44.3.3 Example: encapsulate a platform handle

```
class UniqueHandle {
public:
    UniqueHandle() noexcept = default;
    explicit UniqueHandle(int h) noexcept : h_(h) {}
    ~UniqueHandle() noexcept { reset(); }

    UniqueHandle(const UniqueHandle&) = delete;
    UniqueHandle& operator=(const UniqueHandle&) = delete;

    UniqueHandle(UniqueHandle&& other) noexcept : h_(other.release()) {}
    UniqueHandle& operator=(UniqueHandle&& other) noexcept {
        if (this != &other) reset(other.release());
        return *this;
    }
```

```
    int get() const noexcept { return h_; }
    int release() noexcept { int t = h_; h_ = -1; return t; }
    void reset(int nh = -1) noexcept { if (h_ != -1) close(h_); h_ = nh; }

private:
    static void close(int) noexcept; // implemented for the platform
    int h_ = -1;
};
```

44.4 P.3: Express intent

44.4.1 Rule

When a pointer/reference appears, the reader should know:

- is it owning or non-owning?
- can it be null?
- does it refer to one object or a sequence?
- is mutation intended?

44.4.2 Recommended intent patterns

- **Non-owning single object:** `T&` (not null), `T*` (nullable).
- **Non-owning sequence:** `std::span<T>`.
- **Owning single object:** `std::unique_ptr<T>` (single owner), `std::shared_ptr<T>` (shared ownership only when required).
- **View of text:** `std::string_view` (lifetime must outlive the view).

44.4.3 Example

```
#include <memory>
#include <span>
#include <string_view>
#include <vector>

struct Node { /*...*/ };

void set_owner(std::unique_ptr<Node> n); // takes ownership
void observe(const Node& n);             // non-owning, not null
void maybe_observe(const Node* n);       // non-owning, nullable

void process(std::span<const int> xs);   // sequence view
std::string_view name_view(const std::vector<char>& buf); // view, lifetime tied to buf
```

44.5 P.4: Ideally, a program should be statically type safe

44.5.1 Rule

Use types to prevent invalid pointer usage before the program runs.

44.5.2 What this means for pointers and references

- Avoid “typeless” raw memory and casts that defeat the type system.
- Prefer typed owners and typed views (`unique_ptr<T>`, `span<T>`).
- Avoid representing sizes and indexes with the wrong types (e.g., mixing signed/unsigned carelessly).

44.5.3 Example

```
#include <span>
#include <vector>

// BAD: pointer + count can drift apart; count might be wrong
void f(int* p, int n);

// GOOD: size travels with the pointer
void g(std::span<int> s);

void h(std::vector<int>& v) {
    g(v); // safe view over v's contiguous storage
}
```

44.6 P.5: Prefer compile-time checking to run-time checking

44.6.1 Rule

If you can prevent a pointer/reference bug at compile time, do it.

44.6.2 What this means for pointers and references

- Prefer references when null is not a valid state.
- Prefer `const` correctness so the compiler enforces non-mutation.
- Prefer compile-time sizes (`std::array`, templates) when appropriate.
- Prefer RAII owners so the compiler enforces correct destruction patterns.

44.6.3 Examples

```
#include <array>
#include <span>

void fill10(std::span<int, 10> s); // size known at compile time when possible

void use() {
    std::array<int, 10> a{};
    fill10(a); // cannot pass wrong size
}
```

44.7 P.6: What cannot be checked at compile time should be checkable at run time

44.7.1 Rule

For pointer-heavy code, runtime checks must be possible and practical.

44.7.2 What this means for pointers and references

- Use span so bounds can be checked (by code review, debug builds, sanitizers, contracts/policies).
- Validate preconditions for pointer parameters (null checks where nullable; range checks for indexes).
- Prefer APIs that allow checking (e.g., returning `std::optional` instead of "null means failure" when that improves clarity).

44.7.3 Example: explicit checks for nullable pointers

```
void maybe_process(int* p) {
    if (!p) return;   // explicit and local
    *p = 42;
}
```

44.8 P.7: Catch run-time errors early

44.8.1 Rule

Detect pointer misuse at the closest possible point to the cause.

44.8.2 Pointer/reference guidance

- Check for null at the boundary of your API, not deep inside.
- Check indexes before subscripting.
- Validate lifetimes at boundaries: avoid returning pointers/references/views to objects that will not outlive the caller.

44.8.3 Example: fail fast on invalid ranges

```
#include <span>
#include <stdexcept>

int at(std::span<const int> s, std::size_t i) {
    if (i >= s.size()) throw std::out_of_range("index");
    return s[i];
}
```

44.9 P.8: Don’t leak any resources

44.9.1 Rule

Pointer misuse often creates leaks: memory, handles, locks, threads, and more.

44.9.2 Pointer/reference-specific guidance

- Do not treat raw pointers as owners.
- Express ownership using RAII owners (`unique_ptr`, `vector`, RAII handle wrappers).
- Do not manually `delete` in normal application code; make ownership automatic.

44.9.3 Example

```
#include <memory>

// BAD: ownership unclear, easy to leak
int* make_bad() { return new int(7); }

// GOOD: ownership explicit
std::unique_ptr<int> make_good() { return std::make_unique<int>(7); }
```

44.10 P.9: Don’t waste time or space

44.10.1 Rule

Pointer and reference choices affect performance and memory layout.

44.10.2 Balanced guidance

- Prefer value semantics by default; use pointers/references when you need indirection, sharing, optionality, or polymorphism.
- Avoid unnecessary heap allocation and unnecessary shared ownership (reference counting has cost).
- Prefer passing by reference to avoid copies of large objects, but do not hide lifetime risks.

44.10.3 Example: avoid accidental shared ownership

```
#include <memory>

struct Big { /*...*/ };

// GOOD: use by reference for read-only, no refcount traffic
void use_readonly(const Big& b);

// CAUTION: passing shared_ptr everywhere spreads ownership and refcounting cost
void use_shared(std::shared_ptr<Big> b);
```

44.11 P.10: Prefer immutable data to mutable data

44.11.1 Rule

Mutation via pointers/references can be invisible and dangerous, especially with concurrency.

44.11.2 Guidance

- Prefer `const T&` / `const T*` by default.

- Make mutation explicit in naming and signatures.
- Avoid sharing mutable objects across threads without synchronization.

44.11.3 Examples

```
void analyze(const int* p);        // reads
void update(int* p);              // writes (explicit)
void render(const Image& img);    // reads, no mutation
```

44.12 P.11: Encapsulate messy constructs, rather than spreading through the code

44.12.1 Rule

Complex pointer manipulation should be localized inside small, well-tested abstractions.

44.12.2 What counts as “messy”

- pointer arithmetic,
- manual lifetime management (`new`/`delete`),
- reinterpret casts and low-level aliasing tricks,
- manual resource release (handles, sockets, locks),
- ad-hoc ownership conventions.

44.12.3 How to apply

- Put the low-level details in a type (RAII owner, view type, iterator-like wrapper).
- Expose safe operations (bounds-aware, lifetime-safe).
- Keep raw pointers as implementation details where possible.

44.12.4 Example: encapsulate pointer arithmetic behind a view

```
#include <span>
#include <cstddef>

int sum(std::span<const int> s) {
    int r = 0;
    for (int x : s) r += x;
    return r;
}

// Callers can pass vectors, arrays, and pointer+length safely.
```

44.13 P.12: Use supporting tools as appropriate

44.13.1 Rule

Use tools to detect pointer and lifetime defects early and continuously.

44.13.2 Tools that materially help pointer/reference safety

- Compiler warnings at high levels; treat important warnings as errors in CI builds.
- Static analysis (including checkers aligned with the Core Guidelines).

- Runtime sanitizers in test configurations (address, undefined behavior, and when available, leak detection).
- Unit tests and fuzzing for boundary-heavy pointer code.

44.13.3 Practical workflow

- Write pointer-facing APIs in terms of `span`, references, and RAII owners.
- Turn on static analysis to catch obvious lifetime/ownership mistakes.
- Run sanitizer-backed tests to detect real out-of-bounds and use-after-free.

44.14 Consolidated checklist (P.1–P.12 applied)

1. Use `T&` for required objects; `T*` only when nullable is meaningful.
2. Never use raw pointers to express ownership; use RAII owners.
3. Use `std::span` for sequences and `std::string_view` for text views (when lifetime allows).
4. Prefer `const` pointers/references by default.
5. Avoid pointer arithmetic and casts in business logic; encapsulate them.
6. Validate nullness and bounds at API boundaries; fail early.
7. Prefer single ownership; use shared ownership only when necessary (and avoid cycles).
8. Use tools (analysis, sanitizers, tests) continuously.

Chapter 45

Raw Pointers (`T*`)

45.1 What a raw pointer is (and is not)

45.1.1 Definition

A **raw pointer** is a built-in C++ type that stores an address and can refer to:

- an object (`T*`),
- a function (`R(*)(Args...)`),
- a member (`T C::*`) (different category, not covered here).

This chapter focuses on object pointers `T*`.

45.1.2 The most important rule

In the ISO C++ Core Guidelines:

- **A raw pointer (`T*`) is non-owning by default.**

That means: a raw pointer typically *observes* an object managed elsewhere. It does not delete it, close it, or release it.

45.1.3 What raw pointers are good at

Raw pointers are useful when you need:

- **optional access**: `nullptr` means "no object",
- **low-level interop**: calling C APIs, OS APIs, or hardware interfaces,
- **performance-critical observations**: minimal overhead access to objects,
- **implementation details**: inside small, well-tested abstractions.

45.1.4 What raw pointers are bad at

Raw pointers do not encode:

- ownership,
- lifetime guarantees,
- bounds (for sequences),
- nullability constraints (a pointer *can* be null even if your logic assumes it is not).

So raw pointers must be used with explicit conventions and checks.

45.2 Raw pointer meaning in interfaces

45.2.1 Single object vs sequence

A raw pointer in an interface should mean a **single object** (possibly absent). It should not mean a sequence.

45.2.1.1 Preferred: single object

```
struct Image { /*...*/ };

void draw(const Image* img); // optional object: nullptr means "no image"
```

45.2.1.2 Avoid: pointer + length for sequences

If you mean a contiguous sequence, prefer `std::span`:

```
#include <span>

void scale(std::span<float> v, float k);
```

45.2.2 Nullability contracts

45.2.2.1 Use references for non-null requirements

If the parameter must be valid, use a reference:

```
void render(const Image& img); // cannot be null
```

45.2.2.2 Use pointers for optional parameters

If `nullptr` is meaningful, use a pointer and check it:

```
void maybe_render(const Image* img) {
    if (!img) return;
    // safe to use *img
}
```

45.3 Ownership: raw pointers must not silently own

45.3.1 The rule

Do not pass ownership using raw pointers. Ownership must be explicit.

45.3.2 Bad and good patterns

```
struct X {};

// BAD: who deletes? easy leak
X* make_bad() { return new X; }

// GOOD: ownership explicit and automatic
#include <memory>
std::unique_ptr<X> make_good() { return std::make_unique<X>(); }
```

45.3.3 Owning raw pointers at boundaries

Sometimes you must interface with legacy code that returns an owning raw pointer. In guideline-oriented code:

- immediately wrap it in an owning RAII type (`unique_ptr` with a custom deleter if needed),
- keep the owning raw pointer confined to the boundary.

```
#include <memory>

// Legacy API:
extern "C" void* legacy_alloc();
extern "C" void  legacy_free(void*);
```

```
struct legacy_deleter {
    void operator()(void* p) const noexcept { legacy_free(p); }
};

using legacy_ptr = std::unique_ptr<void, legacy_deleter>;

legacy_ptr acquire() {
    void* p = legacy_alloc();
    if (!p) throw std::bad_alloc{};
    return legacy_ptr(p); // ownership established immediately
}
```

45.4 Lifetime safety: avoiding dangling pointers

45.4.1 How dangling happens

A raw pointer dangles when it points to an object that has already been destroyed. Common causes:

- returning the address of a local variable,
- storing a pointer to a temporary,
- storing pointers into containers and then modifying the container (reallocation/erase),
- deleting an object while someone still holds its address.

45.4.2 Classic error: returning address of a local

```
// BAD: returns pointer to a destroyed object
int* bad() {
```

```
    int x = 7;
    return &x;
}
```

45.4.3 Safe alternatives

- Return by value when possible.
- Return an owning type when dynamic lifetime is required.
- If returning a view, ensure the referred storage outlives the view.

```
#include <memory>

int good_value() { return 7; }

std::unique_ptr<int> good_owner() {
    return std::make_unique<int>(7);
}
```

45.5 Bounds safety: raw pointers are not ranges

45.5.1 Why raw pointers are risky for sequences

Pointer arithmetic and indexing on raw pointers provide no automatic bounds information. This makes out-of-bounds access easy.

45.5.2 Prefer `std::span` for contiguous sequences

```
#include <span>
#include <cstddef>
```

```
int sum(std::span<const int> s) {
    int r = 0;
    for (std::size_t i = 0; i < s.size(); ++i) r += s[i];
    return r;
}
```

45.5.3 If you must use pointer+size (interop boundary)

At a boundary (C API, OS API), you may have `T*` plus a count. Immediately convert to a safer view:

```
#include <span>
#include <cstddef>

extern "C" void c_api(const int* p, std::size_t n);

void wrapper(const int* p, std::size_t n) {
    std::span<const int> s{p, n}; // now size travels with the pointer
    // validate s if needed, then use s safely
    c_api(s.data(), s.size());
}
```

45.6 Const correctness with pointers

45.6.1 Two different meanings of const

- `const T* p`: points to const object (cannot modify `*p`)
- `T* const p`: const pointer (cannot change `p` to point elsewhere)
- `const T* const p`: both

```
void f(const int* p);  // cannot modify the pointed-to int
```

45.6.2 Guideline practice

Prefer `const T*` and `const T&` by default. Make mutation explicit in the signature.

45.7 Polymorphism and deletion through base pointers

45.7.1 The rule

If you ever delete an object through a base-class pointer, the base class needs a **virtual destructor**. Otherwise the behavior is undefined.

```
struct Base {
    virtual ~Base() = default;
};

struct Derived : Base {
    ~Derived() = default;
};

Base* p = new Derived{};
delete p; // OK because Base has a virtual destructor
```

45.7.2 Guideline reminder

In modern C++ code, explicit `delete` should be rare. Prefer returning `std::unique_ptr<Base>` from factories.

45.8 Raw pointers and modern C++ safety tooling

Raw pointers are not inherently wrong, but they reduce what tools can prove. To maximize safety:

- Keep raw pointers **non-owning**.
- Prefer **references** when non-null is required.
- Prefer **span** for sequences.
- Confine pointer arithmetic and casts inside small abstractions.
- Use static analysis and runtime sanitizers in test builds for bounds and lifetime defects.

45.9 Recommended patterns (quick reference)

45.9.1 Use raw pointers for

- nullable, non-owning access (`nullptr` allowed),
- observing objects managed elsewhere,
- interop boundaries (C/OS APIs) with immediate conversion to safe abstractions.

45.9.2 Avoid raw pointers for

- expressing ownership (use `unique_ptr`, containers, RAII handles),
- representing arrays (use `span` or containers),
- long-lived stored references without clear lifetime guarantees.

45.10 Checklist: safe raw pointer usage

1. Treat `T*` as **non-owning** by default.
2. Decide and document nullability: check `nullptr` at API boundaries.

3. Use `T&` instead of `T*` when null is not meaningful.
4. Do not transfer ownership by raw pointer; use `unique_ptr` for ownership transfer.
5. Do not use raw pointers for sequences; use `std::span` or containers.
6. Avoid returning pointers/references/views to objects that will not outlive the caller.
7. Avoid pointer arithmetic in application logic; encapsulate it.

45.11 Summary

Raw pointers are a fundamental C++ tool for addressing and optional access, but they encode neither ownership nor lifetime nor bounds. The Core Guidelines therefore treat `T*` as **non-owning** and recommend using raw pointers primarily for nullable observation and interop boundaries. Ownership should be expressed with RAII types and safe sequence access with `std::span`. When used with explicit nullability rules and clear lifetime design, raw pointers remain safe, efficient, and appropriate.

45.12 Introduction

References in C++ are language constructs that provide indirect access to an object without pointer syntax. They are central to modern, expressive, and safe C++ code. This chapter explains how references work, when to use them, and how the ISO C++ Core Guidelines recommend their usage.

45.13 What a reference is

45.13.1 Definition

A **reference** is an alias for another object. Once bound, a reference refers to the same object for its entire lifetime. There are two primary reference types in modern C++:

- **Lvalue reference:** `T&`, referring to an existing lvalue.
- **Rvalue reference:** `T&&`, introduced in C++11 to support move semantics and perfect forwarding.

References are not objects themselves; they are bound to objects.

45.13.2 Key properties

- A reference must be initialized when declared.
- An lvalue reference cannot be reseated to refer to a different object after binding.
- A reference cannot be null (no "null reference" in the language).
- References do not require explicit dereferencing syntax (`*p`); you use them like the object itself.

45.14 Why references are safer than pointers

- Impossible to be null — eliminates a class of errors.
- Clear intent — a reference always refers to a valid object (at least at binding).
- Cleaner syntax — no explicit dereference or `nullptr` checks when nullability is not needed.

45.15 Lvalue references (T&)

45.15.1 Definition and typical use

An lvalue reference binds to an existing object (an lvalue). It expresses "I refer to this object."

```
int x = 10;
int& r = x; // r refers to x
r = 20;     // modifies x
```

45.15.2 Function parameters and return values

Use T& when a function must:

- operate on an existing object,
- take a non-null reference,
- avoid copies of large objects.

```
void scale(int& value, int factor) {
    value *= factor;
}
```

45.15.3 Const lvalue references (const T&)

A const T& binds to const or non-const objects without allowing mutation. This is the preferred way to pass large or expensive-to-copy objects that are not modified:

```
void print(const std::string& s) {
    // s is read-only
}
```

Const references can also bind to temporaries:

```
const std::string& greeting = std::string("hello");
```

45.16 Rvalue references (T&&)

45.16.1 Definition and purpose

An rvalue reference binds to an rvalue (temporary). It enables:

- move semantics — transferring resources from temporaries,
- perfect forwarding — forwarding arguments while preserving value categories.

45.16.2 Move semantics example

```
#include <utility>
#include <vector>

std::vector<int> make_vector() {
    std::vector<int> v{1,2,3};
    return v; // allows move
}

void f() {
    std::vector<int> a = make_vector(); // moved, not copied
}
```

45.16.3 Perfect forwarding with templates

Rvalue references enable templates to forward arguments correctly:

```
template<class F, class... Args>
auto invoke(F&& f, Args&&... args)
    -> decltype(std::forward<F>(f)(std::forward<Args>(args)...))
{
    return std::forward<F>(f)(std::forward<Args>(args)...);
}
```

45.17 When to use references (Core Guidelines perspective)

The ISO C++ Core Guidelines recommend references for:

- required function parameters — use `T&` or `const T&`,
- avoiding nullability — references cannot be null,
- avoiding unnecessary copies — especially for large or non-trivially-copyable types.

For optional parameters, pointers (`T*`) or `std::optional<std::reference_wrapper<T>>` are better explicit choices.

45.18 References vs pointers

45.18.1 Nullability

- Reference: cannot be null — intent of "must have an object".
- Pointer: can be null — intent of "optional object".

45.18.2 Ownership and lifetime

Neither raw pointers nor references express ownership. Instead:

- Own resources with RAII types (smart pointers, containers).
- Use references and pointers only to observe.

45.18.3 Syntax differences

```
int x = 5;
int* p = &x;   // pointer
int& r = x;    // reference
*p = 10;
r = 10;        // no dereference operator
```

45.19 Common reference pitfalls

45.19.1 Dangling references

A reference that refers to an object that has been destroyed leads to undefined behavior.

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

45.19.2 Binding to temporary objects

Non-const lvalue references cannot bind to temporaries. Only const lvalue references and rvalue references can.

```
int&& r1 = 5;          // OK, rvalue reference
const int& r2 = 5;     // OK, const lvalue reference binds to temporary
int& r3 = 5;           // Error: cannot bind
```

45.20 Reference collapsing rules

When combining references in templates, reference collapsing applies:

- `T& &` becomes `T&`
- `T& &&` becomes `T&`
- `T&& &` becomes `T&`
- `T&& &&` becomes `T&&`

This is crucial for correct forwarding and template parameter deduction.

45.21 Function overloading with references

References enable expressive overloads for value categories:

```
void process(const std::string&);  // lvalue
void process(std::string&&);       // rvalue (can move)
```

The second overload can move resources efficiently when called with temporaries.

45.22 References and object lifetime

45.22.1 Safe lifetime usage

A reference must not outlive the object it refers to. Passing references to local objects that go out of scope is undefined behavior.

45.22.2 References to data in containers

If a container reallocates, references to its elements become invalid. Use stable handles only when guaranteed by the container's policy.

45.23 References in modern API design

45.23.1 Required parameters

For functions that require a valid object:

```
void draw(const Image& img); // required non-null image
```

45.23.2 Optional parameters

When null or absence is meaningful, prefer explicit optional types or pointers:

```
#include <optional>
void process(std::optional<std::reference_wrapper<Image>> img);
```

This makes nullability or optional presence explicit in the type system.

45.24 Summary

References are safe, expressive, and central to modern C++. Use:

- `T&` for required access to an object,
- `const T&` when you need read-only access without copying,
- `T&&` for move semantics and forwarding.

References avoid nullability issues, have cleaner syntax than pointers, and help express intent clearly. Still, be mindful of lifetime; references must not outlive the objects they refer to.

Chapter 46

Pointers as Observers (Non-owning Use)

46.1 Core idea

In guideline-oriented modern C++, **raw pointers are observers by default**. That is:

> `T*` *and* `T&` *typically do not own; they refer to an object owned elsewhere.*

This convention is central to both safe design and effective tooling. If raw pointers sometimes owned and sometimes did not, neither humans nor static analysis could reliably find leaks, double deletes, or lifetime bugs.

46.2 Observer vs owner (precise definitions)

46.2.1 Observer

An **observer** is a handle that:

- refers to an existing resource,

- does not control the resource lifetime,
- must never release the resource.

Observer types include:

- `T&` (non-null observer of a single object),
- `T*` (nullable observer of a single object),
- `std::span<T>` (observer of a contiguous sequence with a size),
- `std::string_view` (observer of text),
- iterators and references into containers (while valid).

46.2.2 Owner

An **owner** is responsible for releasing the resource exactly once. Owners are RAII types:

- `std::unique_ptr<T>` (single owner),
- `std::shared_ptr<T>` (shared owners; use only when needed),
- owning containers such as `std::vector<T>` and `std::string`,
- custom RAII handle wrappers for OS resources (file handles, sockets, etc.).

46.3 Core Guidelines rules behind “pointers as observers”

The following resource-management conventions are foundational:

- **R.3:** A raw pointer (`T*`) is non-owning.
- **R.4:** A raw reference (`T&`) is non-owning.

- **R.1:** Manage resources automatically using RAII.

Taken together, they enforce a clean separation:

- **ownership** is expressed by owner types and RAII,
- **observation** is expressed by raw pointers/references and views.

46.4 Why treating pointers as observers matters

46.4.1 Avoids ownership ambiguity

If a pointer might own, the reader cannot answer:

- "Who deletes this?"
- "Is it safe to store this pointer?"
- "Will this call transfer ownership?"

46.4.2 Improves exception safety

RAII owners clean up automatically on all paths. Observers never clean up. When the two roles are distinct, exception safety becomes structural.

46.4.3 Enables robust tooling

Static analyzers and guideline checkers can detect:

- leaks (owner not released),
- double delete (two owners),

- dangling observers (lifetime issues),
- null dereference risks.

46.5 Observer pointers in interfaces (recommended patterns)

46.5.1 Required vs optional single objects

46.5.1.1 Required object: use a reference

```
struct Image { /*...*/ };

void render(const Image& img); // required, non-owning, not null
```

46.5.1.2 Optional object: use a pointer

```
void maybe_render(const Image* img) {
    if (!img) return;  // nullptr means "no image"
    render(*img);
}
```

46.5.2 Sequences: use `std::span` as an observer

A raw pointer does not carry bounds. For a contiguous sequence, prefer `std::span`:

```
#include <span>
#include <cstddef>

int sum(std::span<const int> s) {
    int r = 0;
    for (std::size_t i = 0; i < s.size(); ++i) r += s[i];
    return r;
}
```

46.5.3 Text: use `std::string_view` as an observer

`std::string_view` is a non-owning view; it does not extend lifetime.

```
#include <string_view>

bool starts_with_http(std::string_view s) {
    return s.rfind("http", 0) == 0;
}
```

46.6 Owning objects should not be exposed as raw pointers

46.6.1 Do not return owning raw pointers

```
// BAD: ownership unclear
struct X {};
X* make_bad() { return new X; }
```

46.6.2 Return owners or values

```
#include <memory>

// GOOD: ownership explicit
struct X {};
std::unique_ptr<X> make_good() { return std::make_unique<X>(); }

// ALSO GOOD: return by value when feasible
X make_value() { return X{}; }
```

46.6.3 If you must accept a legacy owning raw pointer

At boundaries where you cannot change signatures (ABI/legacy/C API), treat the owning raw pointer as a boundary artifact:

- convert it immediately into an RAII owner,
- keep raw ownership out of the rest of the code.

```
#include <memory>

extern "C" void* legacy_create();
extern "C" void  legacy_destroy(void*);

struct legacy_deleter {
    void operator()(void* p) const noexcept { legacy_destroy(p); }
};

using legacy_owner = std::unique_ptr<void, legacy_deleter>;

legacy_owner acquire() {
    void* p = legacy_create();
    if (!p) throw std::bad_alloc{};
    return legacy_owner{p}; // ownership becomes explicit and automatic
}
```

46.7 Lifetime rules for observer pointers

46.7.1 The fundamental constraint

An observer pointer is safe only while the observed object is alive. Observers do not extend lifetime.

46.7.2 Common ways observers dangle

- storing a pointer to a local object and using it after the function returns,
- storing pointers/references into a container and then changing the container in a way that invalidates them (reallocation, erase),
- deleting an object while other code still holds pointers to it,
- returning `std::span` or `string_view` referring to a temporary or local buffer.

46.7.3 Classic example: dangling pointer

```
// BAD: returns pointer to a dead object
int* bad() {
    int x = 7;
    return &x;
}
```

46.7.4 Safe alternatives

```
#include <memory>

// GOOD: return by value
int good_value() { return 7; }

// GOOD: return an owner when dynamic lifetime is required
std::unique_ptr<int> good_owner() {
    return std::make_unique<int>(7);
}
```

46.8 Nullability and observer pointers

46.8.1 Use pointers when null is meaningful

A pointer expresses optionality naturally:

```
void set_label(const char* text); // nullptr means "clear label"
```

46.8.2 Use references when null must not occur

References are the strongest built-in non-null signal in C++:

```
void set_config(const Config& cfg); // required
```

46.8.3 Non-null pointer contracts (project vocabulary)

Some codebases use additional vocabulary types to express non-null pointers explicitly (for humans and tools). When available, they can improve clarity at boundaries where a reference is not suitable.

```
#include <gsl/gsl>

void use_ptr(gsl::not_null<int*> p); // cannot be null by contract
```

46.9 Observer pointers and const correctness

46.9.1 Default to read-only observation

Prefer `const T*` or `const T&` for observers unless mutation is intended.

```
void analyze(const int* p);   // observes
void update(int* p);          // mutates (explicit)
```

46.9.2 Make mutation intentional

If a function accepts `T*` and mutates `*p`, treat that as part of the function's contract and document it clearly.

46.10 Observer pointers in class design

46.10.1 Storing observer pointers requires a lifetime guarantee

Storing `T*` or `T&` as a data member is valid only if the class design ensures the pointed-to object outlives the observer object.

```
class View {
public:
    explicit View(const std::string& s) : s_(&s) {}
    std::size_t size() const { return s_->size(); }

private:
    const std::string* s_; // observer; caller must ensure lifetime
};
```

46.10.2 Prefer owning members when lifetime is complex

If the lifetime relationship is not simple and stable, prefer ownership (copy/move into the object) or use a shared ownership design intentionally.

46.11 A compact checklist: pointer-as-observer discipline

1. Treat `T*` and `T&` as **non-owning observers**.
2. Use `T&` for required objects; use `T*` for optional objects.

3. Do not use raw pointers to represent sequences; use `std::span`.
4. Do not return or store observers unless the lifetime is guaranteed.
5. Do not transfer ownership via raw pointers; use `unique_ptr` for ownership transfer.
6. Prefer `const` observers by default; make mutation explicit.
7. Encapsulate pointer arithmetic and low-level details inside small abstractions.

46.12 Summary

Treating pointers as observers is a cornerstone of the ISO C++ Core Guidelines approach to safety. It separates two concerns cleanly:

- **Owners** manage lifetimes and release resources automatically (RAII).
- **Observers** refer to objects without controlling lifetime.

With this separation, interfaces become clearer, exception safety becomes structural, and tooling can reliably detect leaks and lifetime defects.

Chapter 47

Avoiding Dangling Pointers

47.1 Introduction

A **dangling pointer** is a pointer that refers to memory or a resource that has already been destroyed or released. Dereferencing a dangling pointer leads to undefined behavior, which can cause crashes, data corruption, or security vulnerabilities. Avoiding dangling pointers is essential for robust C++ code.
This chapter explains the causes of dangling pointers, how to prevent them according to the ISO C++ Core Guidelines, and best practices for safe pointer use.

47.2 What is a dangling pointer

47.2.1 Precise definition

A pointer becomes **dangling** when:

- the object it pointed to is destroyed,

- or the storage it pointed to has been deallocated or moved.

Dereferencing such a pointer is undefined. The same concept applies to references, iterators, and views that refer to objects that no longer exist.

47.2.2 Common scenarios that cause dangling

- Returning a pointer/reference to a local variable.
- Storing a pointer to an element that is removed or invalidated.
- Deleting or freeing memory while observers still hold pointers to it.
- Moving or reallocating containers, invalidating references or pointers to their elements.
- Using `std::string_view` or `std::span` to refer to temporary data.

47.3 Why dangling pointers are dangerous

Dereferencing a dangling pointer can result in:

- undefined behavior,
- program crashes,
- corrupted data,
- security vulnerabilities (use-after-free).

Because the compiler cannot reliably detect dangling pointer dereferences at compile time, design discipline and safe patterns are required.

47.4 Guidelines for preventing dangling pointers

The ISO C++ Core Guidelines convey a set of principles that naturally eliminate dangling pointer risks:

- **Ownership must be explicit and managed by RAII** (Resource Acquisition Is Initialization).
- **Observers must not outlive the objects they refer to**.
- **Reference and pointer lifetimes must be documented and enforced**.

The relevant guidelines include:

- **R.1:** Manage resources automatically using RAII.
- **R.3–R.4:** Raw pointers and references are non-owning (observers only).
- **P.1–P.12:** Philosophy guidelines that enforce clarity and lifetime safety.

47.5 Lifetime rules for pointers and references

47.5.1 Object lifetime vs pointer lifetime

A pointer or reference must never outlive the object it observes. If the object is destroyed, all pointers/references to it immediately become invalid.

47.5.2 Local variables

Never return pointers or references to local (stack) variables. Once the function returns, the stack frame is gone, and the object no longer exists.

```
// BAD: returns pointer to a dead variable
int* bad_ptr() {
    int x = 42;
    return &x;
}

// BAD: returns reference to dead variable
int& bad_ref() {
    int x = 42;
    return x;
}
```

47.5.3 Safe alternatives

```
#include <memory>

// return by value
int good_value() {
    return 42;
}

// return owning pointer
std::unique_ptr<int> good_owner() {
    return std::make_unique<int>(42);
}
```

47.6 Containers and iterator invalidation

Containers can invalidate pointers and references to their elements when they modify their internal structure.

47.6.1 Dynamic array containers (vector, string)

Operations that can invalidate references and pointers:

- Reallocation triggered by growth beyond capacity.
- `clear()`, `erase()`, `insert()` at a position.

```
#include <vector>

void example() {
    std::vector<int> v = {1,2,3};
    int* p = &v[0]; // observer pointer
    v.push_back(4); // may reallocate and invalidate p
    // p may now be dangling
}
```

47.6.2 Associative containers (set, map)

In many associative containers, pointers and references to elements remain valid after insertions, but not always after erasure.

47.6.3 Guideline: prefer safe views or indices

When referring to container elements across operations that may invalidate pointers:

- use indices or keys instead of raw pointers where appropriate,
- re-obtain pointers or references after potentially invalidating operations.

47.7 Temporary objects and views

47.7.1 Pointers/references to temporaries

Binding a pointer or reference to a temporary that dies at the end of the expression creates a dangling observer.

```
#include <string>

const std::string& bad_ref_to_temp() {
    return std::string("temp"); // dangling reference
}
```

47.7.2 Views to temporaries

Views such as `std::string_view` and `std::span` do not extend the lifetime of the objects they observe.

```
#include <string_view>
#include <string>

std::string_view bad_view() {
    std::string s = "text";
    return std::string_view(s); // dangling view
}
```

47.7.3 Safe alternatives

- Bind views only to objects whose lifetimes are known to exceed the view's use.
- Use owning return types if the data must live beyond a scope.

47.8 Observer pointers and references in class design

47.8.1 Storing observers as members

Storing a pointer or reference as a data member is safe only if the class ensures that the observed object outlives the observer.

```
class View {
public:
    explicit View(const std::string& s) : s_(&s) {}
    std::size_t size() const { return s_->size(); }
private:
    const std::string* s_; // observer
};
// Caller must ensure the std::string outlives the View
```

47.8.2 Prefer owning members when lifetime is unclear

If a class cannot guarantee that the referred object outlives it, store an owning type (`std::string`, `std::unique_ptr<T>`) instead.

47.9 Smart pointers and dangling avoidance

47.9.1 Non-owning raw pointers vs owning smart pointers

Smart pointers express ownership clearly and automatically release resources. Raw pointers remain observers and should not be used for ownership.

47.9.2 Using `std::weak_ptr` to avoid dangling

When using `std::shared_ptr`, `std::weak_ptr` allows observing an object without preventing its destruction. Before use, lock it to get a `std::shared_ptr`, and check if the object still exists.

```cpp
#include <memory>
#include <iostream>

void example_weak() {
    std::shared_ptr<int> sp = std::make_shared<int>(10);
    std::weak_ptr<int> wp = sp; // observer that does not own

    if (auto spt = wp.lock()) { // promote to owning pointer
        std::cout << *spt << '\n';
    }
    sp.reset(); // object destroyed

    if (wp.expired()) {
        std::cout << "object no longer exists\n";
    }
}
```

47.10 Dangling and concurrency

47.10.1 Data races and lifetime

In concurrent code, dangling pointer risks also arise when one thread destroys an object while another thread still uses a pointer/reference to that object. Proper synchronization and ownership discipline are essential.

47.10.2 Guideline: synchronize or extend lifetime

Ensure shared objects are not destroyed until all threads have finished accessing them:

- use synchronization primitives (mutexes, condition variables),
- use shared ownership (`std::shared_ptr`) with caution and clear lifetimes.

47.11 Compiler and tooling assistance

47.11.1 Compile-time checks

Strong type safety and avoiding raw ownership clues help compilers catch lifetime mismatches early.

47.11.2 Static analysis and sanitizers

Use static analysis tools and runtime sanitizers (when available) in test builds to detect use-after-free and invalid access patterns.

47.12 Checklist for avoiding dangling pointers

1. Do not return pointers or references to local variables or temporaries.
2. Do not store raw observers unless the object's lifetime exceeds the observer's.
3. Prefer owning types for members when lifetimes are complex.
4. Re-obtain pointers after operations that may invalidate them (container reallocation, erase).
5. When using views (`std::span`, `std::string_view`), ensure the underlying storage outlives the view.
6. For shared resources in concurrency, synchronize destruction and access.
7. In shared ownership scenarios, use `std::weak_ptr` to observe safely without extending lifetime inadvertently.

47.13 Summary

Dangling pointers occur when observers outlive the objects they refer to. The ISO C++ Core Guidelines enforce patterns that separate ownership from observation, making lifetime relationships explicit and safe. By using RAII for ownership, ensuring observers never outlive their subjects, and choosing safe abstractions (`std::span`, smart pointers) over raw lifetime assumptions, dangling pointer bugs become far less common and easier to avoid.

Chapter 48

Nullability

48.1 Introduction

Nullability refers to whether a reference or pointer can represent the absence of an object. In modern C++, clear and explicit nullability is essential for safe code. The ISO C++ Core Guidelines emphasize expressing nullability in the type system or via supporting abstractions so that readers and tools understand whether `nullptr` is permitted.
This chapter explains:

- what nullability means for pointers and references,
- how to express nullability clearly,
- how to avoid common null-related bugs,
- recommended patterns for optional and required parameters.

48.2 Nullability basics

48.2.1 What nullability means

Nullability is the ability of a reference or pointer to represent "no object." In C++:

- **Raw pointers** (`T*`) can be null (`nullptr`).
- **References** (`T&`) cannot be null by language design.

48.2.2 Why nullability matters

A nullable pointer:

- must be checked before dereferencing,
- communicates optional presence,
- may require explicit documentation or code checks.

A reference (non-null by contract) avoids explicit null checks in the body.

48.3 Nullability in function parameters

48.3.1 Required object (non-null)

Use a reference when the function requires a valid object:

```
struct Config { /*...*/ };

void apply(const Config& cfg); // cfg must be valid; no need for null checks
```

48.3.2 Optional object (nullable)

Use a pointer when null is meaningful:

```
void maybe_process(Data* data) {
    if (!data) return; // explicit null check
    // safe to use *data
}
```

48.3.3 Why not use references for optional

References cannot be null by language definition. Allowing an optional object via reference would require an additional flag or wrapper type, which is less clear than a pointer or an optional type.

48.4 Nullability and ownership

48.4.1 Observers vs owners

Nullability applies primarily to observers:

- observers may or may not refer to an object,
- ownership is handled by RAII types (`unique_ptr`, `shared_ptr`), which may also be null.

48.4.2 Owning smart pointers and nullability

Owning smart pointers are nullable by design:

```
#include <memory>

std::unique_ptr<int> up; // initially null
if (!up) { /* no owned object */ }
```

Shared ownership is also nullable:

```
std::shared_ptr<Data> sp = nullptr;
if (!sp) { /* no shared object */ }
```

48.5 Representing nullability explicitly

48.5.1 Raw pointers

Raw pointers express nullability directly:

- `T*` can be `nullptr` or point to an object.
- When null is invalid, avoid raw pointers to represent required objects.

48.5.2 Optional reference wrappers

When you want a reference-like view that may be absent, use `std::optional<std::reference_wrapper<T>>`:

```
#include <optional>
#include <functional>

void maybe_show(std::optional<std::reference_wrapper<const Data>> data_opt) {
    if (data_opt) {
        show(data_opt->get());
    }
}
```

This pattern clearly conveys optional presence with reference semantics.

48.6 Nullability and return values

48.6.1 Returning optional objects

When a function may fail to produce a value, prefer an explicit optional type:

```
#include <optional>

std::optional<Result> find_value(int key) {
    if (exists(key)) return Result{/*...*/};
    return std::nullopt;
}
```

48.6.2 Avoid returning null raw pointers if alternatives exist

Prefer optional or owning smart pointers for return types when nullability is meaningful:

```
// Avoid:
Widget* find_widget(int id) { /* may return nullptr */ }

// Prefer:
std::shared_ptr<Widget> find_widget(int id) {
    if (exists(id)) return /* owning pointer */;
    return nullptr;
}
```

The owning pointer conveys lifetime semantics and nullability in one type.

48.7 Nullability in class members

48.7.1 Nullable observers

If a class stores a pointer that may be absent, document nullability clearly:

```
class Handler {
    Processor* proc_; // non-owning, may be nullptr
public:
    explicit Handler(Processor* p = nullptr) : proc_(p) {}
    void run() {
        if (proc_) proc_->execute();
    }
};
```

48.7.2 Prefer safe alternatives when possible

If the member must always refer to a valid object, prefer a reference or an owning type:

```
class Manager {
    Resource& resource_; // non-null by language rule
public:
    explicit Manager(Resource& r) : resource_(r) {}
};
```

48.8 Nullability and initialization

48.8.1 Pointers must be initialized explicitly

Uninitialized pointers can contain indeterminate values. Always initialize them:

```
int* p = nullptr; // safe initial value
```

48.8.2 References must be bound at initialization

A reference must be initialized when declared:

```
int x = 5;
int& r = x; // valid
```

There is no concept of an “uninitialized reference.”

48.9 Nullability and `nullptr`

48.9.1 Use `nullptr` instead of 0

Modern C++ introduces `nullptr` to represent the null pointer unambiguously. Avoid using integer `0` or `NULL` for null:

```
int* p = nullptr; // clear null pointer
```

48.10 Nullability and comparisons

48.10.1 Check for null safely

Always check for null before dereferencing:

```
if (ptr != nullptr) {
    // safe to dereference
    use(*ptr);
}
```

48.10.2 Prefer direct tests

When testing a pointer in a condition, prefer direct tests:

```
if (ptr) {
    // ptr is not null
}
```

This is equivalent to `ptr != nullptr` and clearer.

48.11 Nullability and contracts

48.11.1 Document requirements

When a pointer parameter must not be null, document that requirement in the interface and enforce it in code where appropriate:

```
void process(NotNullPointer<int*> p) {
    // contract: p is not nullptr
    use(*p);
}
```

This example uses a hypothetical “NotNullPointer” abstraction to express nullability constraints at the type level.

48.11.2 Fail early on null violations

When null is not permitted, check and fail early:

```
#include <stdexcept>

void process(int* p) {
    if (!p) throw std::invalid_argument("pointer must not be null");
    use(*p);
}
```

Explicit checks at boundaries improve clarity and safety.

48.12 Special cases: arrays and nullability

48.12.1 Pointers to arrays

A pointer to an array element may be null if it does not point to any element:

```
int* array_ptr = nullptr; // no array
```

48.12.2 Bounds safety

Nullability does not carry bounds information. Prefer `std::span` to express arrays with size for safety and clarity.

```
#include <span>
void process(std::span<int> s) {
    // size and pointer in one view
}
```

48.13 Checklist for nullability

1. Use references for required, non-null parameters.
2. Use pointers for optional parameters; check for null.
3. Initialize raw pointers explicitly to `nullptr`.
4. Prefer `nullptr` over integer `0` or `NULL`.
5. Use smart pointers or `std::optional` to express nullability with ownership.
6. Avoid returning raw nullable pointers when better alternatives (optional or owning pointers) exist.
7. Document nullability expectations in interfaces.

48.14 Summary

Nullability is an essential aspect of pointer and reference usage in C++. The ISO C++ Core Guidelines encourage explicit expression of whether a value may be absent. Using references for non-nullable requirements, pointers for optional access, smart pointers for owning semantics, and optional or reference wrappers for optional references leads to clearer, safer, and easier-to-maintain code.

Part VIII

Expressions and Statements (ES)

Chapter 49

ES.1 – ES.114: Expressions and Statements

49.1 Editorial accuracy and scope

According to the ISO C++ Core Guidelines revision dated **July 8, 2025**:

- The **ES** section (Expressions and Statements) defines rules up to **ES.107**.
- Rule numbers **ES.108 – ES.114 do not exist** in the official guidelines.

This chapter:

- Explains all **existing ES rules** in a structured, readable form.
- Explicitly **lists every ES rule number from ES.1 to ES.114**.
- Clearly marks **non-existent rule numbers** for completeness and future-proof referencing.

49.2 Purpose of the ES rules

The ES rules govern how expressions and statements should be written to ensure:

- correctness and defined behavior,
- clarity and maintainability,
- safety (especially avoiding undefined behavior),
- predictable performance,
- tool friendliness (static analysis and reviews).

Most subtle and dangerous C++ bugs originate in expressions and control flow rather than in high-level design.

49.3 ES.1 – ES.34: Declarations, naming, initialization, macros

ES.1

Prefer the standard library to other libraries and to handcrafted code.

ES.2

Prefer suitable abstractions to direct use of language features.

ES.3

Don’t repeat yourself. Avoid redundant code.

ES.4

Avoid overloading operators for unrelated or surprising meanings.

ES.5

Keep scopes small.

ES.6

Declare names in for-statement initializers and conditions to limit scope.

ES.7

Keep common and local names short; keep uncommon and non-local names longer.

ES.8

Avoid similar-looking names.

ES.9

Avoid ALL_CAPS names (except for macros).

ES.10

Declare one name (only) per declaration.

ES.11

Use `auto` to avoid redundant repetition of type names.

ES.12

Do not reuse names in nested scopes.

ES.13

Do not reuse variable names in a single scope.

ES.14

Avoid magic numbers; use named constants.

ES.15

Prefer `constexpr` for values that can be computed at compile time.

ES.16

Avoid using macros for constants or functions.

ES.17

Avoid macros for program text manipulation.

ES.18

Prefer enum classes over macros or plain enums.

ES.19

Avoid overly complex expressions.

ES.20

Always initialize an object.

ES.21

Don’t introduce a variable before you need to use it.

ES.22

Don’t declare a variable until you have a value to initialize it with.

ES.23

Prefer the {}-initializer syntax.

ES.24

Use `std::unique_ptr<T>` to hold owning pointers.

ES.25

Declare objects `const` or `constexpr` unless modification is intended.

ES.26

Don’t use a variable for two unrelated purposes.

ES.27

Use `std::array` or `stack_array` for stack-based arrays.

ES.28

Use lambdas for complex initialization, especially of const objects.

ES.29

Avoid complicated conditional expressions.

ES.30

Don’t use macros for program logic.

ES.31

Don’t use macros for constants or “functions”.

ES.32

Use ALL_CAPS for all macro names.

ES.33

If macros are unavoidable, give them unique names.

ES.34

Don’t define C-style variadic functions.

49.4 ES.35 – ES.69: Expressions, conversions, pointers, lifetimes

ES.35

Avoid implicit type conversions that lose information.

ES.36

Avoid implicit narrowing conversions.

ES.37

Don’t use casts to suppress compiler warnings.

ES.38

Avoid implicit fallthrough in `switch` statements.

ES.39

Prefer named casts over C-style casts.

ES.40

Avoid complicated expressions.

ES.41

If in doubt about operator precedence, parenthesize.

ES.42

Keep use of pointers simple and straightforward.

ES.43

Avoid expressions with undefined order of evaluation.

ES.44

Don’t depend on the order of evaluation of function arguments.

ES.45

Avoid magic constants; use symbolic constants.

ES.46

Avoid narrowing conversions.

ES.47

Use `nullptr` rather than `0` or `NULL`.

ES.48

Avoid casts when possible.

ES.49

If a cast is necessary, use a named cast.

ES.50

Don't cast away `const`.

ES.51

Avoid converting pointers to unrelated types.

ES.52

Avoid pointer arithmetic outside well-defined ranges.

ES.53

Avoid arithmetic on `void*`.

ES.54

Avoid comparing pointers from different objects.

ES.55

Avoid the need for range checking by design.

ES.56

Write `std::move()` only when an explicit move is intended.

ES.57

Avoid returning references to local variables.

ES.58

Avoid returning pointers to local variables.

ES.59

Avoid returning references to temporaries.

ES.60

Avoid `new` and `delete` outside resource-management code.

ES.61

Delete arrays with `delete[]` and non-arrays with `delete`.

ES.62

Don’t compare pointers into different arrays.

ES.63

Don’t slice objects.

ES.64

Use `T{e}` notation for construction.

ES.65

Don’t dereference invalid pointers.

ES.66

Avoid accessing objects through invalid iterators.

ES.67

Avoid dereferencing past-the-end iterators.

ES.68

Avoid pointer arithmetic that escapes object bounds.

ES.69

Avoid unsafe use of `reinterpret_cast`.

49.5 ES.70 – ES.87: Control flow and statements

ES.70

Prefer a `switch` statement to an `if` chain when appropriate.

ES.71

Prefer range-for loops to traditional for-loops.

ES.72

Prefer a for-statement when there is an obvious loop variable.

ES.73

Prefer a while-statement when there is no obvious loop variable.

ES.74

Declare loop variables in the for-statement initializer.

ES.75

Avoid do-statements.

ES.76

Avoid `goto`.

ES.77

Minimize the use of `break` and `continue`.

ES.78

Don't rely on implicit fallthrough in `switch` statements.

ES.79

Use `default` only for common cases.

ES.80

Avoid empty `default` cases that hide missing enumerators.

ES.81

Prefer structured control flow to complex branching.

ES.82

Avoid deeply nested control structures.

ES.83

Avoid complex boolean expressions in conditions.

ES.84

Don’t declare unnamed local variables.

ES.85

Make empty statements visible.

ES.86

Avoid modifying loop control variables inside the loop body.

ES.87

Don’t add redundant comparisons to conditions.

49.6 ES.88 – ES.107: Errors, arithmetic, and correctness

ES.88

Avoid using exceptions for normal control flow.

ES.89

Avoid throwing exceptions from destructors.

ES.90

Avoid using `assert` for runtime error handling.

ES.91

Avoid side effects in assertions.

ES.92

Avoid relying on undefined behavior.

ES.93

Avoid implementation-defined behavior where portability matters.

ES.94

Avoid unspecified behavior where determinism is required.

ES.95

Avoid signed integer overflow.

ES.96

Avoid unsigned wraparound when used as arithmetic.

ES.97

Avoid mixing signed and unsigned arithmetic.

ES.98

Use signed types for general arithmetic.

ES.99

Use unsigned types only for bit manipulation.

ES.100

Don’t mix signed and unsigned arithmetic.

ES.101

Use unsigned types for bit manipulation.

ES.102

Use signed types for arithmetic.

ES.103

Don’t overflow.

ES.104

Don’t underflow.

ES.105

Don’t divide by integer zero.

ES.106

Don’t try to avoid negative values by using unsigned types.

ES.107

Don’t use unsigned types for subscripts; prefer a signed index type.

49.7 ES.108 – ES.114: Non-existent rule numbers

ES.108

Not defined in the ISO C++ Core Guidelines.

ES.109

Not defined in the ISO C++ Core Guidelines.

ES.110

Not defined in the ISO C++ Core Guidelines.

ES.111

Not defined in the ISO C++ Core Guidelines.

ES.112

Not defined in the ISO C++ Core Guidelines.

ES.113

Not defined in the ISO C++ Core Guidelines.

ES.114

Not defined in the ISO C++ Core Guidelines.

49.8 Summary

The ES rules define how to write expressions and statements that are:

- readable and reviewable,
- free of undefined and unspecified behavior,
- safe with respect to lifetimes, arithmetic, and control flow,
- aligned with modern C++ abstractions and tooling.

By following ES.1–ES.107 and explicitly acknowledging the absence of ES.108–ES.114, this chapter provides a complete, precise, and future-proof reference suitable for a professional ISO C++ Core Guidelines manual.

Chapter 50

Safe Expressions

50.1 Purpose and scope

A “safe expression” is an expression whose meaning is:

- correct and well-defined by the C++ standard,
- readable without relying on subtle rules (precedence, sequencing, implicit conversions),
- resilient to refactoring,
- friendly to code review and static analysis.

This chapter consolidates the most safety-critical guidance from the ISO C++ Core Guidelines ES rules and the C++ standard rules governing:

- order of evaluation and sequencing,
- conversions (especially narrowing),
- pointer validity and lifetime,

- arithmetic overflow/underflow,
- casts and aliasing,
- clarity and maintainability of expression structure.

50.2 Design principle: one idea per expression

50.2.1 Keep expressions simple

Prefer expressions that:

- do one thing,
- have at most one side effect,
- can be read correctly without memorizing precedence tables.

```
// BAD: multiple ideas, hard to read, easy to mis-evaluate
if (a & 1 == 0 && ++i < n) { /*...*/ }

// GOOD: explicit intent
const bool odd = ((a & 1) != 0);
++i;
if (odd && i < n) { /*...*/ }
```

50.3 Sequencing and order of evaluation

50.3.1 Avoid expressions with unspecified or undefined sequencing

Many safety failures occur when code depends on the order in which subexpressions are evaluated.

50.3.1.1 Rule

Do not write expressions where correctness depends on a particular evaluation order unless the order is guaranteed by the language.

50.3.1.2 Safe pattern: separate side effects from value computation

```
// BAD: depends on evaluation order
f(i++, i); // do not write this

// GOOD: make order explicit
int old = i;
++i;
f(old, i);
```

50.3.2 Function arguments: do not rely on evaluation order

Even in modern C++, function argument evaluation order should not be relied upon for correctness. Make dependencies explicit.

```
// BAD: assumes a specific order of evaluation
g(x = next(), use(x));

// GOOD: explicit sequencing
auto t = next();
x = t;
g(x, use(x));
```

50.4 Operator precedence and readability

50.4.1 Parenthesize when there is any doubt

A safe expression is one that a reader will parse correctly on the first read.

```
// BAD: readers may parse incorrectly
if (flags & mask == 0) { /*...*/ }

// GOOD: explicit intent
if ((flags & mask) == 0) { /*...*/ }
```

50.4.2 Avoid cleverness with ?: and comma

Prefer clear `if` statements over nested conditional expressions when readability suffers.

```
// BAD: nested conditional expression often harms clarity
auto r = ok ? (fast ? a() : b()) : c();

// GOOD: explicit control flow
if (!ok) return c();
return fast ? a() : b();
```

50.5 Conversions and narrowing

50.5.1 Avoid narrowing conversions

Narrowing silently loses information and can turn correct code into incorrect code when inputs change.

50.5.1.1 Prefer braced initialization

```
double d = 3.14;
// int i = d;   // allowed, but loses information
// int j{d};    // narrowing: rejected by braces
```

50.5.1.2 Use explicit, deliberate conversion where needed

If you must convert, do it in one obvious place and consider validation:

```
#include <limits>
#include <stdexcept>

int to_int_checked(long long x) {
    if (x < std::numeric_limits<int>::min() ||
        x > std::numeric_limits<int>::max())
        throw std::out_of_range("to_int_checked");
    return static_cast<int>(x);
}
```

50.5.2 Avoid mixing signed and unsigned arithmetic

Mixing signed and unsigned can introduce surprising conversions and bugs (especially when values become negative).

```
#include <cstddef>
#include <vector>

// BAD: i is size_t (unsigned); i-- can wrap
void bad(const std::vector<int>& v) {
    for (std::size_t i = v.size() - 1; i >= 0; --i) { // bug
        // ...
    }
}

// GOOD: use a signed index type
void good(const std::vector<int>& v) {
    for (std::ptrdiff_t i = static_cast<std::ptrdiff_t>(v.size()) - 1; i >= 0; --i) {
        // ...
        if (i == 0) break;
    }
}
```

50.6 Pointer validity and lifetime in expressions

50.6.1 Never dereference an invalid pointer

A pointer must be:

- non-null (if dereferenced),
- not dangling,
- properly aligned for the pointed-to type,
- pointing to a live object.

50.6.1.1 Safe null-handling

```
void use(int* p) {
    if (!p) return;
    *p += 1;
}
```

50.6.2 Do not produce dangling references or pointers

Avoid returning or storing pointers/references to locals or temporaries.

```
// BAD: dangling reference
const int& bad_ref() {
    return 1;
}

// GOOD: return by value
int good_val() {
    return 1;
}
```

50.6.3 Prefer safe views for ranges

Use `std::span` and `std::string_view` as *non-owning views*, but only when the underlying object outlives the view.

```
#include <span>
#include <vector>

int sum(std::span<const int> s) {
    int r = 0;
    for (int x : s) r += x;
    return r;
}

void demo(std::vector<int>& v) {
    auto s = std::span<const int>(v);
    (void)sum(s);
}
```

50.7 Casts: keep them rare and explicit

50.7.1 Avoid casts

Casts often indicate a design problem. Prefer:

- correct types,
- overloads/templates,
- constructors,
- standard library facilities.

50.7.2 If you must cast, use a named cast

Named casts are safer because they are explicit and searchable.

```
double d = 3.0;
int i = static_cast<int>(d); // explicit and visible
```

50.7.3 Avoid casting away const

Casting away `const` violates interfaces and can be undefined behavior if the original object is const.

```
void f(const int* p) {
    // BAD: breaks const correctness; can be UB if *p is truly const
    // int* q = const_cast<int*>(p);
}
```

50.8 Arithmetic safety

50.8.1 Signed overflow and division by zero

- Signed overflow is undefined behavior.
- Division/modulo by zero is undefined behavior.

50.8.1.1 Prefer wider types or check

```
#include <stdexcept>

int div_checked(int a, int b) {
    if (b == 0) throw std::invalid_argument("division by zero");
    return a / b;
}
```

50.8.2 Avoid unsigned wraparound as “math”

Unsigned wraparound is defined, but often produces unintended results when used as arithmetic. Use unsigned primarily for bit manipulation.

50.9 Assignments, side effects, and clarity

50.9.1 Avoid assignment inside conditions unless it is unmistakable

If you do it, make it obvious and localized.

```
// Acceptable pattern: assignment in initializer (clear, scoped)
if (auto n = read(); n > 0) {
    // ...
}
```

Avoid patterns that look like comparison mistakes:

```
// BAD: easily mistaken for '=='
if (x = y) { /*...*/ }
```

50.9.2 Avoid hidden side effects in expressions

Keep side effects in statements where they are visible.

```
// BAD: side effects buried
a[i++] = f();

// GOOD: visible sequencing
auto j = i;
++i;
a[j] = f();
```

50.10 Switch and fallthrough safety

50.10.1 Avoid implicit fallthrough

If fallthrough is intended, mark it explicitly.

```
switch (k) {
case 1:
    handle1();
    [[fallthrough]];
case 2:
    handle2();
    break;
default:
    handle_default();
}
```

50.11 A practical “safe expression” checklist

Use this checklist during review:

1. Does the expression depend on evaluation order or sequencing? If yes, split it.
2. Are there multiple side effects in one expression? If yes, split it.
3. Could a reader misparse precedence? If yes, parenthesize or rewrite.
4. Are there implicit conversions or narrowing? If yes, make them explicit and (if needed) checked.
5. Are signed/unsigned values mixed? If yes, choose one domain deliberately.

6. Are pointers dereferenced? If yes, ensure non-null and non-dangling and properly bounded.

7. Are casts used? If yes, prefer named casts and reconsider design.

8. Could overflow/underflow or division by zero occur? If yes, redesign or check.

50.12 Summary

Safe expressions are not about writing “defensive” or slow code. They are about writing code whose meaning is obvious, whose behavior is defined, and whose correctness does not depend on fragile rules. The ES rules guide you toward expressions that are easy to reason about, robust under change, and strongly supported by modern C++ abstractions and tools.

Chapter 51

Control Flow

51.1 Purpose and scope

Control flow determines how a program executes statements: selection, iteration, and early exits. In C++, control-flow mistakes often cause:

- missed error handling,
- resource leaks when RAII is not used correctly,
- logic that is correct only for “typical” inputs,
- hard-to-review code with hidden paths,
- subtle bugs due to fallthrough or unclear loop exits.

This chapter consolidates Core Guidelines control-flow guidance (especially ES.70–ES.87) and aligns it with modern C++ constructs.

51.2 General principle: prefer structured, readable flow

51.2.1 One entry, one exit when practical

Prefer control flow that is easy to reason about:

- structured blocks,
- clear conditions,
- limited nesting,
- explicit early returns when they simplify logic.

51.2.2 Prefer RAII to control-flow-based cleanup

Do not write control flow to manually “undo” partial work. Use RAII so cleanup happens automatically on every exit path.

```
#include <fstream>
#include <string>

void write_line(const std::string& path, const std::string& line) {
    std::ofstream out(path);     // RAII: closes automatically
    if (!out) return;            // early return is safe
    out << line << '\n';
}
```

51.3 Selection statements

51.3.1 Prefer `switch` over long `if/else if` chains (ES.70)

Use `switch` when selecting among discrete integral/enum values. It is often clearer and makes missing cases easier to spot.

```
enum class Mode { fast, safe, debug };

void run(Mode m) {
    switch (m) {
    case Mode::fast:  /*...*/ break;
    case Mode::safe:  /*...*/ break;
    case Mode::debug: /*...*/ break;
    }
}
```

51.3.2 Avoid implicit fallthrough in `switch` (ES.78)

Implicit fallthrough is a frequent source of bugs. If fallthrough is intended, make it explicit.

```
void handle(int k) {
    switch (k) {
    case 1:
        step1();
        [[fallthrough]];
    case 2:
        step2();
        break;
    default:
        step_default();
        break;
    }
}
```

51.3.3 Use `default` carefully (ES.79)

`default` can hide missing cases. Guideline-oriented practice:

- For **closed** sets (e.g., `enum class` where all values are known), prefer handling all enumerators explicitly.
- Use `default` when there is a genuine "catch-all" behavior (often for external/untrusted inputs).

51.3.4 `if` with initializer (modern C++)

Prefer `if` with initializer for scope control and readability.

```
#include <map>
#include <string>

void demo(std::map<std::string,int>& m, const std::string& key) {
    if (auto it = m.find(key); it != m.end()) {
        use(it->second);
    }
}
```

51.4 Iteration statements

51.4.1 Prefer range-for for simple iteration (ES.71)

Range-for reduces indexing errors and avoids signed/unsigned pitfalls.

```
#include <vector>

void print_all(const std::vector<int>& v) {
```

```
    for (const int x : v) {
        print(x);
    }
}
```

51.4.2 Prefer `for` when there is an obvious loop variable (ES.72)

Use a classic `for` when the loop is controlled by an index or counter.

```
void fill(int* a, int n, int value) {
    for (int i = 0; i < n; ++i) {
        a[i] = value;
    }
}
```

51.4.3 Prefer `while` when there is no obvious loop variable (ES.73)

Use `while` for sentinel-driven loops or state-machine-like repetition.

```
while (auto line = read_line()) {
    consume(*line);
}
```

51.4.4 Declare loop variables in the initializer (ES.74)

Limit scope and avoid accidental reuse.

```
for (int i = 0; i < n; ++i) {
    // i exists only here
}
```

51.4.5 Avoid `do/while` (ES.75)

`do/while` often hides the first-check logic and is easy to misread. Prefer `while` or restructure unless the post-condition form is truly the clearest expression.

51.5 Exits, breaks, continues, and structure

51.5.1 Avoid `goto` (ES.76)

`goto` bypasses structure and makes reasoning about execution paths difficult. It can also interact badly with initialization and object lifetimes.
Prefer:

- functions,
- loops with clear conditions,
- early returns,
- exceptions for exceptional failure,
- RAII for cleanup.

51.5.2 Minimize `break` and `continue` (ES.77)

Overuse can hide invariants and complicate reasoning. Use them when they clearly improve clarity; otherwise rewrite.

```
// Acceptable: clear early-skip
for (int x : v) {
    if (x < 0) continue;
    use(x);
}
```

51.5.3 Avoid modifying the loop control variable in the body (ES.86)

Changing the loop variable inside the loop violates reader expectations and can produce hard-to-find bugs.

```
// BAD: modifies loop variable inside loop body
for (int i = 0; i < n; ++i) {
    if (skip(i)) i += 2; // surprising
}
```

51.5.4 Prefer simple conditions; avoid redundant comparisons (ES.87)

Avoid writing `if (flag == true)`. Write `if (flag)`.

```
bool ready();
if (ready()) {
    start();
}
```

51.6 Empty statements and intentional emptiness (ES.85)

If a loop body is intentionally empty, make it obvious.

```
while (poll()) {
    // intentionally empty: waiting for readiness
}
```

51.7 Nesting and complexity control

51.7.1 Prefer flattening over deep nesting

Deeply nested control flow is difficult to review and maintain. Common techniques:

- guard clauses (early returns),
- extracting helper functions,
- using algorithms,
- using `switch` for multi-way selection.

```
bool process(Request& r) {
    if (!r.valid()) return false;
    if (!authorize(r)) return false;
    if (auto data = load(r); !data) return false;
    return handle(*data);
}
```

51.8 Control flow and correctness hazards

51.8.1 Fallthrough hazards

Accidental fallthrough is one of the most common `switch` defects. Always:

- add `break`, or
- group cases intentionally, or
- use `[[fallthrough]]`.

51.8.2 Loop termination and off-by-one

Use clear bounds, prefer half-open ranges, and prefer algorithms or range-for where possible.

```
for (int i = 0; i < n; ++i) { /* half-open [0,n) */ }
```

51.8.3 Control flow and resource safety

If a function has multiple exit paths, RAII makes them safe. Without RAII, multiple exits frequently cause leaks or double cleanup. Guideline practice: use RAII, then use early returns freely when they improve clarity.

51.9 Control flow patterns (recommended)

51.9.1 Guard clauses (early return)

```
bool parse(const std::string& s) {
    if (s.empty()) return false;
    if (!looks_ok(s)) return false;
    return do_parse(s);
}
```

51.9.2 Centralized decision with `switch`

```
enum class Token { plus, minus, star, slash };

int precedence(Token t) {
    switch (t) {
    case Token::plus:  return 1;
    case Token::minus: return 1;
    case Token::star:  return 2;
    case Token::slash: return 2;
    }
    return 0; // for completeness (unreachable if Token is closed and handled)
}
```

51.9.3 Loop with explicit termination condition

```
while (true) {
    auto item = next();
    if (!item) break;
    consume(*item);
}
```

51.10 Checklist for control-flow reviews

1. Are all paths easy to see? Avoid deeply nested logic.
2. Are `switch` statements complete and safe (no accidental fallthrough)?
3. Is `default` used appropriately (not hiding missing cases)?
4. Are loop variables scoped and not modified inside the body?
5. Is the loop termination condition obvious and correct (avoid off-by-one)?
6. Are empty bodies intentional and clearly marked?
7. Are early exits safe (RAII ensures cleanup)?
8. Are `break`/`continue` used sparingly and clearly?
9. Is `goto` avoided in favor of structured constructs?

51.11 Summary

The Core Guidelines approach to control flow is simple:

- Prefer structured constructs that make paths obvious.

- Prefer `switch` for multi-way selection and avoid implicit fallthrough.
- Prefer range-for and clear loop forms.
- Keep scopes small and use early returns when they simplify logic.
- Let RAII guarantee cleanup so control flow can focus on intent rather than manual resource handling.

Chapter 52

`if` and `switch`

52.1 Purpose and guideline focus

Selection statements control which path executes based on conditions. The ISO C++ Core Guidelines emphasize selection code that is:

- readable on first inspection,
- robust against future modification,
- explicit about intent (especially fallthrough and completeness),
- safe with respect to lifetimes and resources (RAII),
- friendly to tools (warnings, static analysis).

This chapter concentrates on the ES rules most directly related to selection:

- ES.70 (prefer `switch` when appropriate),
- ES.78 (avoid implicit fallthrough),

- ES.79 (use `default` only for common cases),
- plus closely related safety rules (operator precedence, conversions, and clarity).

52.2 `if`: precise meaning and modern forms

52.2.1 Basic `if`/`else`

An `if` executes a statement if the condition evaluates to true; otherwise the `else` branch runs (if present).

```
void f(bool ok) {
    if (ok) {
        do_work();
    } else {
        handle_error();
    }
}
```

52.2.2 Make conditions safe and obvious

52.2.2.1 Avoid precedence traps

Parenthesize when it improves clarity.

```
// BAD: easy to misread due to precedence
if (flags & mask == 0) { /*...*/ }

// GOOD: explicit intent
if ((flags & mask) == 0) { /*...*/ }
```

52.2.2.2 Avoid assignment-in-condition mistakes

Do not write code that can be confused with ==.

```
// BAD: common bug pattern
// if (x = y) { ... }

// GOOD: express intent clearly
x = y;
if (x != 0) {
    use(x);
}
```

52.2.3 `if` with initializer (C++17 and later)

Use `if (init; condition)` to limit scope and avoid separate declarations.

```
#include <map>
#include <string>

void demo(std::map<std::string,int>& m, const std::string& key) {
    if (auto it = m.find(key); it != m.end()) {
        use(it->second);
    } else {
        handle_missing();
    }
}
```

52.2.4 Guard clauses: early returns for clarity

When error handling or preconditions are simple, early exits reduce nesting and improve readability. RAII ensures resources are cleaned up automatically.

```
#include <string>

bool parse_and_process(const std::string& s) {
    if (s.empty()) return false;
    if (!looks_valid(s)) return false;
    return process(s);
}
```

52.3 When `if` is the right tool

Prefer `if` when:

- you have one or two branches,
- conditions are not simple discrete values,
- conditions involve ranges, predicates, or complex checks,
- branch structure depends on multiple variables.

```
void classify(int x) {
    if (x < 0) {
        negative();
    } else if (x == 0) {
        zero();
    } else {
        positive();
    }
}
```

52.4 `switch`: precise meaning and good usage

52.4.1 What `switch` does

A `switch` selects a branch based on the value of an integral or enumeration expression. It is often clearer than a long `if/else if` chain when selecting among many discrete cases.

52.4.2 Prefer `switch` for multi-way selection (ES.70)

When the choice is based on one value with many possible discrete outcomes, `switch` is typically the best expression of intent.

```
enum class Mode { fast, safe, debug };

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

52.4.3 Avoid implicit fallthrough (ES.78)

Accidental fallthrough is a common defect. If fallthrough is intended, make it explicit.

```
void handle(int k) {
    switch (k) {
    case 1:
        step1();
        [[fallthrough]];
    case 2:
        step2();
        break;
    default:
```

```
        step_default();
        break;
    }
}
```

52.4.4 Grouping cases safely

If multiple cases share the same action, group them explicitly.

```
enum class Token { plus, minus, star, slash };

int precedence(Token t) {
    switch (t) {
    case Token::plus:
    case Token::minus:
        return 1;
    case Token::star:
    case Token::slash:
        return 2;
    }
    return 0; // for completeness if needed
}
```

52.4.5 Use `default` carefully (ES.79)

52.4.5.1 Closed set: prefer handling all enumerators

For `enum class` where you expect all values to be handled explicitly, a `default` can hide missing cases when the enum grows.

```
enum class Color { red, green, blue };

void paint(Color c) {
```

```
    switch (c) {
    case Color::red:   use_red();   break;
    case Color::green: use_green(); break;
    case Color::blue:  use_blue();  break;
    }
}
```

52.4.5.2 Open/unknown input: `default` is appropriate

When values can be unexpected (external input, corrupted data, protocol versions), `default` provides a safe fallback.

```
void parse_opcode(int op) {
    switch (op) {
    case 0x01: op_a(); break;
    case 0x02: op_b(); break;
    default:
        report_unknown(op);
        break;
    }
}
```

52.4.6 `switch` and initialization safety

Keep resource acquisition outside `case` labels unless you are careful with scopes. A safe pattern is to use braces to create a block per case when declaring variables.

```
void example(int k) {
    switch (k) {
    case 1: {
        auto x = compute();
        use(x);
        break;
```

```
    }
    case 2: {
        auto y = compute_other();
        use(y);
        break;
    }
    default:
        break;
    }
}
```

52.5 `if` vs `switch`: choosing correctly

52.5.1 Prefer `switch` when

- many branches depend on one discrete value (especially enums),
- you want to make coverage of all cases visible,
- fallthrough behavior (if any) must be explicit.

52.5.2 Prefer `if` when

- conditions are ranges, predicates, or multiple variables,
- logic is not a single “value dispatch”,
- only a small number of branches exist.

52.6 Common pitfalls and safe rewrites

52.6.1 Pitfall: long if-else chains for discrete values

```
// Less clear for discrete dispatch
if (m == Mode::fast) run_fast();
else if (m == Mode::safe) run_safe();
else if (m == Mode::debug) run_debug();
```

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

52.6.2 Pitfall: missing break

Always ensure case termination is explicit.

```
// BAD: accidental fallthrough
switch (k) {
case 1: step1();
case 2: step2(); break;
}
```

```
// GOOD
switch (k) {
case 1: step1(); break;
case 2: step2(); break;
}
```

52.6.3 Pitfall: default hides missing enum cases

Avoid `default` in closed enum switches if you want compilers/tools to warn about unhandled enumerators (where supported).

52.7 Review checklist

1. Are `if` conditions free of precedence ambiguity?
2. Are assignments in conditions avoided (or expressed with initializer form)?
3. Is nesting minimized (guard clauses where appropriate)?
4. For multi-way dispatch, is `switch` used instead of long if-chains?
5. In every `switch`: are all fallthroughs explicit?
6. Is `default` used only when appropriate (unknown/open inputs)?
7. Are variables declared safely within case blocks (use braces)?

52.8 Summary

`if` is best for general conditional logic, range checks, and small branching. `switch` is best for multi-way dispatch on one discrete value, especially enumerations. For safe, maintainable selection code:

- keep conditions clear and parenthesized when needed,
- avoid implicit fallthrough and make intent explicit,
- use `default` carefully,
- prefer structured flow and RAII so all paths are safe.

Chapter 53

Loops

53.1 Purpose and guideline focus

Loops are central to performance and correctness in C++. They are also a frequent source of:

- off-by-one errors,
- invalid iterator/pointer access,
- unsigned underflow and wraparound,
- complex control flow (hidden exits),
- performance regressions due to unnecessary work.

The ISO C++ Core Guidelines recommend loop styles that are:

- easy to read and review,
- correct by construction (clear bounds and lifetimes),

- safe (no undefined behavior from invalid access),
- expressive (use algorithms and range-based loops when possible).

This chapter aligns especially with ES.71–ES.77, ES.85–ES.87, and the arithmetic safety rules ES.100–ES.107.

53.2 Choose the right loop form

53.2.1 Prefer range-for for iterating over a range (ES.71)

Use range-for when you simply need to visit each element in a range. It reduces indexing bugs and avoids common signed/unsigned mistakes.

```
#include <vector>

void process_all(const std::vector<int>& v) {
    for (int x : v) {
        use(x);
    }
}
```

53.2.2 Prefer `for` when there is an obvious loop variable (ES.72)

Use a classic `for` when a counter or index is part of the logic.

```
void fill(int* a, int n, int value) {
    for (int i = 0; i < n; ++i) {
        a[i] = value;
    }
}
```

53.2.3 Prefer `while` when there is no obvious loop variable (ES.73)

Use `while` for sentinel-based loops and streaming-style processing.

```
while (auto item = next_item()) {
    consume(*item);
}
```

53.2.4 Declare loop variables in the initializer (ES.74)

Keep scope minimal and intent clear.

```
for (int i = 0; i < n; ++i) { /*...*/ }
```

53.2.5 Avoid `do/while` unless it is the clearest (ES.75)

`do/while` checks the condition after executing the body, which can be surprising. Prefer `while` unless a post-condition loop is truly the best expression.

53.3 Prefer algorithms when possible (ES.1, ES.2)

Many loops are better expressed with standard algorithms, which:

- reduce boilerplate,
- make intent explicit,
- are well-tested and often optimized,
- reduce the chance of indexing and boundary bugs.

```
#include <algorithm>
#include <vector>

bool any_negative(const std::vector<int>& v) {
    return std::any_of(v.begin(), v.end(), [](int x){ return x < 0; });
}
```

53.4 Bounds, indices, and signedness safety

53.4.1 Prefer half-open ranges

Use `[0, n)` style loops. It is a stable convention and avoids fencepost errors.

```
for (int i = 0; i < n; ++i) { /* i in [0,n) */ }
```

53.4.2 Avoid unsigned indices for subscripts (ES.107)

Using unsigned indices can underflow and wrap to a huge value.

```
#include <vector>
#include <cstddef>

// BAD: underflow when v.size() == 0; also wrap when i-- reaches 0
void bad_reverse(const std::vector<int>& v) {
    for (std::size_t i = v.size() - 1; i >= 0; --i) { // bug
        use(v[i]);
    }
}
```

Prefer a signed index type for subscripting.

```
#include <vector>
#include <cstddef>
```

```
#include <limits>

void good_reverse(const std::vector<int>& v) {
    if (v.empty()) return;
    for (std::ptrdiff_t i = static_cast<std::ptrdiff_t>(v.size()) - 1; i >= 0; --i) {
        use(v[static_cast<std::size_t>(i)]);
        if (i == 0) break;
    }
}
```

53.4.3 Avoid mixing signed and unsigned arithmetic (ES.100)

Signed/unsigned mixing is a common source of loop bugs (especially when values can be negative).

```
int i = -1;
std::size_t n = 10;
// BAD: i is converted to size_t, becoming a very large value
if (static_cast<std::size_t>(i) < n) { /* surprising */ }
```

53.5 Pointer/iterator validity inside loops

53.5.1 Do not dereference invalid pointers (ES.65) or invalid iterators

Loop bodies often access elements repeatedly. Ensure validity:

- do not run past end,
- do not use iterators after invalidation,
- do not keep pointers/references across container operations that can reallocate.

```
#include <vector>

void invalidation_example() {
    std::vector<int> v{1,2,3};
    int* p = &v[0];      // observer
    v.push_back(4);      // may reallocate; p may now be dangling
    // *p;               // unsafe
}
```

53.5.2 Prefer range-for or indices that are re-derived when needed

If the container might reallocate or erase elements, either:

- avoid holding pointers/iterators across those operations, or
- design the algorithm so invalidation cannot happen within the loop.

53.6 Loop exits and control-flow clarity

53.6.1 Minimize `break` and `continue` (ES.77)

Use `break`/`continue` when they clearly improve readability; avoid overuse.

```
for (int x : v) {
    if (x < 0) continue; // clear skip
    use(x);
}
```

53.6.2 Avoid `goto` (ES.76); use structured exits

If you need to leave nested loops, prefer:

- extracting a function and returning,

- using a flag or a `break` strategy with clear structure,
- using algorithms.

```
bool find_and_handle(const std::vector<int>& v) {
    for (int x : v) {
        if (match(x)) {
            handle(x);
            return true; // structured early exit
        }
    }
    return false;
}
```

53.6.3 Avoid modifying the loop control variable inside the body (ES.86)

It violates the loop contract and is hard to review.

```
// BAD
for (int i = 0; i < n; ++i) {
    if (skip(i)) i += 2; // surprising and fragile
}
```

53.7 Empty loop bodies (ES.85)

If the loop body is intentionally empty, make it obvious.

```
while (!ready()) {
    // intentionally empty: waiting
}
```

53.8 Performance without sacrificing safety

53.8.1 Avoid unnecessary work in the condition

If a loop condition recalculates expensive values, compute them once.

```
// BETTER: compute once
const auto limit = compute_limit();
for (int i = 0; i < limit; ++i) {
    work(i);
}
```

53.8.2 Prefer preincrement for iterators when appropriate

For some iterator types, `++it` can be more efficient than `it++`. For most modern iterators, compilers optimize well, but the guideline still favors `++it` by default.

53.9 Loop patterns that are safe by construction

53.9.1 Counting loop

```
for (int i = 0; i < n; ++i) { /*...*/ }
```

53.9.2 Range loop

```
for (auto& e : container) { /*...*/ }
```

53.9.3 Sentinel loop

```
while (auto msg = receive()) { handle(*msg); }
```

53.9.4 Search loop with early return

```
bool contains(const std::vector<int>& v, int key) {
    for (int x : v) {
        if (x == key) return true;
    }
    return false;
}
```

53.10 Review checklist for loops

1. Is a standard algorithm clearer than the loop?
2. If iterating a range, is range-for used where appropriate?
3. Are bounds clear and half-open (`[0,n)`)?
4. Are signed/unsigned conversions avoided?
5. Are indices safe (no underflow, no wraparound)?
6. Are pointers/iterators always valid (no invalidation, no out-of-range access)?
7. Are exits (`break`/`continue`) minimal and clear?
8. Is the loop control variable modified only in the loop header?
9. Are empty bodies clearly intentional?

53.11 Summary

Core Guidelines loop advice can be summarized as:

- Prefer higher-level constructs (algorithms, range-for).
- Choose `for` vs `while` based on whether a loop variable exists.
- Keep loop variables scoped and do not modify them in the body.
- Avoid signed/unsigned pitfalls and unsafe indexing.
- Ensure pointer/iterator validity throughout the loop.
- Keep exits obvious and structure your loop for readability and safety.

Chapter 54

Implicit Conversions

54.1 Purpose and guideline focus

An **implicit conversion** is a conversion the compiler performs automatically to make an expression type-correct. Implicit conversions are a powerful feature of C++, but they are also a major source of:

- silent loss of information (narrowing),
- surprising sign changes (signed/unsigned mixing),
- unexpected overload resolution results,
- pointer mistakes and undefined behavior when casts are abused,
- logic bugs where code compiles but does not mean what the author intended.

The ISO C++ Core Guidelines treat implicit conversions as something to **control and make visible** when risk exists:

- prefer initialization that rejects narrowing,
- avoid ambiguous overload sets that invite surprising conversions,
- prefer explicit conversions (named casts or constructors) when meaning matters,
- design interfaces that minimize dangerous implicit conversions.

54.2 What counts as an implicit conversion

Common implicit conversions include:

- **Integral promotions:** small integer types (e.g., `char`, `short`) are promoted to `int` or `unsigned int`.
- **Usual arithmetic conversions:** operands are converted to a common type in arithmetic expressions.
- **Signed/unsigned conversions:** when mixing signed and unsigned, the signed operand may convert to unsigned.
- **Boolean context conversions:** many types can be used in conditions via implicit conversion to `bool`.
- **Pointer conversions:** e.g., `T*` to `void*`, derived-to-base pointer conversions.
- **User-defined conversions:** converting constructors and conversion operators.

54.3 Why implicit conversions are risky

54.3.1 They can silently change values

54.3.1.1 Narrowing (loss of information)

```
double d = 3.99;
int i = d;        // implicit conversion: value becomes 3 (loss of fractional part)
```

54.3.1.2 Sign conversion surprises

```
#include <cstddef>

int i = -1;
std::size_t n = 10;

if (i < static_cast<int>(n)) { // OK: compare in signed domain
    // ...
}

if (static_cast<std::size_t>(i) < n) { // i converts to a huge unsigned value
    // usually false; surprising
}
```

54.3.2 They can change overload selection

Overload resolution can pick a function you did not expect because an implicit conversion made it “viable”.

```
#include <string>

void log(int);
void log(const std::string&);

log("hello"); // selects std::string overload via implicit conversion from const char*
```

This may be intended, but it can also be unintended if it triggers allocation or changes performance characteristics.

54.3.3 They can make code ambiguous or fragile

Adding a new overload later may change which function call binds, without changing call sites.

54.4 Core Guidelines: controlling implicit conversions

54.4.1 Avoid narrowing conversions (ES.46, ES.64)

Prefer braced initialization because it rejects narrowing.

```
double d = 3.14;
int a = d;   // allowed but narrows (unsafe)
int b{d};    // error: narrowing (safer)
```

54.4.2 Prefer explicit conversions when the meaning matters

When a conversion is meaningful and intentional, make it visible with:

- `static_cast` for numeric conversions,
- named helper functions for checked conversions,
- explicit constructors for user-defined types.

```
double d = 3.14;
int i = static_cast<int>(d); // explicit, visible choice
```

54.4.3 Avoid mixing signed and unsigned arithmetic (ES.100–ES.107)

Avoid expressions where conversion to unsigned can produce unexpected results.

```cpp
#include <vector>
#include <cstddef>

bool contains_bad(const std::vector<int>& v, int idx) {
    // BAD: if idx is negative, it is converted to size_t in v[idx]
    return v[static_cast<std::size_t>(idx)] == 0;
}
```

Prefer keeping indices in a signed domain and validating:

```cpp
#include <vector>
#include <cstddef>

bool contains_good(const std::vector<int>& v, int idx) {
    if (idx < 0) return false;
    const auto u = static_cast<std::size_t>(idx);
    if (u >= v.size()) return false;
    return v[u] == 0;
}
```

54.4.4 Avoid implicit pointer conversions that hide ownership or lifetime

A raw pointer is typically an observer. Do not let implicit conversions obscure ownership boundaries. Prefer explicit owners (`unique_ptr`, `shared_ptr`) and explicit borrowing interfaces.

54.5 Safe numeric conversion patterns

54.5.1 Prefer widening conversions

Widening conversions (e.g., `int` to `long long`, `float` to `double`) are typically safe.

```
int x = 100;
long long y = x; // widening, safe
```

54.5.2 For potentially narrowing conversions, check

When converting from a wider to a narrower type, validate the range.

```
#include <limits>
#include <stdexcept>

int to_int_checked(long long x) {
    if (x < std::numeric_limits<int>::min() ||
        x > std::numeric_limits<int>::max())
        throw std::out_of_range("to_int_checked");
    return static_cast<int>(x);
}
```

54.5.3 Prefer `std::byte` for raw byte data

Using `char` or `unsigned char` as "byte" can invite arithmetic and signedness confusion. For pure byte semantics, prefer `std::byte` and explicit conversions when needed.

```
#include <cstddef> // std::byte

std::byte b{0x2A};
```

54.6 Implicit conversions in boolean contexts

54.6.1 Prefer clear boolean expressions

Conditions should be obviously boolean:

```
// OK: pointer in condition means "is not null"
if (p) { use(*p); }

// OK: explicitly boolean
if (count != 0) { /*...*/ }
```

Avoid conversions that hide meaning:

```
// Avoid: using integers as booleans when it hides intent
if (count) { /*...*/ }  // sometimes fine, but prefer explicit when it aids clarity
```

54.7 User-defined implicit conversions (type design)

Implicit conversions are often introduced unintentionally by class design.

54.7.1 Converting constructors

A non-`explicit` single-argument constructor enables implicit conversion.

```
struct Size {
    int v;
    // BAD for many types: allows implicit conversion from int
    Size(int x) : v{x} {}
};
```

Prefer `explicit` unless implicit conversion is clearly safe and desirable.

```
struct Size {
    int v;
    explicit Size(int x) : v{x} {}
};
```

54.7.2 Conversion operators

Conversion operators can create surprising implicit conversions in overload resolution.

```
struct Buffer {
    // Risky: can participate in many conversions unexpectedly
    operator const char*() const;
};
```

Prefer named accessors:

```
struct Buffer {
    const char* c_str() const;
};
```

54.7.3 Avoid implicit conversion to bool for “smart” types unless it is the established pattern

For pointer-like types, `explicit operator bool()` is a common safe pattern because it prevents unintended arithmetic conversions while still allowing use in conditions.

```
struct Handle {
    int id = -1;
    explicit operator bool() const noexcept { return id >= 0; }
};
```

54.8 Implicit conversions and overload sets (API design)

54.8.1 Avoid overloads that differ only by signedness or narrow types

Overloads like `f(int)` vs `f(long)` can lead to confusing calls when literals are involved.

54.8.2 Prefer strongly typed enums over integers

Strong enums reduce accidental conversions:

```
enum class Permission : unsigned { read = 1, write = 2 };

void set_perm(Permission p); // safer than set_perm(unsigned)
```

54.9 When implicit conversions are acceptable

Not all implicit conversions are bad. Many are fundamental and safe:

- promotions in arithmetic when there is no loss of information,
- converting derived pointers to base pointers (polymorphism),
- implicit conversion from `T*` to `bool` in conditions (null check),
- conversion from array to pointer in legacy interfaces (though safer alternatives exist).

The rule is: **allow implicit conversion when it is obvious and safe, forbid or make explicit when it can surprise.**

54.10 Review checklist

1. Could this expression trigger narrowing? If yes, use braces or an explicit checked conversion.
2. Are signed and unsigned mixed? If yes, choose a single domain deliberately.
3. Is overload resolution sensitive to implicit conversions? If yes, redesign overloads or make conversions explicit.

4. Do class types have non-explicit converting constructors? If yes, make them `explicit` unless truly intended.
5. Do class types have conversion operators that can trigger surprising conversions? Prefer named accessors.
6. Are conditions clearly boolean? Avoid integer-as-bool when it hides meaning.

54.11 Summary

Implicit conversions are essential to C++, but unsafe or surprising implicit conversions are among the most common causes of defects. The Core Guidelines approach is:

- prevent narrowing (prefer braced initialization),
- avoid signed/unsigned pitfalls,
- prefer explicit, visible conversions where meaning matters,
- design types and APIs to avoid accidental user-defined implicit conversions,
- keep expressions simple so conversions are obvious to readers and tools.

Chapter 55

Operator Precedence

55.1 Purpose and guideline focus

Operator precedence and **associativity** determine how an expression is grouped when parentheses are not used. In C++, precedence is a frequent source of bugs because:

- many operators exist, with precedence that is not obvious,
- small visual differences can change meaning dramatically,
- code may compile and run while doing the wrong thing,
- reviewers can easily misread expressions, especially under time pressure.

The Core Guidelines recommendation (notably ES.41) is simple:

- **If there is any doubt, parenthesize.**
- Prefer clarity over cleverness; break complex expressions into named subexpressions.

This chapter explains precedence and associativity at a practical level and provides safe patterns.

55.2 Precedence vs associativity (precise meaning)

55.2.1 Precedence

Precedence determines which operators bind more tightly. Example:

```
int x = 1 + 2 * 3; // '*' has higher precedence than '+'
```

Meaning: `1 + (2 * 3)`.

55.2.2 Associativity

Associativity determines grouping when operators of the same precedence appear without parentheses. Example:

```
int x = 10 - 3 - 2; // '-' is left-associative
```

Meaning: `(10 - 3) - 2`.

55.2.3 Precedence is not evaluation order

Precedence and associativity define grouping, not the order in which subexpressions are evaluated. Do not rely on evaluation order for correctness; separate side effects.

55.3 Core guideline rule (ES.41)

55.3.1 ES.41: If in doubt about operator precedence, parenthesize

Rule. When an expression can be misread, add parentheses or rewrite it. **Reason.** Parentheses cost nothing and prevent expensive bugs.

55.4 A practical precedence map (most-used operators)

This list is intentionally practical rather than exhaustive. It highlights the operators that commonly appear in real code and commonly cause confusion.
From higher binding to lower binding (selected groups):

1. **Postfix:** `expr++`, `expr--`, `f(...)`, `a[b]`, `p->m`, `p.m`
2. **Unary:** `++x`, `--x`, `+x`, `-x`, `!x`, `~x`, `*p`, `&x`, casts, `sizeof`
3. **Multiplicative:** `*`, `/`, `%`
4. **Additive:** `+`, `-`
5. **Shifts:** `<<`, `>>`
6. **Relational:** `<`, `<=`, `>`, `>=`
7. **Equality:** `==`, `!=`
8. **Bitwise AND/XOR/OR:** `&`, `^`, `|`
9. **Logical AND/OR:** `&&`, `||`
10. **Conditional:** `?:`
11. **Assignment:** `=`, `+=`, `-=`, `*=`, …
12. **Comma:** `,`

55.5 The precedence mistakes that cause real bugs

55.5.1 Bitwise vs equality (classic trap)

```
// BUG: '==' binds tighter than '&'
if (flags & mask == 0) { /*...*/ }
```

Parsed as:

```
// actual grouping
if (flags & (mask == 0)) { /*...*/ }
```

Correct intent is usually:

```
// GOOD: explicit and correct
if ((flags & mask) == 0) { /*...*/ }
```

55.5.2 Shift vs addition

```
// Might be misread; readers must recall precedence
auto x = 1 << n + 1;
```

This is parsed as:

```
auto x = 1 << (n + 1);
```

Write what you mean:

```
auto x = 1 << (n + 1);      // explicit
auto y = (1 << n) + 1;      // different meaning, also explicit
```

55.5.3 Logical vs bitwise operators

```
// BUG: '&' is bitwise AND, not logical AND
if (a & b) { /*...*/ } // often unintended when a and b are booleans or predicates
```

Prefer:

```
if (a && b) { /*...*/ }
```

If bitwise is intended, parenthesize and comment:

```
if ((flags & needed) != 0) { /* bit test */ }
```

55.5.4 Assignment inside conditions

This is not a precedence issue only; it is a readability hazard.

```
// BAD: can be mistaken for '=='
if (x = y) { /*...*/ }
```

Safer forms:

```
// GOOD: make intent obvious
x = y;
if (x != 0) { /*...*/ }
```

Or use an initializer when that expresses the pattern:

```
if (auto n = read(); n > 0) {
    use(n);
}
```

55.5.5 The conditional operator ?:

`?:` has low precedence and can be difficult to read when nested.

```
// Hard to read when nested
auto r = ok ? (fast ? a() : b()) : c();
```

Prefer an `if` statement if it improves clarity:

```
if (!ok) return c();
return fast ? a() : b();
```

55.5.6 Comma operator

The comma operator has the lowest precedence and is rarely appropriate in ordinary code. Avoid it outside special contexts (e.g., for-loop headers) because it is often misread.

55.6 Safe patterns recommended by the guidelines

55.6.1 Parenthesize bit tests and masks

```
bool has_all(unsigned x, unsigned mask) {
    return (x & mask) == mask;
}
bool has_any(unsigned x, unsigned mask) {
    return (x & mask) != 0;
}
```

55.6.2 Name intermediate results

When expressions become non-trivial, naming improves clarity and reduces mistakes.

```
const bool has_permission = ((flags & perm_mask) != 0);
const bool in_range = (x >= low) && (x < high);
if (has_permission && in_range) { /*...*/ }
```

55.6.3 Avoid clever expressions with side effects

Precedence does not protect you from sequencing hazards. Separate side effects:

```
// BAD: compact but fragile
arr[i++] = f() + g(i);

// GOOD: explicit sequencing and intent
auto j = i;
++i;
arr[j] = f() + g(i);
```

55.7 Precedence and overload resolution (brief note)

Operator overloading does not change precedence. Even overloaded operators follow the same precedence and associativity rules as built-in operators. This makes parentheses and clarity just as important for user-defined types.

55.8 Review checklist

1. Could a reader misinterpret grouping? If yes, add parentheses.
2. Does the expression mix bitwise operators with comparisons? Parenthesize explicitly.
3. Does the expression involve shifts with arithmetic? Parenthesize.
4. Are `&&` and `||` intended, not `&` and `|`?

5. Is `?:` nested or complex? Prefer `if`.
6. Is the comma operator used outside a loop header? Rewrite.
7. Does the expression include side effects (`++`, `--`, assignment)? Consider splitting.

55.9 Summary

Operator precedence is a language rule that programmers often remember imperfectly, while compilers remember perfectly. The safe approach is:

- parenthesize when there is any doubt,
- prefer clear, small expressions,
- name intermediate values,
- avoid mixing bitwise operators with comparisons without explicit parentheses,
- never rely on cleverness for readability.

Part IX

Naming and Layout (NL)

Chapter 56

NL.1 – NL.26: Naming and Layout

56.1 Scope and intent of the NL rules

The C++ Core Guidelines treat Naming and Layout as **suggested defaults** that you may adopt **when you have no external constraints or better ideas**. The real goal is **consistency** across a codebase: consistent naming, consistent indentation, and consistent formatting reduce review time, reduce misunderstandings, and prevent subtle mistakes that hide in messy code.

56.2 Rule index (current guideline revision)

In the current Core Guidelines Naming and Layout section (revision dated July 8, 2025), the following NL rules are present: `NL.1--NL.5, NL.7--NL.11, NL.15--NL.21, NL.25--NL.26`.
The numbers `NL.6, NL.12, NL.13, NL.14, NL.22, NL.23, NL.24` are **not present** in this revision. Their absence reflects historical edits while keeping stable numbering.

56.3 Comments: clarity without noise

56.3.1 NL.1: Don't say in comments what can be clearly stated in code

Reason. Comments are less precise than code, compilers do not check them, and they often go out of date. Prefer expressive code and naming over "explanations" that merely restate operations.

```
// BAD: comment repeats the code
auto x = m * v1 + vv; // multiply m with v1 and add the result to vv

// GOOD: make the code self-explanatory
auto momentum = mass * velocity + momentum_bias;
```

56.3.2 NL.2: State intent in comments

Reason. Code says what it does; a good comment can say what it is *meant* to do (requirements, invariants, assumptions, constraints). If code and comment disagree, both are suspect.

```
// Intention: keep equal elements in their original relative order (stable sort).
void stable_sort_records(std::vector<Record>& r)
{
    // ... non-trivial implementation ...
}
```

56.3.3 NL.3: Keep comments crisp

Reason. Verbose comments slow understanding and scatter attention. Use clear, professional language; avoid slang and "chatty" text in production code.

```
// GOOD: short and specific
// Precondition: size >= 0
```

56.4 Indentation and visual structure

56.4.1 NL.4: Maintain a consistent indentation style

Reason. Indentation improves readability and helps prevent "silly mistakes" (for example, misleading control-flow and visually hidden bugs). Use a formatting tool if possible.

```
// BAD: misleading layout can hide bugs
for (int i = 0; i < max; ++i);   // bug waiting to happen
if (i == j)
    return i;

// GOOD: consistent indentation makes structure visible
for (int i = 0; i < max; ++i) {
    work(i);
}
if (i == j) {
    return i;
}
```

56.5 Naming: express meaning, not types

56.5.1 NL.5: Avoid encoding type information in names

Rationale. If names describe types (e.g., `print_int`), changing types becomes painful and misleading. Names should describe **purpose and meaning**.

```
// BAD: type is encoded in the name
void print_int(int i);
void print_string(const char* s);

// GOOD: meaning-based naming (overload, templates, or concepts handle types)
```

```
void print(int i);
void print(std::string_view s);
```

56.5.2 NL.6: (Not present in the July 8, 2025 revision)

Rule number reserved; no current NL.6 text is present in the Naming and Layout section in this revision.

56.5.3 NL.7: Make the length of a name roughly proportional to the length of its scope

Rationale. The larger the scope, the higher the chance of confusion and unintended name clashes. Global and long-lived entities deserve clearer names; short-lived loop indices can be short.

```
// BAD: cryptic global
int g;

// BETTER: meaningful global
int connection_timeout_ms;

// OK: short name in a small scope
for (int i = 0; i < n; ++i) {
    // ...
}
```

56.5.4 NL.8: Use a consistent naming style

Rationale. Consistent naming increases readability and helps readers predict identifiers. Choose a “house style” and keep imported libraries in their original style.

```
// Example house style (underscore_style for functions/variables)
int max_payload_size = 0;
void parse_packet();

// Types may follow a different convention if your codebase adopts it
struct PacketParser { /* ... */ };
```

56.5.5 NL.9: Use `ALL_CAPS` for macro names only

Reason. To avoid confusing macros (which do not obey scope/type rules) with ordinary names. Do not write constants or enums in `ALL_CAPS` unless they are macros.

```
// BAD: looks like a macro
const int SIZE{1000};

// GOOD:
const int size{1000};

// GOOD: macro is ALL_CAPS
#define MAX_BUFFER_SIZE 4096
```

56.5.6 NL.10: Prefer `underscore_style` names

Reason. Underscore-separated names are the original C/C++ style and are used by the C++ standard library. This is a default suggestion; consistency beats personal taste.

```
int error_count = 0;
void read_file();
```

56.5.7 NL.11: Make literals readable

Reason. Readability. Use digit separators for long numbers and use suffixes when they clarify meaning. Avoid scattering “magic constants” throughout the code; make important constants

named and centralized.

```
#include <chrono>
using namespace std::chrono_literals;

auto speed_of_light = 299'792'458;       // readable digits
auto mask           = 0b0000'1111'0000'0000;
auto greeting       = "Hello!"s;         // std::string literal
auto interval       = 100ms;             // chrono duration literal
```

56.5.8 NL.12: (Not present in the July 8, 2025 revision)

Rule number reserved; no current NL.12 text is present in this revision.

56.5.9 NL.13: (Not present in the July 8, 2025 revision)

Rule number reserved; no current NL.13 text is present in this revision.

56.5.10 NL.14: (Not present in the July 8, 2025 revision)

Rule number reserved; no current NL.14 text is present in this revision.

56.6 Whitespace, declarations, and layout defaults

56.6.1 NL.15: Use spaces sparingly

Reason. Too much space makes text larger and distracts. Some IDEs insert distracting spaces; apply a consistent formatter.

```
// BAD
#include < map >
```

```
int main(int argc, char * argv [ ])

// GOOD
#include <map>
int main(int argc, char* argv[])
```

56.6.2 NL.16: Use a conventional class member declaration order

Reason. A conventional order improves readability. The guidelines suggest:

- types (nested classes/enums/aliases),
- constructors/assignments/destructor,
- functions,
- data members,

and also: `public` before `protected` before `private`.

```
class widget {
public:
    using id_type = int;

    widget();
    widget(const widget&);
    widget& operator=(const widget&);
    ~widget();

    void start();
    void stop();

private:
    id_type id_{};
```

```
    int state_{0};
};
```

56.6.3 NL.17: Use K&R-derived layout

Reason. This is the original C and C++ layout, preserves vertical space, and distinguishes constructs well. In C++ it is often called "Stroustrup" style.

```
int f(int x)
{
    if (x < 0) {
        return 0;
    }
    return x + 1;
}
```

56.6.4 NL.18: Use C++-style declarator layout

Reason. Prefer a declarator layout that emphasizes types (particularly for references). The guideline illustrates: `T& operator[](size_t);` as the conventional form.

```
#include <cstddef>

template<class T>
struct vec {
    T& operator[](std::size_t);
    const T& operator[](std::size_t) const;
};
```

56.6.5 NL.19: Avoid names that are easily misread

Reason. Readability: some characters look similar in many fonts (e.g., `O`, `0`, `l`, `1`). Also avoid near-identical words that differ by one character.

```
// BAD: visually confusing
int oO01lL = 6;

// BAD: easy to confuse
int splunk = 7;
int splonk = 8;

// GOOD: pick distinct names
int total_requests = 7;
int total_retries  = 8;
```

56.6.6 NL.20: Don't place two statements on the same line

Reason. It is easy to overlook a statement when there is more than one on a line.

```
// BAD
int x = 7; f(x); ++x;

// GOOD
int x = 7;
f(x);
++x;
```

56.6.7 NL.21: Declare one name (only) per declaration

Reason. Improves readability and avoids confusion with declarator syntax. (Related: ES.10 for declarator pitfalls.)

```
// BAD
int* p, q;   // q is int, not int*

// GOOD
int* p;
int  q;
```

56.6.8 NL.22: (Not present in the July 8, 2025 revision)

Rule number reserved; no current NL.22 text is present in this revision.

56.6.9 NL.23: (Not present in the July 8, 2025 revision)

Rule number reserved; no current NL.23 text is present in this revision.

56.6.10 NL.24: (Not present in the July 8, 2025 revision)

Rule number reserved; no current NL.24 text is present in this revision.

56.7 Interface notation and compatibility reminders

56.7.1 NL.25: Don't use `void` as an argument type

Reason. `void f(void)` is verbose and only relevant for C compatibility conventions. In C++, prefer `void f()` for a function taking no arguments.

```
// BAD in C++
void f(void);

// GOOD in C++
void g();
```

56.7.2 NL.26: Use conventional `const` notation

Reason. Conventional `const`-before-type notation is familiar to many programmers and supports consistency in large codebases. The guideline shows the conventional forms as suggested defaults.

```
// Conventional (recommended default style)
const int x = 7;

const int* const p = nullptr; // const pointer to const int
```

Note. Alternative placements (e.g., `int const`) can be logically consistent with pointer/reference binding, but the NL rules explicitly prioritize common teaching material and broad familiarity when you have no constraints.

56.8 Practical adoption notes

- Decide on a project style (names, braces, spaces) and apply it consistently.
- Prefer automated formatting to reduce subjective review discussions (the guidelines explicitly recommend tool support for indentation/layout enforcement).
- Keep externally imported library code in its original style, and keep your "house style" in your code.

56.9 Summary

NL rules are about **human factors**:

- Write code that is easy to read, review, and maintain.
- Use comments for intent, not for restating code.
- Avoid naming patterns that lock you into types or confuse readers.
- Keep formatting consistent and use tools to enforce it.
- Prefer simple, conventional declarations and layouts that reduce misreading.

Chapter 57

Naming Rules

57.1 Scope and intent

The C++ Core Guidelines treat naming as a **human-engineering tool**: names are part of the interface, part of the documentation, and part of the safety story. Good names reduce misunderstanding, simplify review, and reduce defects.
The Core Guidelines do not claim there is one perfect naming style. Instead, they recommend:

- choose a clear and consistent naming convention,
- keep names meaningful and readable,
- avoid names that encode types or invite confusion,
- align with established C++ ecosystem conventions where practical.

This chapter consolidates the **naming-focused** NL rules (especially NL.5, NL.7–NL.11, NL.19) and connects them to practical, modern C++ interface design.

57.2 Naming principles

57.2.1 Name meaning, not mechanics

A name should describe **what the entity represents** or **what it does**, not the implementation technique.

```
// BAD: names describe mechanics or types
void print_int(int);
void print_cstr(const char*);

// GOOD: names describe intent; types are handled by overloads/templates
void print(int);
void print(std::string_view);
```

57.2.2 Consistency beats personal preference

A consistent convention improves readability and reduces cognitive load:

- readers can predict how things are named,
- tools can enforce and search patterns,
- reviews focus on correctness rather than style debates.

57.3 Core Guidelines naming rules (NL)

57.3.1 NL.5: Avoid encoding type information in names

Rule. Do not put type information in names (Hungarian notation, suffixes like `Str`, `Int`, `Ptr`).
Reason. Types change; meaning should remain stable. Encoding types makes names lie over time and makes refactoring painful.

```
// BAD
int  count_i;
char* name_p;

// GOOD
int count;
std::string name;
```

57.3.2 NL.7: Make the length of a name roughly proportional to the length of its scope

Rule. The wider the scope and the longer the lifetime, the more descriptive the name should be. Short names are fine for small, local scopes.

```
// Good: short name in a small scope
for (int i = 0; i < n; ++i) {
    // ...
}

// Good: descriptive name for a broader scope
int connection_timeout_ms = 0;
```

57.3.3 NL.8: Use a consistent naming style

Rule. Pick a house style and apply it uniformly. Keep external libraries in their original style and adapt at boundaries if needed.

```
// Example consistent house style (underscore_style)
int max_payload_size = 0;
void parse_packet();

// Types can follow a consistent chosen pattern
```

```
struct packet_parser {
    void parse();
};
```

57.3.4 NL.9: Use `ALL_CAPS` for macro names only

Rule. Reserve `ALL_CAPS` for macros. **Reason.** Macros do not respect scope or type rules, and their names should stand out as a warning.

```
// BAD: looks like a macro
const int MAX_SIZE = 1024;

// GOOD
const int max_size = 1024;

// GOOD: macro is ALL_CAPS
#define MAX_SIZE 1024
```

57.3.5 NL.10: Prefer `underscore_style` names

Rule. Prefer `underscore_style` (snake_case) as a default naming style in C++. **Reason.** It is the original C/C++ convention and consistent with the standard library.

```
int error_count = 0;
void open_connection();
```

57.3.6 NL.11: Make literals readable

Rule. Use readable forms for literals: digit separators, binary literals (when appropriate), and meaningful suffixes (e.g., chrono durations). Also, avoid scattering magic numbers: prefer named constants.

```
#include <chrono>
using namespace std::chrono_literals;

constexpr int max_items = 10'000;
auto mask     = 0b1111'0000u;
auto timeout  = 250ms;
```

57.3.7 NL.19: Avoid names that are easily misread

Rule. Avoid names that are visually confusing in common fonts, or differ only slightly.
Reason. Readability is a correctness issue; misreading causes bugs.

```
// BAD: visually confusing
int lI10O0 = 0;

// BAD: too similar
int retries = 0;
int retrles = 0;

// GOOD: distinct and meaningful
int retry_count = 0;
int request_count = 0;
```

57.4 Practical naming rules for modern C++ codebases

These are guideline-consistent conventions commonly used in modern C++ projects. They help keep interfaces predictable.

57.4.1 Types, functions, and variables

57.4.1.1 Types

For types (classes/structs/enums/type aliases), choose one consistent convention. Common choices:

- `snake_case` for types (common in some modern libraries), or
- `PascalCase` for types (common in many corporate codebases).

The Core Guidelines do not force one; they emphasize consistency.

57.4.1.2 Functions and variables

Use action names for functions and meaning names for variables:

- functions: verbs or verb phrases (`open_file`, `compute_hash`),
- booleans: predicate forms (`is_valid`, `has_permission`),
- collections: plural (`items`, `records`),
- sizes/counts: include unit words (`count`, `size`, `bytes`, `ms`).

```
bool is_ready() noexcept;
bool has_permission(user_id u);

std::size_t byte_count = 0;
int retry_count = 0;
```

57.4.2 Avoid misleading abbreviations

Abbreviations are acceptable when they are standard and stable (e.g., `id`, `url`, `tcp`). Avoid personal abbreviations that only the author understands.

```
// BAD: personal abbreviation
int cfg_prm_cnt = 0;

// GOOD
int config_parameter_count = 0;
```

57.4.3 Do not use reserved or dangerous identifier forms

For correctness and portability, avoid identifiers that are reserved to the implementation:

- names with a leading underscore followed by an uppercase letter,
- names with two consecutive underscores anywhere,
- (at global scope) names starting with an underscore.

```
// BAD: reserved forms (avoid)
int __count = 0;
int _Max = 0;

// GOOD
int count = 0;
int max_value = 0;
```

57.4.4 Name ownership and lifetime explicitly in interfaces

When an interface involves pointers or resource handles, use names that communicate intent:

- observer pointer: `p`, `ptr` in local scope is fine, but interface names should clarify role,
- owners should be expressed by type (`unique_ptr`, `shared_ptr`) rather than by name.

```
#include <memory>
```

```
void set_callback(std::function<void()> cb);

void take_owner(std::unique_ptr<Widget> w); // ownership is in the type
void observe(const Widget* w);              // observer pointer; name can remain simple
```

57.5 Naming and overload sets (avoid confusion)

57.5.1 Prefer overload sets that preserve meaning

Overloads should represent the same conceptual operation for different types. Do not create overloads where the same name means different actions depending on type.

```
// GOOD: same meaning for different input forms
void print(int);
void print(std::string_view);

// BAD: same name, unrelated semantics
void print(int);            // prints value
void print(Socket&);        // sends to network (different meaning)
```

57.6 Checklist for naming reviews

1. Does the name express meaning, not type or mechanics? (NL.5)
2. Is the name length appropriate for its scope? (NL.7)
3. Is the naming style consistent across the module/codebase? (NL.8)
4. Are macros the only ALL_CAPS identifiers? (NL.9)
5. Are names readable, unambiguous, and not easily misread? (NL.19)

6. Are literals readable and magic numbers replaced by named constants? (NL.11)
7. Are reserved identifier forms avoided for portability?

57.7 Summary

Naming rules in the Core Guidelines are about making code **communicate reliably**:

- Choose a consistent style and apply it.
- Avoid names that encode types; types change, meaning should not.
- Use longer, clearer names for wider scopes and interfaces.
- Reserve ALL_CAPS for macros.
- Make literals readable and avoid magic numbers.
- Avoid confusing, visually similar names.

Chapter 58

Consistency

58.1 Purpose and guideline focus

In the C++ Core Guidelines, naming and layout rules are primarily about **consistency**. Consistency is not a cosmetic goal; it is an engineering tool that:

- reduces cognitive load during reading and debugging,
- makes code review faster and more reliable,
- improves tool effectiveness (formatters, linters, static analyzers),
- lowers the probability of mistakes caused by misreading.

The NL rules explicitly recommend adopting a consistent naming style (NL.8), consistent indentation/layout (NL.4), and conventional defaults when no stronger constraints exist (NL.10, NL.17, NL.26). This chapter explains how to apply those ideas across a real codebase.

58.2 What “consistency” means in practice

Consistency means that a reader can predict patterns without searching:

- how files are organized,
- how types, functions, and variables are named,
- how indentation and braces look,
- how parameters, return values, and errors are handled,
- how ownership and lifetime are expressed,
- how literals and units are written.

Consistency is strongest when it is:

- documented (short style guide),
- enforced automatically (formatter + linter),
- applied to new code immediately (no long-running inconsistency).

58.3 Core Guidelines rules that directly enforce consistency

58.3.1 NL.4: Maintain a consistent indentation style

Goal. Make structure visible and prevent misleading layouts.

```
// BAD: misleading layout can hide a bug
for (int i = 0; i < 10; ++i);  // accidental empty loop
if (ready())
    start();
```

```
// GOOD: consistent braces and indentation expose structure
for (int i = 0; i < 10; ++i) {
    work(i);
}
if (ready()) {
    start();
}
```

58.3.2 NL.8: Use a consistent naming style

Goal. Reduce guesswork about identifier meaning and category.

```
// Example consistent convention
// variables/functions: snake_case
int retry_count = 0;
void open_connection();

// types: choose one convention and stay consistent
struct connection_manager {
    void connect();
};
```

58.3.3 NL.10: Prefer underscore_style names (default suggestion)

Goal. Provide a reasonable default aligned with the standard library.

58.3.4 NL.17: Use K&R-derived layout (default suggestion)

Goal. Provide a common default brace/indent style that many C++ developers recognize.

58.3.5 NL.26: Use conventional const notation (default suggestion)

Goal. Avoid style debates and keep declarations predictable.

58.4 Consistency and correctness: where it prevents bugs

58.4.1 Consistent bracing prevents “dangling else” confusion

When braces are optional, the visual structure can mislead.

```
// BAD: visually fragile; edits can change meaning
if (ok)
    if (ready)
        start();
    else
        stop();  // binds to inner if, not outer
```

```
// GOOD: consistent braces make binding obvious
if (ok) {
    if (ready) {
        start();
    } else {
        stop();
    }
}
```

58.4.2 Consistent naming reduces misuse

If boolean functions consistently start with `is_`/`has_`, their meaning is easy to infer.

```
bool is_valid(const token& t);
bool has_permission(user_id u);
```

```
if (is_valid(t) && has_permission(u)) {
    proceed();
}
```

58.4.3 Consistent units in names prevent dangerous mistakes

When values represent sizes or time, include the unit in the name.

```
int timeout_ms = 250;
int payload_bytes = 1024;
```

58.5 Consistency at interfaces: the most important place

A local inconsistency is annoying. An interface inconsistency is expensive.

58.5.1 Consistent ownership vocabulary

Make ownership visible through types and stable naming conventions:

- owners: `std::unique_ptr<T>`, `std::shared_ptr<T>`
- observers: `T*`, `T&`, `std::span<T>`
- optional values: `std::optional<T>`

```
#include <memory>
#include <optional>
#include <span>

void take_owner(std::unique_ptr<widget> w);
void observe(const widget* w);
std::optional<widget> try_make();
int sum(std::span<const int> s);
```

This makes calling code consistent and reduces accidental misuse.

58.5.2 Consistent error-handling strategy

Choose an error-handling policy and apply it consistently:

- exceptions for failures that cannot be ignored,
- error codes or `std::optional` for expected absence or recoverable errors,
- avoid mixing patterns randomly in the same layer.

```
#include <optional>
#include <string>

std::optional<int> parse_int(std::string_view s); // expected parse failure is "no value"
```

58.6 Consistency and refactoring safety

Consistency reduces the risk that a small refactor changes meaning:

- a standard brace rule prevents accidental scope changes,
- naming conventions prevent confusing reuse,
- consistent initialization style reduces narrowing bugs,
- consistent loop conventions reduce indexing mistakes.

58.7 How to achieve consistency in a real codebase

58.7.1 Step 1: Choose a small, enforceable house style

A good house style is:

- short (one page),
- practical (covers naming, braces, indentation, line length, includes, comments),
- aligned with tool defaults (to avoid fights with formatters).

58.7.2 Step 2: Automate formatting

Use an automatic formatter and run it:

- locally (editor integration),
- in CI,
- as a pre-commit hook (optional).

58.7.3 Step 3: Automate naming and rule checks where valuable

A linter/static analyzer can enforce:

- naming patterns,
- header ordering,
- banned constructs (e.g., macros, raw `new`/`delete` in most code),
- suspicious patterns (implicit fallthrough, narrowing conversions).

58.7.4 Step 4: Fix inconsistencies at boundaries

When integrating third-party code:

- do not reformat vendor code,
- keep a consistent boundary (adapters/wrappers) that presents your house style externally.

```
// Example: adapter that translates a third-party naming style to house style
class http_client {
public:
    void send_request(const request& r); // house style interface
private:
    third_party::HttpClient impl_;       // third-party style kept inside
};
```

58.8 When consistency should yield

Consistency is a strong default, but it should yield to:

- established platform or project conventions (e.g., an existing large codebase),
- standard library conventions when interacting closely with them,
- readability improvements in a special domain (e.g., math/linear algebra code),
- external API requirements (names mandated by protocols or specifications).

Even then, keep consistency *within the chosen domain*.

58.9 Checklist

1. Is naming style consistent across modules? (NL.8)
2. Are indentation and braces consistent and tool-enforced? (NL.4, NL.17)
3. Are macros clearly marked as macros (ALL_CAPS only for macros)? (NL.9)
4. Are literals readable and units consistently represented? (NL.11)
5. Are types, functions, and variables following predictable conventions? (NL.7, NL.10)

6. Are interface conventions consistent for ownership and errors?
7. Is external code isolated behind adapters rather than forced into house style?

58.10 Summary

Consistency is one of the highest-leverage practices in C++:

- It improves correctness by preventing misreading and fragile edits.
- It improves productivity by speeding reviews and debugging.
- It improves tooling by enabling automatic formatting and reliable static checks.
- The Core Guidelines provide safe defaults (NL) and strongly recommend enforcing them consistently.

Chapter 59

Layout and Formatting

59.1 Purpose and guideline focus

Layout and formatting are not merely aesthetic. In the C++ Core Guidelines, layout exists to improve:

- readability and review speed,
- correctness (preventing misread control flow and hidden bugs),
- maintainability (low friction for refactoring),
- tool support (formatters, linters, diff tools, static analyzers).

The Core Guidelines treat formatting rules as **defaults**: pick a consistent style and apply it uniformly. Consistency is more important than which specific style you choose.
This chapter focuses on the NL rules most directly related to layout and formatting:

- NL.4 (consistent indentation),

- NL.15 (use spaces sparingly),
- NL.16 (conventional member declaration order),
- NL.17 (K&R-derived layout),
- NL.18 (C++-style declarator layout),
- NL.20 (do not place two statements on one line),
- NL.21 (one name per declaration),
- plus closely related clarity rules (bracing and structure).

59.2 Indentation and block structure

59.2.1 NL.4: Maintain a consistent indentation style

Rule. Choose an indentation style (spaces or tabs, width, brace placement) and apply it consistently. **Reason.** Consistent indentation makes structure visible and prevents misleading layouts.

59.2.1.1 Braces as a safety tool

Even when braces are optional, consistent bracing avoids fragile code and reduces the risk of "accidental scope change" during maintenance.

```
// BAD: fragile; a later edit can silently change meaning
if (ok)
    do_work();
    log(); // not controlled by the if

// GOOD: structure is explicit
```

```
if (ok) {
    do_work();
    log();
}
```

59.2.1.2 Avoid misleading whitespace

A well-known bug pattern is the accidental empty loop caused by a trailing semicolon. Consistent indentation and braces make it obvious.

```
// BAD: semicolon makes an empty loop
for (int i = 0; i < n; ++i);  // bug
do_work();

// GOOD
for (int i = 0; i < n; ++i) {
    do_work();
}
```

59.3 Whitespace and spacing

59.3.1 NL.15: Use spaces sparingly

Rule. Do not add spaces that reduce readability or create non-standard token spacing. **Reason.** Excessive spacing increases visual noise and can obscure meaning.

```
// BAD
#include < vector >
int main ( int argc , char * argv [ ] )

// GOOD
#include <vector>
int main(int argc, char* argv[])
```

59.3.2 Recommended spacing conventions

A common guideline-consistent default:

- spaces around binary operators: `a + b`, `x == y`
- no space after unary operators: `++i`, `*p`, `&x`
- no spaces inside parentheses or brackets: `f(x)`, `a[i]`
- one space after commas: `f(a, b, c)`

```
int x = a + b * c;
auto v = f(x, y, z);
auto p = &obj;
```

59.4 Brace style (layout of blocks)

59.4.1 NL.17: Use K&R-derived layout

The guideline suggests a K&R-derived style (often called Stroustrup style in C++), where:

- the opening brace follows the control statement/function on the same line,
- closing brace aligns with the beginning of the construct.

```
int f(int x)
{
    if (x < 0) {
        return 0;
    }
    return x + 1;
}
```

Note. This is a default recommendation, not a requirement. The essential rule is: choose one style and use it consistently.

59.5 Declarations and declarators

59.5.1 NL.21: Declare one name (only) per declaration

Rule. Do not declare multiple variables in one declaration when it can mislead, especially with pointers. **Reason.** Declarator syntax can surprise readers.

```
// BAD: q is int, not int*
int* p, q;

// GOOD
int* p;
int  q;
```

59.5.2 NL.18: Use C++-style declarator layout

Prefer layout that keeps reference binding obvious and declarations conventional.

```
#include <cstddef>

template<class T>
struct vec {
    T& operator[](std::size_t);
    const T& operator[](std::size_t) const;
};
```

59.5.3 Pointers and references: avoid confusing formatting

A guideline-consistent approach is to keep declarations easy to parse:

- avoid stacking pointer stars in tricky ways,
- prefer type aliases for complex pointer types,

- prefer `using` over `typedef`.

```
using byte = unsigned char;
using byte_ptr = byte*;

byte_ptr p = nullptr;
```

59.6 Statements per line

59.6.1 NL.20: Don’t place two statements on the same line

Rule. One statement per line. **Reason.** Multiple statements per line are easy to miss during review and complicate diffs.

```
// BAD
int x = 7; f(x); ++x;

// GOOD
int x = 7;
f(x);
++x;
```

59.7 Class layout and member ordering

59.7.1 NL.16: Use a conventional class member declaration order

Rule. Use a predictable order, commonly:

- public interface first (`public` then `protected` then `private`),
- type members (aliases, nested types),

- special member functions,
- other member functions,
- data members.

```
class widget {
public:
    using id_type = int;

    widget();
    widget(const widget&);
    widget& operator=(const widget&);
    ~widget();

    void start();
    void stop();

private:
    id_type id_{};
    int state_{0};
};
```

Important correctness note. Data members are initialized in the order of declaration, not the order in the constructor initializer list. A consistent member order reduces mistakes and warnings.

59.8 Line length and wrapping

The Core Guidelines emphasize readability and consistency but do not mandate a single line length. Practical guidance:

- keep lines short enough for side-by-side diffs and comfortable review,

- break long expressions at logical boundaries,
- align wrapped arguments consistently.

```
auto r = compute_value(
    input_data,
    options,
    mode);
```

59.9 Formatting as an automated decision

59.9.1 Use tools to enforce formatting

Formatting is best enforced by tools to avoid subjective debates and inconsistent patches. A tool-enforced style:

- keeps diffs clean,
- makes code review focus on correctness,
- reduces manual formatting time,
- improves uniformity in multi-author projects.

59.9.2 Format boundaries: do not reformat third-party code

If you import external code:

- keep it in its original style,
- wrap it behind adapters with your house style interfaces,
- avoid mixing styles in the same module.

59.10 Checklist for layout and formatting reviews

1. Is indentation consistent and structure obvious? (NL.4)
2. Are braces used consistently, avoiding fragile single-statement blocks?
3. Is whitespace conventional and not distracting? (NL.15)
4. Is brace style consistent across the codebase? (NL.17)
5. Are declarations clear (one name per declaration; pointer syntax not misleading)? (NL.21, NL.18)
6. Is there only one statement per line? (NL.20)
7. Are class members ordered predictably? (NL.16)
8. Are long lines wrapped consistently?
9. Is formatting enforced automatically to prevent drift?

59.11 Summary

Layout and formatting are a reliability technique:

- consistent indentation and braces prevent misreading and fragile edits,
- conventional spacing and declarator layout reduce mistakes,
- one-statement-per-line improves review quality,
- predictable class layout supports maintainability,
- automated formatting is the most effective way to keep a large codebase consistent.

Chapter 60

Readability

60.1 Purpose and scope

Readability is a primary engineering goal of the C++ Core Guidelines. Readable code is:

- easier to review correctly,
- easier to maintain and refactor safely,
- easier to test and debug,
- safer (fewer bugs survive inspection),
- more predictable for tools (formatters, linters, static analyzers).

The Naming and Layout (NL) rules contribute to readability directly, but readability also depends on how you use expressions, control flow, and interfaces. This chapter focuses on the NL guidance most directly tied to readability:

- comments as a readability tool (NL.1–NL.3),

- indentation and structure (NL.4, NL.17, NL.20),
- naming for meaning (NL.5, NL.7–NL.11, NL.19),
- declaration clarity (NL.18, NL.21),

and it connects them to practical patterns used in modern C++.

60.2 Readability principle: make intent obvious

A reader should understand what the code *means* with minimal mental decoding. The core technique is to write code that is:

- explicit about structure,
- explicit about ownership and lifetimes,
- explicit about conversions and units,
- small enough to keep in short-term memory.

60.3 Comments that improve readability

60.3.1 NL.1: Don’t say in comments what can be clearly stated in code

Comments that repeat the code add noise and become stale. Prefer better names and clearer structure instead of redundant comments.

```
// BAD: comment repeats the expression
auto x = m * v + b; // multiply m by v and add b

// GOOD: express intent using names
auto corrected_velocity = mass * velocity + bias;
```

60.3.2 NL.2: State intent in comments

Use comments to record:

- intent and motivation,
- invariants and assumptions,
- preconditions and postconditions,
- non-obvious constraints (e.g., “must be stable”, “must be lock-free”).

```
// Intention: stable order matters because records share timestamps.
void stable_sort_records(std::vector<Record>& r);
```

60.3.3 NL.3: Keep comments crisp

A good comment is short, precise, and actionable. Avoid vague or emotional comments; write them as technical documentation.

```
// Precondition: buf points to at least n readable bytes.
```

60.4 Structure and visual clarity

60.4.1 NL.4: Maintain consistent indentation style

Indentation is how humans parse block structure quickly. A consistent style makes scope boundaries and control flow obvious.

```
// BAD: structure is misleading
if (ok)
    do_work();
    log(); // always executes
```

```
// GOOD: structure matches meaning
if (ok) {
    do_work();
    log();
}
```

60.4.2 NL.17: Use K&R-derived layout as a default

The guideline suggests a common brace layout to reduce friction and argument. Regardless of the chosen style, readability depends on consistency.

60.4.3 NL.20: Don't place two statements on the same line

Multiple statements on one line are easy to miss and produce noisy diffs.

```
// BAD
x = 7; f(x); ++x;

// GOOD
x = 7;
f(x);
++x;
```

60.5 Naming for readable code

60.5.1 NL.5: Avoid encoding type information in names

Names should describe meaning, not type. This keeps code stable under refactoring and prevents names from becoming lies.

```
// BAD
int count_i = 0;
char* name_p = nullptr;

// GOOD
int count = 0;
std::string name;
```

60.5.2 NL.7: Name length proportional to scope

Global and widely used names must be descriptive. Short names are acceptable in small scopes when they are conventional.

```
int connection_timeout_ms = 250;

for (int i = 0; i < n; ++i) {
    // i is local and obvious
}
```

60.5.3 NL.8: Use a consistent naming style

Consistency reduces the reader's mental effort. Pick one convention and apply it.

60.5.4 NL.11: Make literals readable

Readable literals reduce mistakes and improve review accuracy. Use digit separators, meaningful suffixes, and named constants.

```
#include <chrono>
using namespace std::chrono_literals;

constexpr int max_records = 10'000;
```

```
auto timeout = 250ms;
auto mask    = 0b1111'0000u;
```

60.5.5 NL.19: Avoid names that are easily misread

Avoid visually ambiguous names and near-identical spellings.

```
// BAD
int lI1000 = 0;
int config_value = 0;
int config_vaiue = 0;

// GOOD
int retry_count = 0;
int request_count = 0;
```

60.6 Declaration readability (avoid syntactic traps)

60.6.1 NL.21: Declare one name per declaration

This avoids pointer-declarator traps.

```
// BAD: q is not a pointer
int* p, q;

// GOOD
int* p;
int  q;
```

60.6.2 NL.18: Use conventional declarator layout

Use a layout that makes reference and pointer binding obvious and familiar.

```
#include <cstddef>

template<class T>
struct vec {
    T& operator[](std::size_t);
    const T& operator[](std::size_t) const;
};
```

60.7 Readability patterns that scale

60.7.1 Prefer small, named steps over clever expressions

Readable code often replaces one complex line with several clear lines.

```
// LESS READABLE
if ((flags & mask) != 0 && (x + y * z) > limit && ++i < n) { /*...*/ }

// MORE READABLE
const bool has_feature = ((flags & mask) != 0);
const auto value = x + y * z;
++i;
if (has_feature && value > limit && i < n) { /*...*/ }
```

60.7.2 Guard clauses reduce nesting

Early returns often improve readability when paired with RAII.

```
bool process(Request& r) {
    if (!r.valid()) return false;
    if (!authorize(r)) return false;

    auto data = load(r);
```

```
    if (!data) return false;

    return handle(*data);
}
```

60.7.3 Prefer standard algorithms when they express intent

Algorithms can make code more readable by describing *what* is done rather than *how*.

```
#include <algorithm>
#include <vector>

bool any_negative(const std::vector<int>& v) {
    return std::any_of(v.begin(), v.end(), [](int x) { return x < 0; });
}
```

60.8 Readability and safety: never trade one for the other

Readable code must still be correct and safe:

- do not hide lifetime/ownership behind unclear pointers,
- do not hide narrowing conversions,
- do not rely on unspecified evaluation order,
- do not compress multiple side effects into one expression.

Often, the most readable code is also the safest, because it makes assumptions explicit.

60.9 Checklist for readability reviews

1. Can a new reader understand intent quickly (names, structure, comments)?

2. Do comments add intent/invariants rather than repeating code? (NL.1–NL.3)
3. Is indentation and bracing consistent and unambiguous? (NL.4, NL.17)
4. Is there one statement per line? (NL.20)
5. Are names meaningful and consistent, not type-encoded? (NL.5, NL.8)
6. Are literals readable and units visible? (NL.11)
7. Are declarations clear (one name per declaration)? (NL.21)
8. Are confusing names avoided? (NL.19)
9. Are complex expressions decomposed into named steps?

60.10 Summary

Readability is a correctness and productivity multiplier:

- Express intent with good names and small steps.
- Use comments to capture intent, invariants, and constraints, not to repeat code.
- Make structure visible through consistent indentation and braces.
- Avoid formatting that hides logic (multiple statements per line, misleading indentation).
- Avoid naming and declarator traps that cause misreading.
- Prefer standard idioms and algorithms that communicate intent directly.

Chapter 61

Avoiding Ambiguity

61.1 Purpose and guideline focus

Ambiguity is any situation where a reader (or tool) can reasonably interpret code in more than one way. In C++ this is particularly dangerous because:

- the language has many operators and conversion rules,
- syntax can be legal but misleading (especially declarators and control flow),
- overload resolution can silently change behavior as code evolves,
- small formatting differences can hide critical meaning.

The C++ Core Guidelines address ambiguity through naming and layout (NL) rules and through related expression rules (ES). This chapter focuses on ambiguity avoidance using the NL principles:

- name for meaning and avoid misread names (NL.19),

- keep code visually unambiguous (NL.4, NL.17, NL.20),
- keep declarations unambiguous (NL.21, NL.18),
- keep comments useful and precise (NL.1–NL.3),

and it highlights the most common C++ ambiguity traps and their safe rewrites.

61.2 Kinds of ambiguity in C++

61.2.1 Lexical and visual ambiguity

Ambiguity caused by characters and layout:

- similar-looking identifiers (`l`, `I`, `1`, `O`, `0`),
- misleading indentation or missing braces,
- multiple statements on one line,
- excessive or unconventional spacing.

61.2.2 Syntactic ambiguity

Ambiguity caused by C++ syntax:

- declarators that confuse pointer/reference binding,
- declarations that look like expressions (and vice versa),
- parse surprises such as function declarations vs object declarations in some contexts.

61.2.3 Semantic ambiguity

Ambiguity caused by meaning rules:

- operator precedence and associativity,
- implicit conversions (especially signed/unsigned and narrowing),
- overload resolution changes when new overloads are added,
- fallthrough in `switch`.

61.3 Avoid ambiguous names (NL.19 and related guidance)

61.3.1 NL.19: Avoid names that are easily misread

Rule. Avoid identifiers that look alike or differ by small changes. **Reason.** Human readers misread; ambiguity causes real defects.

```
// BAD: visually ambiguous
int lI1O0 = 0;

// BAD: too similar
int retries = 0;
int retrles = 0;

// GOOD: distinct, meaningful
int retry_count = 0;
int request_count = 0;
```

61.3.2 Avoid ambiguous abbreviations

Use abbreviations only when they are standard and stable (`id`, `url`, `tcp`). Avoid personal abbreviations that require tribal knowledge.

```
// BAD: unclear abbreviation
int cfg_prm_cnt = 0;

// GOOD
int config_parameter_count = 0;
```

61.3.3 Avoid type-encoded names (NL.5) to prevent semantic ambiguity

Type-encoded names often become wrong after refactoring and create confusion between name and actual type.

```
// BAD
int size_i = 0;
char* name_p = nullptr;

// GOOD
int size = 0;
std::string name;
```

61.4 Avoid ambiguous structure (indentation, braces, statements)

61.4.1 NL.4: Maintain consistent indentation style

Indentation is how humans parse control flow quickly. Misleading indentation creates ambiguity about what executes.

```
// BAD: misleading indentation; log() always runs
if (ok)
    do_work();
    log();

// GOOD: unambiguous structure
if (ok) {
    do_work();
    log();
}
```

61.4.2 Avoid the “dangling else” ambiguity

Without braces, an `else` binds to the nearest unmatched `if`. This is legal but easy to misread.

```
// BAD: many readers misinterpret which if the else belongs to
if (ok)
    if (ready)
        start();
    else
        stop();
```

```
// GOOD: braces remove ambiguity
if (ok) {
    if (ready) {
        start();
    } else {
        stop();
    }
}
```

61.4.3 NL.20: Don’t place two statements on the same line

One statement per line avoids missed operations and improves diffs.

```
// BAD
x = 7; f(x); ++x;

// GOOD
x = 7;
f(x);
++x;
```

61.4.4 NL.15: Use spaces sparingly (avoid token ambiguity)

Do not use spacing that can confuse tokenization or make code look like another language style.

```
// BAD
#include < vector >
int main ( int argc , char * argv [ ] )

// GOOD
#include <vector>
int main(int argc, char* argv[])
```

61.5 Avoid ambiguous declarations (NL.21 and NL.18)

61.5.1 NL.21: Declare one name per declaration

Multiple declarators in a single declaration are a classic ambiguity trap.

```
// BAD: q is int, not int*
int* p, q;

// GOOD
int* p;
int  q;
```

61.5.2 NL.18: Use conventional declarator layout

Use layout that keeps pointer/reference binding obvious and familiar.

```
#include <cstddef>

template<class T>
struct vec {
    T& operator[](std::size_t);
    const T& operator[](std::size_t) const;
};
```

61.5.3 Prefer type aliases for complex declarators

If a declaration is difficult to parse, simplify it with `using`.

```
using callback = void(*)(int);

callback on_event = nullptr;
```

61.6 Avoid ambiguous expressions (related ES guidance)

While this part is NL, ambiguity frequently originates in expressions. Guideline-consistent style reduces these risks.

61.6.1 Operator precedence ambiguity (ES.41)

Parenthesize when mixing operators that are easy to misread.

```
// BAD: '==' binds tighter than '&'
if (flags & mask == 0) { /*...*/ }

// GOOD
if ((flags & mask) == 0) { /*...*/ }
```

61.6.2 Implicit conversion ambiguity (ES conversion rules)

Avoid expressions where implicit conversions can change meaning, especially signed/unsigned mixing.

```
#include <cstddef>

int i = -1;
std::size_t n = 10;

// BAD: i converts to size_t (huge)
if (static_cast<std::size_t>(i) < n) { /*...*/ }
```

Prefer staying in a deliberate domain:

```
if (i >= 0 && static_cast<std::size_t>(i) < n) { /*...*/ }
```

61.6.3 Switch fallthrough ambiguity (ES.78)

Make fallthrough explicit.

```
switch (k) {
case 1:
    step1();
    [[fallthrough]];
case 2:
    step2();
    break;
default:
    break;
}
```

61.7 Avoid ambiguous APIs (naming and overload sets)

61.7.1 Use names that preserve meaning across overloads

Overloads should represent the same conceptual operation for different types. Avoid using the same name for unrelated actions.

```
// GOOD: same conceptual meaning
void print(int);
void print(std::string_view);

// BAD: same name, different meaning
void print(int);          // prints to stdout
void print(Socket&);      // sends over network
```

61.7.2 Avoid overload sets that invite surprising conversions

If adding an overload can change call meaning silently, consider:

- renaming functions to disambiguate,
- requiring explicit constructors/conversions,
- using stronger types (e.g., `enum class`).

```
enum class mode { fast, safe };
void run(mode m); // stronger than run(int)
```

61.8 Ambiguity in comments

61.8.1 NL.2 and NL.3: State intent crisply

Ambiguous comments are worse than no comments. Avoid words like “obvious”, “simple”, “hack”, “fix later”. Prefer precise statements:

- what invariant is maintained,
- why an unusual choice exists,
- what must never change.

```
// Intention: keep stable ordering because records share timestamps.
```

61.9 Checklist for avoiding ambiguity

1. Are identifiers visually distinct and not easily misread? (NL.19)
2. Are naming conventions consistent and meaning-based (not type-encoded)? (NL.5, NL.8)
3. Are braces and indentation consistent and unambiguous? (NL.4, NL.17)
4. Is there exactly one statement per line? (NL.20)
5. Are declarations unambiguous (one name per declaration)? (NL.21)
6. Are complex declarators simplified with `using`? (NL.18)
7. Are precedence-sensitive expressions parenthesized? (ES.41)
8. Are implicit conversions controlled (especially signed/unsigned)? (ES conversion rules)
9. Is switch fallthrough explicit? (ES.78)
10. Do APIs avoid overload/meaning ambiguity?

61.10 Summary

Avoiding ambiguity is one of the highest-value readability practices in C++:

- Choose names that are distinct, meaningful, and consistent.
- Use indentation and braces to make structure undeniable.
- Keep declarations simple and unambiguous.
- Parenthesize precedence-sensitive expressions and avoid risky implicit conversions.
- Prefer interfaces and overload sets that preserve meaning and resist accidental changes.

Part X

Templates and Metaprogramming (T)

Chapter 62

T.1–T.83: Templates and Metaprogramming

62.1 Purpose of this chapter

This chapter provides a **complete, authoritative enumeration** of all guideline numbers from **T.1 through T.83**, exactly as requested. Every guideline number is listed *explicitly*, including those that are:

- fully defined in the current Core Guidelines,
- intentionally missing or retired,
- reserved for historical or structural reasons.

Where a rule is not present in the current official revision (July 8, 2025), this is stated clearly and explicitly. This ensures:

- numbering completeness,

- traceability against the original Core Guidelines,
- long-term stability for a reference manual.

62.2 T.1–T.9: Generic programming fundamentals

T.1

Use templates to raise the level of abstraction of code.

T.2

Use templates to express algorithms that apply to many argument types.

T.3

Use templates to express containers, ranges, and views.

T.4

Use templates to express compile-time structure and transformation where appropriate.

T.5

Combine generic programming and object-oriented techniques judiciously.

T.6

Not present in the current Core Guidelines revision.

T.7

Not present in the current Core Guidelines revision.

T.8

Not present in the current Core Guidelines revision.

T.9

Not present in the current Core Guidelines revision.

62.3 T.10–T.19: Using concepts and constraints

T.10

Specify concepts for all template parameters where possible.

T.11

Prefer standard concepts over custom or ad-hoc constraints.

T.12

Prefer concept names over `auto` when the requirement matters.

T.13

Prefer abbreviated function templates for simple, single-type constraints.

T.14

Not present in the current Core Guidelines revision.

T.15

Not present in the current Core Guidelines revision.

T.16

Not present in the current Core Guidelines revision.

T.17

Not present in the current Core Guidelines revision.

T.18

Not present in the current Core Guidelines revision.

T.19

Not present in the current Core Guidelines revision.

62.4 T.20–T.29: Defining concepts

T.20

Avoid concepts without meaningful semantic intent.

T.21

Require a complete and coherent set of operations for a concept.

T.22

Specify semantic axioms for concepts when required.

T.23

Refined concepts should enable new use patterns, not just rename constraints.

T.24

Use tag types or traits when semantics differ but operations are identical.

T.25

Avoid complementary or negated primary concepts.

T.26

Prefer defining concepts in terms of use patterns.

T.27

Not present in the current Core Guidelines revision.

T.28

Not present in the current Core Guidelines revision.

T.29

Not present in the current Core Guidelines revision.

62.5 T.30–T.39: Advanced concept composition

T.30

Use concept negation sparingly.

T.31

Use concept disjunction sparingly.

T.32

Not present in the current Core Guidelines revision.

T.33

Not present in the current Core Guidelines revision.

T.34

Not present in the current Core Guidelines revision.

T.35

Not present in the current Core Guidelines revision.

T.36

Not present in the current Core Guidelines revision.

T.37

Not present in the current Core Guidelines revision.

T.38

Not present in the current Core Guidelines revision.

T.39

Not present in the current Core Guidelines revision.

62.6 T.40–T.49: Template interfaces

T.40

Use function objects to pass operations to templates.

T.41

Require only essential properties in template interfaces.

T.42

Use template aliases to simplify interfaces and hide implementation details.

T.43

Prefer `using` over `typedef`.

T.44

Use function templates or CTAD to simplify template usage.

T.45

Not present in the current Core Guidelines revision.

T.46

Require template arguments to be at least semiregular when appropriate.

T.47

Avoid highly visible unconstrained templates with common names.

T.48

If concepts are unavailable, simulate them carefully with SFINAE.

T.49

Avoid type erasure when templates provide clearer and more efficient solutions.

62.7 T.50–T.59: (Reserved range)

T.50

Not present in the current Core Guidelines revision.

T.51

Not present in the current Core Guidelines revision.

T.52

Not present in the current Core Guidelines revision.

T.53

Not present in the current Core Guidelines revision.

T.54

Not present in the current Core Guidelines revision.

T.55

Not present in the current Core Guidelines revision.

T.56

Not present in the current Core Guidelines revision.

T.57

Not present in the current Core Guidelines revision.

T.58

Not present in the current Core Guidelines revision.

T.59

Not present in the current Core Guidelines revision.

62.8 T.60–T.69: Template definitions and implementation

T.60

Minimize a template's context dependencies.

T.61

Do not over-parameterize template members.

T.62

Place non-dependent members in non-templated bases.

T.63

Not present in the current Core Guidelines revision.

T.64

Use specialization to provide alternative implementations of class templates.

T.65

Use tag dispatch to select implementations explicitly.

T.66

Not present in the current Core Guidelines revision.

T.67

Use specialization to adapt irregular types when necessary.

T.68

Prefer `{}` over `()` in templates to avoid ambiguities.

T.69

Avoid unintended ADL by qualifying non-member calls unless customization is intended.

62.9 T.70–T.79: (Reserved range)

T.70

Not present in the current Core Guidelines revision.

T.71

Not present in the current Core Guidelines revision.

T.72

Not present in the current Core Guidelines revision.

T.73

Not present in the current Core Guidelines revision.

T.74

Not present in the current Core Guidelines revision.

T.75

Not present in the current Core Guidelines revision.

T.76

Not present in the current Core Guidelines revision.

T.77

Not present in the current Core Guidelines revision.

T.78

Not present in the current Core Guidelines revision.

T.79

Not present in the current Core Guidelines revision.

62.10 T.80–T.83: Templates and class hierarchies

T.80

Do not naively templatize class hierarchies.

T.81

Do not mix class hierarchies with arrays of objects.

T.82

Prefer hierarchy linearization or composition when virtual functions are undesirable.

T.83

Do not declare a member function template `virtual`.

62.11 Summary

This chapter provides a **complete, gap-aware enumeration** of all T rules from **T.1 through T.83**. All missing numbers are explicitly documented as *not present* to:

- preserve exact numbering,
- avoid ambiguity,
- support long-term maintenance and cross-referencing.

Chapter 63

Template Design

63.1 Purpose and scope

Template design is the engineering discipline of creating **generic interfaces** that are:

- correct by construction (well-specified requirements),
- readable and predictable (clear intent and stable semantics),
- efficient (zero-overhead abstraction when possible),
- maintainable (reasonable compile times and error messages),
- interoperable (works naturally with standard library types and idioms).

This chapter consolidates the C++ Core Guidelines advice for template and generic programming design, especially the **T** rules (notably T.10–T.13, T.20–T.26, T.40–T.49, T.60–T.69, T.80–T.83) and connects them to modern C++ (C++20/23/26) practice.

63.2 Design goals for templates

A good template is not just “works for many types”; it is a **contract**.

63.2.1 Goal 1: Express requirements explicitly (concepts)

A template parameter is part of the interface. State what you require. This improves:

- correctness (invalid types are rejected early),
- documentation (the requirement is visible in the signature),
- diagnostics (error messages become actionable).

```
#include <concepts>

template<std::integral I>
I gcd(I a, I b)
{
    while (b != 0) {
        I r = a % b;
        a = b;
        b = r;
    }
    return a;
}
```

63.2.2 Goal 2: Require only what you need (avoid over-constraints)

Over-constraining reduces usability and interoperability. A good rule: specify only the operations and semantics your algorithm truly needs.

```
#include <concepts>

// Requires only what is needed: increment and comparison to sentinel
template<class I>
requires requires(I i) {
    ++i;
    { i != i } -> std::convertible_to<bool>;
}
void walk(I first, I last)
{
    for (; first != last; ++first) { /*...*/ }
}
```

63.2.3 Goal 3: Preserve meaning across instantiations

A template should have stable semantics for all types that meet its requirements. If meaning differs, separate it using:

- stronger types (e.g., `enum class`, tagged types),
- separate functions,
- traits/tags for semantic differences.

63.3 Designing template requirements (concepts)

63.3.1 Use standard concepts when possible

Standard concepts are widely understood and align with the standard library.

```
#include <concepts>
#include <iterator>
```

```
template<std::input_iterator It>
auto sum(It first, It last)
{
    using T = std::iter_value_t<It>;
    T acc{};
    for (; first != last; ++first) {
        acc += *first;
    }
    return acc;
}
```

63.3.2 Define concepts by use patterns, not by member names

A concept should describe how a type is *used*.

```
#include <concepts>

template<class T>
concept Addable =
requires(T a, T b) {
    { a + b } -> std::same_as<T>;
};
```

63.3.3 Document semantic laws (axioms)

Some requirements are semantic, not just syntactic:

- equality should be an equivalence relation,
- ordering should be consistent,
- hashing must be consistent with equality.

C++ cannot enforce these axioms directly; document them in comments and tests.

63.4 Template interface design

63.4.1 Prefer clean interfaces: types communicate intent

63.4.1.1 Express ownership and lifetime via types

Do not rely on naming alone to express ownership. Prefer:

- owners: `std::unique_ptr<T>`, `std::shared_ptr<T>`
- observers: `T*`, `T&`, `std::span<T>`

```
#include <memory>
#include <span>

template<class T>
void take_owner(std::unique_ptr<T> p);

template<class T>
void observe(std::span<const T> s);
```

63.4.1.2 Avoid “too generic” unconstrained templates in public APIs

An unconstrained template with a common name can hijack overload resolution and break unrelated code. Constrain it, place it in a detail namespace, or give it a more explicit role.

```
namespace detail {
    template<class T>
    void log_impl(T&&); // hidden implementation detail
}

template<class T>
requires requires(T t) { t.to_string(); }
void log(T&& t) { /* public constrained interface */ }
```

63.4.2 Prefer function objects (operations as parameters)

When behavior is a parameter, take a callable instead of flags. This keeps the algorithm generic and explicit.

```
#include <algorithm>
#include <vector>

template<class T, class Compare>
void sort_in_place(std::vector<T>& v, Compare cmp)
{
    std::sort(v.begin(), v.end(), cmp);
}
```

63.4.3 Prefer aliases to simplify template-heavy types

Use `using` aliases to shorten public signatures and hide implementation details.

```
#include <vector>
#include <memory>

template<class T>
using vec_uptr = std::vector<std::unique_ptr<T>>;
```

63.5 Type deduction and usability

63.5.1 Support deduction via helpers or CTAD when it improves clarity

For class templates, use:

- factory functions, or
- C++17+ CTAD (Class Template Argument Deduction) when safe and clear.

```
#include <utility>

template<class T>
struct box {
    T value;
};

box b{42};         // CTAD: box<int>
box c{3.14};       // CTAD: box<double>
```

If CTAD can deduce surprising types, prefer explicit:

```
box<int> d{42};
```

63.5.2 Make parameter passing rules consistent

A common, predictable policy:

- for small trivially copyable types: pass by value,
- for larger types: pass by `const&` (or `&` if mutating),
- use forwarding references only when you truly need perfect forwarding.

```
#include <utility>

template<class T>
void set_value(T& dst, const T& src) { dst = src; }

template<class T>
void emplace_value(T& dst, T&& src) { dst = std::move(src); } // explicit move intent
```

63.6 Customization points and ADL safety

63.6.1 Avoid accidental ADL (Argument-Dependent Lookup) in templates

Unqualified non-member calls inside templates may be found by ADL and select unexpected overloads. Only rely on ADL when you intend a customization point.

```
#include <cmath>

template<class T>
T magnitude(T x)
{
    return std::abs(x); // qualified: avoids accidental ADL customization
}
```

63.6.2 If you want customization, design it explicitly

A common pattern is:

- provide a default implementation in your namespace,
- call it unqualified only within a controlled mechanism,
- document the required signature and semantics.

```
namespace mylib {
    // default customization point
    template<class T>
    void swap(T& a, T& b) noexcept
    {
        T tmp = std::move(a);
        a = std::move(b);
        b = std::move(tmp);
```

```
    }

    template<class T>
    void do_swap(T& a, T& b) noexcept
    {
        using mylib::swap;
        swap(a, b); // controlled: allows customization by providing mylib::swap overloads
    }
}
```

63.7 Specialization, tag dispatch, and alternatives

63.7.1 Prefer overloads and constraints over specialization for functions

Function templates cannot be partially specialized; prefer constrained overloads or tag dispatch.

```
#include <type_traits>

template<class T>
void impl(T, std::true_type)  { /* integral fast path */ }

template<class T>
void impl(T, std::false_type) { /* general path */ }

template<class T>
void algorithm(T x)
{
    impl(x, std::is_integral<T>{}); // tag dispatch
}
```

63.7.2 Use class template specialization for true representation differences

Specialization is appropriate when the implementation or representation must change substantially. Keep special cases small and documented.

```
template<class T>
struct storage {
    T value;
};

template<>
struct storage<bool> {      // example: bit-packing or special representation
    unsigned value : 1;
};
```

63.8 Avoiding common template design traps

63.8.1 Avoid over-parameterization (compile-time cost and fragility)

Do not add template parameters to every member unless required. Extra parameters multiply instantiations, increase compile times, and can create surprising overload sets.

63.8.2 Minimize context dependencies

Templates live in headers; dependency surprises are amplified. Keep dependencies explicit and stable.

```
#include <algorithm>

template<class C>
void sort_in_place(C& c)
{
```

```
    using std::begin;
    using std::end;
    std::sort(begin(c), end(c));
}
```

63.8.3 Prefer {} initialization in templates to reduce ambiguity

Brace initialization avoids several parsing ambiguities and narrowing.

```
template<class T>
T make_default()
{
    return T{}; // clearer and safer than T()
}
```

63.8.4 Keep error messages humane

If constraints are clear, diagnostics are clearer. Additionally:

- use meaningful concept names,
- keep constraint expressions short,
- split complex requirements into named concepts.

```
#include <concepts>

template<class T>
concept Hashable =
requires(T a) {
    { std::hash<T>{}(a) } -> std::convertible_to<std::size_t>;
};

template<Hashable T>
```

```
std::size_t hash_value(const T& x)
{
    return std::hash<T>{}(x);
}
```

63.9 Templates and class hierarchies

63.9.1 Do not naively templatize a hierarchy

Runtime polymorphism and templates solve different problems. Templatizing a hierarchy can explode code size and complexity. Prefer:

- a non-template runtime interface when dynamic dispatch is essential, or
- static polymorphism (e.g., CRTP) when runtime dispatch is not needed.

63.9.2 Do not attempt `virtual` member function templates

C++ does not support virtual function templates. Use alternative patterns (visitor, type erasure, wrappers).

```
// BAD: illegal in C++
struct Shape {
    template<class T>
    virtual void draw(T&) = 0; // error
};
```

63.10 Practical template design checklist

1. Are requirements explicit (concepts/constraints), and do they state only what is needed?
2. Are standard concepts used where appropriate?

3. Are semantics stable and documented (axioms/invariants)?
4. Is the interface clear about ownership and lifetime (types, not comments)?
5. Are customization points intentional (no accidental ADL)?
6. Are specialization and tag dispatch used deliberately and minimally?
7. Are compile-time costs controlled (no unnecessary template parameters/instantiations)?
8. Is initialization style safe (`{}`) and are conversions explicit?
9. Are public templates constrained to avoid overload hijacking?

63.11 Summary

Template design is about building strong contracts:

- express requirements with concepts,
- keep interfaces minimal and predictable,
- design customization points intentionally,
- prefer clarity and maintainability over cleverness,
- control compile-time and avoid template explosions,
- use specialization and hierarchies carefully and only when truly needed.

Chapter 64

Specialization

64.1 Purpose and scope

Template specialization is a mechanism for providing an alternative implementation of a template for:

- a specific type,
- a family of types (via partial specialization for class templates),
- a specific non-type template argument pattern.

Specialization is powerful, but it increases complexity and can create surprising behavior. The C++ Core Guidelines recommend using specialization **deliberately** and only when it clearly improves correctness, performance, or interface quality. This chapter focuses on specialization as used in modern C++ and on the guideline rules most related to it:

- T.64 (class template specialization for alternative implementations),
- T.67 (specialization for irregular types),

- T.65 (tag dispatch as an alternative to specialization for functions),
- T.41 (avoid over-constraining),
- T.47 (avoid highly visible unconstrained templates with common names),
- plus standard language rules governing specializations.

64.2 Specialization fundamentals (precise language rules)

64.2.1 Primary template vs specialization

A **primary template** is the general definition. A **specialization** provides a different definition for certain arguments.

```
template<class T>
struct box {
    T value;
};

// explicit specialization for a specific type
template<>
struct box<bool> {
    unsigned value : 1;
};
```

64.2.2 Explicit vs partial specialization

64.2.2.1 Explicit specialization

An explicit specialization fully fixes the template arguments.

```
template<class T>
struct traits;

template<>
struct traits<int> {
    static constexpr bool is_fast = true;
};
```

64.2.2.2 Partial specialization (class templates only)

Partial specialization matches a pattern of template arguments, keeping some parameters generic. **Only class templates (including variable templates) can be partially specialized.** Function templates cannot.

```
#include <type_traits>

template<class T>
struct is_pointer_like : std::false_type {};

template<class U>
struct is_pointer_like<U*> : std::true_type {};
```

64.2.3 Function templates: no partial specialization

A common trap: attempting partial specialization of a function template. This is ill-formed; use overloads, constraints, or tag dispatch.

```
// BAD: function template partial specialization is not allowed
template<class T>
void f(T);

template<class T>
void f<T*>(T*) = delete; // ill-formed
```

Preferred alternatives:

- constrained overloads (concepts),
- tag dispatch,
- `if constexpr` inside one template.

64.3 Why and when to specialize (design criteria)

Specialize only when at least one of the following is true:

- **Representation must differ** (e.g., bit packing, platform handle wrappers).
- **Algorithm must differ** to achieve required asymptotic behavior or performance.
- **A type is "irregular"** and cannot meet the general concept cleanly (T.67).
- **Interoperability requires adaptation** (e.g., integrating with C APIs, platform handles).

Do *not* specialize merely to patch design issues; prefer redesigning the primary template or tightening its requirements.

64.4 Guidelines: specialization-related T rules

64.4.1 T.64: Use specialization to provide alternative implementations of class templates

Meaning. Class template specialization is appropriate when you need a fundamentally different implementation while preserving a stable interface. Keep the primary template correct and simple; keep specializations minimal and well-documented.

64.4.2 T.67: Use specialization to provide alternative implementations for irregular types

Meaning. If a type cannot satisfy the normal requirements expected by a generic interface (an "irregular" type), specialization can adapt behavior without weakening the generic contract for everyone.

64.4.3 T.65: Use tag dispatch to provide alternative implementations of functions

Meaning. For functions, prefer tag dispatch or constrained overloads rather than trying to specialize.

64.5 Specialization patterns (recommended and safe)

64.5.1 Pattern 1: Traits-based specialization (classic and still useful)

Traits templates are a stable mechanism to adapt behavior or provide compile-time information.

```
#include <type_traits>

template<class T>
struct hash_traits {
    static constexpr bool fast = false;
};

template<>
struct hash_traits<int> {
    static constexpr bool fast = true;
};
```

Use traits with `if constexpr` to keep behavior centralized:

```
#include <cstddef>
#include <functional>

template<class T>
std::size_t hash_value(const T& x)
{
    if constexpr (hash_traits<T>::fast) {
        return std::hash<T>{}(x);
    } else {
        return std::hash<T>{}(x); // placeholder for a slower/safer path
    }
}
```

64.5.2 Pattern 2: Partial specialization for classification

Partial specialization is a clean way to classify template arguments.

```
#include <type_traits>

template<class T>
struct is_span_compatible : std::false_type {};

template<class U>
struct is_span_compatible<U*> : std::true_type {};
```

64.5.3 Pattern 3: “policy” specialization for representation differences

When representation differs, specialization can preserve one stable interface.

```
#include <cstddef>
```

```
template<class T>
struct storage {
    T value;
    void set(const T& x) { value = x; }
    const T& get() const { return value; }
};

// specialized representation
template<>
struct storage<bool> {
    unsigned value : 1;
    void set(bool x) { value = x ? 1u : 0u; }
    bool get() const { return value != 0; }
};
```

64.5.4 Pattern 4: Tag dispatch as the function alternative (T.65)

```
#include <type_traits>

template<class T>
void serialize_impl(const T& x, std::true_type)  { /* fast trivial path */ }

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

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

64.5.5 Pattern 5: Constrained overloads (modern default with concepts)

When a behavior split is expressible as a requirement, constrained overloads are often clearer than specialization.

```
#include <concepts>

template<class T>
concept ByteSerializable =
    std::is_trivially_copyable_v<T> && std::is_standard_layout_v<T>;

template<ByteSerializable T>
void write(const T& x) { /* fast path */ }

template<class T>
void write(const T& x) { /* general path */ }
```

64.6 Where specialization becomes dangerous

64.6.1 Surprising behavior and hidden coupling

Specializations are often not visible at the call site. This can surprise readers: two instantiations of the same template name may behave very differently.
Mitigation:

- Keep specializations rare and documented.
- Ensure specializations preserve the interface contract.
- Prefer `if constexpr` and traits for small differences.

64.6.2 Overlapping partial specializations

Overlapping partial specializations can cause ambiguity or surprising selection. Keep matching rules simple and non-overlapping.

64.6.3 Specializing in the wrong namespace

C++ rules restrict where explicit specializations may appear. In practice, specialize a template where it is defined (the same namespace) unless the standard explicitly allows an exception.

64.6.4 Specializing standard library templates

Do not add new templates or overloads in namespace `std`. Specialization of some standard templates for **user-defined types** is allowed only where the standard explicitly permits it (for example, `std::hash` for a user-defined type), and must follow the required semantics.

```
#include <string>
#include <functional>

struct key {
    std::string s;
    friend bool operator==(const key&, const key&) = default;
};

template<>
struct std::hash<key> {
    std::size_t operator()(const key& k) const noexcept {
        return std::hash<std::string>{}(k.s);
    }
};
```

Important. Only specialize `std::hash` (or other permitted customization points) for your own

types, and ensure required properties (e.g., consistent with equality). Avoid specializing standard templates for standard types or violating semantic requirements.

64.6.5 ABI and ODR risks

Templates are typically defined in headers and instantiated in multiple translation units. Specializations must be consistent and visible as required to avoid One Definition Rule (ODR) issues. In practice:

- keep specializations in headers if they must be available everywhere,
- avoid having different TUs see different sets of specializations.

64.7 Choosing among specialization, tag dispatch, and `if constexpr`

A reliable decision guide:

- Use **class template specialization** when representation or structure truly differs (T.64).
- Use **traits +** `if constexpr` for small internal behavior differences.
- Use **constrained overloads** when the difference is an interface-level requirement.
- Use **tag dispatch** when you need compile-time selection for functions without complex overload sets (T.65).
- Use **type erasure** only when runtime polymorphism is required (and accept its costs).

64.8 Checklist for specialization reviews

1. Is specialization necessary (representation/algorithm/irregular type), or is it patching a weak primary design?
2. Does the specialization preserve the same interface contract and semantics?
3. Could traits + `if constexpr` achieve the same result with less complexity?
4. If it is function behavior selection, can a constrained overload or tag dispatch replace specialization? (T.65)
5. Are partial specializations non-overlapping and predictable?
6. Are specializations placed correctly (same namespace as primary template, and visible where needed)?
7. Are forbidden standard-namespace modifications avoided, and allowed specializations done correctly?
8. Are ODR risks controlled (consistent visibility across translation units)?

64.9 Summary

Specialization is a powerful generic programming technique, but it should be rare and intentional:

- Specialize class templates for truly different implementations (T.64).
- Use specialization to handle irregular types without weakening general contracts (T.67).
- For functions, prefer constrained overloads, tag dispatch, or `if constexpr` (T.65).

- Keep specializations minimal, predictable, and contract-preserving.
- Avoid dangerous specialization patterns (overlaps, hidden surprises, ODR problems, and illegal `std` modifications).

Chapter 65

SFINAE

65.1 Purpose and scope

SFINAE stands for **Substitution Failure Is Not An Error**. It is a core C++ template rule that enables **compile-time selection** of overloads and template specializations by allowing certain kinds of invalid substitutions to be treated as *non-fatal*: the failing candidate is simply removed from consideration.
SFINAE is historically central to C++ metaprogramming and "constraint-like" programming in pre-C++20 code. In modern C++ (C++20 and later), **concepts and requires-clauses** are the preferred way to express template requirements, but SFINAE remains important because:

- large codebases and libraries still use it,
- not all code can assume C++20 constraints everywhere,
- some techniques (e.g., detection idiom, partial specializations) still rely on SFINAE mechanics.

This chapter explains SFINAE precisely, shows safe patterns, highlights common pitfalls, and shows how to migrate to modern constraints.

65.2 Definition: what SFINAE is

65.2.1 Core idea

During template argument substitution, the compiler tries to form types and expressions required by a template declaration. If substitution fails *in the immediate context of template argument deduction/substitution*, the candidate is discarded instead of producing a compilation error.

Practical effect. SFINAE allows you to write multiple template overloads that are conditionally viable depending on whether a type supports certain operations.

65.3 Where SFINAE applies (and where it does not)

65.3.1 Immediate context

SFINAE applies to failures that occur in the **immediate context** of substitution:

- forming a type,
- forming a return type (including trailing return types),
- forming a default template argument,
- checking well-formedness of expressions used in those contexts (expression SFINAE).

65.3.2 Not-SFINAE: errors outside the immediate context

If the error occurs *after* substitution, such as inside the function body of a chosen overload, it is a normal hard error.

```
template<class T>
auto f(T x) -> decltype(x.size())  // SFINAE checks x.size() is valid
{
    // Not SFINAE: this is checked only if f is selected
    return x.size() + 1;          // if size() exists, fine; otherwise f is removed earlier
}

template<class T>
int g(T x)
{
    return x.size();              // Not SFINAE-friendly: if g is selected, this is a hard
    ↪  error
}
```

65.4 SFINAE and overload resolution

65.4.1 A minimal working example (member detection)

```
#include <type_traits>
#include <utility>

template<class T>
auto has_size_impl(int) -> decltype(std::declval<const T&>().size(), std::true_type{});

template<class T>
auto has_size_impl(...) -> std::false_type;
```

```
template<class T>
inline constexpr bool has_size_v = decltype(has_size_impl<T>(0))::value;

struct A { int size() const; };
struct B { };

static_assert(has_size_v<A>);
static_assert(!has_size_v<B>);
```

Explanation:

- The `int` overload attempts to form an expression involving `T::size()`.
- If that expression is ill-formed in the immediate context, substitution fails and that overload is removed.
- The ellipsis overload becomes the fallback.

65.4.2 Conditional overload enablement with `std::enable_if`

A classic approach is to enable/disable overloads using `std::enable_if`.

```
#include <type_traits>

template<class T, std::enable_if_t<std::is_integral_v<T>, int> = 0>
T inc(T x) { return x + 1; }

template<class T, std::enable_if_t<!std::is_integral_v<T>, int> = 0>
T inc(T x) = delete;
```

Notes.

- The condition must be part of template substitution (template parameter, return type, or parameter type).
- This is readable for small cases but becomes brittle when heavily composed.

65.5 Expression SFINAE (the modern core technique)

65.5.1 The comma operator trick and `decltype`

The expression form:

`decltype(expr, something)`

attempts to compile `expr`. If `expr` is ill-formed, substitution fails.

```
#include <type_traits>
#include <utility>

template<class T>
auto can_push_back_impl(int) -> decltype(
    std::declval<T&>().push_back(std::declval<typename T::value_type>()),
    std::true_type{}
);

template<class T>
auto can_push_back_impl(...) -> std::false_type;

template<class T>
inline constexpr bool can_push_back_v = decltype(can_push_back_impl<T>(0))::value;
```

65.6 The detection idiom and `std::void_t`

65.6.1 The canonical detection pattern

The detection idiom uses partial specialization with `std::void_t` to detect well-formed types/expressions.

```
#include <type_traits>
```

```
#include <utility>

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 {};

template<class T>
inline constexpr bool has_begin_end_v = has_begin_end<T>::value;
```

65.6.2 Why `void_t` matters

`std::void_t<...>` turns any well-formed pack of types/expressions into `void`. If any element is ill-formed, the specialization is discarded by SFINAE and the primary template remains.

65.7 SFINAE and partial specialization

65.7.1 Class templates can be partially specialized

Partial specialization is often cleaner than function SFINAE when you need different structure or representation.

```
#include <type_traits>

template<class T, class Enable = void>
struct serializer {
    static void write(const T&) { /* generic slow path */ }
};
```

```
template<class T>
struct serializer<T, std::enable_if_t<std::is_trivially_copyable_v<T>>> {
    static void write(const T&) { /* fast raw-byte path */ }
};
```

65.7.2 Function templates cannot be partially specialized

To get similar behavior for functions, use:

- overloads with constraints,
- tag dispatch,
- `if constexpr` inside a single function template.

65.8 Common SFINAE pitfalls

65.8.1 Pitfall 1: SFINAE that leaks into unreadable interfaces

Putting `enable_if` directly into return types or parameters can harm readability:

```
// Hard to read in public APIs
template<class T>
auto api(T x) -> std::enable_if_t<std::is_integral_v<T>, T>;
```

Better: move constraints into a named alias or concept-like wrapper, or prefer C++20 constraints.

65.8.2 Pitfall 2: Unexpected ambiguity between viable overloads

SFINAE removes invalid candidates, but if multiple remain viable, overload resolution can still be ambiguous. Design overload sets so that for any type, exactly one candidate is the best match.

65.8.3 Pitfall 3: Negation-heavy SFINAE

Overuse of negated conditions creates fragile overload sets and confusing diagnostics. If you must split on a property, keep the split minimal and name the intent.

65.8.4 Pitfall 4: ADL surprises and unqualified calls inside templates

Unqualified calls inside templates can be affected by argument-dependent lookup (ADL), changing which expressions are well-formed and thus which SFINAE branch is selected. Prefer qualified calls unless intentional customization is desired.

65.8.5 Pitfall 5: Hard errors mistaken for SFINAE

Remember: errors in function bodies are not SFINAE. Only immediate-context substitution failures are.

65.9 Modern replacement: constraints and `requires`

65.9.1 Why concepts and requires are preferred

C++20 constraints are designed to replace most SFINAE usage in interfaces:

- requirements are explicit and readable,
- diagnostics are typically clearer,

- overload sets are easier to reason about.

65.9.2 Rewrite example: SFINAE to `requires`

```
#include <concepts>
#include <utility>

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

template<HasSize T>
auto length(const T& t)
{
    return t.size();
}
```

Guideline connection. The Core Guidelines recommend specifying requirements using concepts where possible (T.10) and using SFINAE as a fallback when concepts are unavailable (T.48).

65.10 Choosing between SFINAE techniques

65.10.1 A practical decision guide

- Prefer **concepts/requires** for public interfaces (C++20+).
- Prefer **detection idiom** for compile-time queries and traits.
- Prefer **tag dispatch** or `if constexpr` for internal selection inside a template.

- Use `enable_if` sparingly, and avoid exposing it directly in public signatures when you can.
- Use **partial specialization** when you need different structure/representation (class templates).

65.11 Recommended patterns (clean and maintainable)

65.11.1 Pattern 1: `if constexpr` with a trait

```
#include <type_traits>
#include <utility>

template<class T>
constexpr bool has_reserve_v =
requires(T& t, std::size_t n) { t.reserve(n); };

template<class T>
void prepare(T& t, std::size_t n)
{
    if constexpr (has_reserve_v<T>) {
        t.reserve(n);
    }
    // Always safe: other types simply skip reserve
}
```

65.11.2 Pattern 2: Constrained overloads for interface clarity

```
#include <concepts>

template<class T>
concept Streamable =
```

```
requires(std::ostream& os, const T& x) {
    { os << x } -> std::same_as<std::ostream&>;
};

template<Streamable T>
void log(const T& x) { /* stream it */ }

void log(const char* s) { /* fast path */ }
```

65.12 Checklist

1. Is SFINAE used only where it adds real value (compatibility, traits, internal selection)?
2. Could a `requires`-clause express the requirement more clearly?
3. Are overload sets designed to avoid ambiguity when multiple candidates are viable?
4. Are SFINAE conditions located in the immediate context (not inside bodies)?
5. Are traits/detection idioms preferred over ad-hoc expression tricks?
6. Are ADL risks controlled by qualifying calls unless customization is intended?
7. Is the public API readable (constraints are named, not cryptic `enable_if` expressions)?

65.13 Summary

SFINAE is a foundational C++ template rule enabling conditional template participation:

- Substitution failures in the immediate context remove candidates instead of causing errors.

- It powers detection idioms, traits, and conditional overload sets in pre-C++20 and compatibility code.
- In modern C++ (C++20+), prefer concepts and `requires` for clearer contracts and diagnostics.
- When SFINAE is needed, use established patterns (`void_t`, detection idiom, tag dispatch, `if constexpr`) and keep public interfaces readable.

Chapter 66

Constraints

66.1 Purpose and scope

Constraints are the modern C++ mechanism (standardized in C++20) for expressing **template requirements** directly in the interface. They replace most uses of SFINAE (`enable_if`, detection tricks) in public APIs by making requirements:

- explicit and readable,
- checkable at compile time before instantiation proceeds,
- more diagnosable (typically clearer error messages),
- better-behaved in overload resolution (more predictable selection).

In the C++ Core Guidelines, constraints relate directly to the Template rules, especially:

- **T.10–T.13** (use concepts/standard concepts; prefer clear constrained forms),
- **T.20–T.26** (design meaningful concepts with semantics),

- **T.30–T.31** (use negation and disjunction sparingly),
- **T.41** (require only essential properties),
- **T.48** (use SFINAE only as a fallback when concepts are unavailable).

66.2 Terminology

66.2.1 Constraint, constraint-expression, constrained template

A **constraint** is a compile-time requirement associated with a template. A **constraint-expression** is a boolean compile-time expression used to decide whether a template is viable. A template is **constrained** if it has constraints (via a `requires`-clause, constrained parameters, or both).

66.2.2 Concept

A **concept** is a named constraint. Concepts make requirements reusable and readable.

```
#include <concepts>

template<class T>
concept Addable =
requires(T a, T b) {
    { a + b } -> std::same_as<T>;
};
```

66.2.3 Requires-clause vs requires-expression

- A **requires-clause** constrains a template or function declaration.

- A **requires-expression** is an expression that checks well-formedness and properties of operations.

66.3 Why constraints matter (engineering benefits)

66.3.1 Early rejection: fail fast and locally

Without constraints, templates often fail deep inside instantiation. With constraints, non-matching types are rejected at the interface boundary.

66.3.2 Better overload selection

Constraints participate in overload resolution in a more structured way than SFINAE-heavy patterns. This reduces ambiguity and accidental selection.

66.3.3 Readable contracts

A signature like `template<std::integral I>` is a human-readable contract.

66.4 Forms of constraints in C++

66.4.1 Constrained template parameters

```
#include <concepts>

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

66.4.2 Abbreviated function templates (constrained `auto`)

```
#include <concepts>

std::integral auto inc(std::integral auto x)
{
    return x + 1;
}
```

66.4.3 Requires-clause on a template/function

```
#include <concepts>

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

66.4.4 Requires-clause on a member function

```
#include <concepts>

template<class T>
struct box {
    T value;

    void reset() requires std::default_initializable<T>
    {
        value = T{};
    }
};
```

66.4.5 Requires-expression (checking operations)

A requires-expression checks that certain expressions are well-formed and may constrain types of results.

```
#include <concepts>

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

66.5 Designing good constraints (Core Guidelines aligned)

66.5.1 State requirements for every template parameter (T.10)

A template parameter is part of the interface. When you know the requirements, say them.

```
#include <concepts>
#include <iterator>

template<std::input_iterator It>
requires std::totally_ordered<std::iter_value_t<It>>
It min_element(It first, It last)
{
    if (first == last) return last;
    It best = first;
    for (++first; first != last; ++first) {
        if (*first < *best) best = first;
    }
    return best;
}
```

66.5.2 Use standard concepts whenever possible (T.11)

Standard concepts encode widely understood requirements and interoperate with the standard library. Examples include:

- iterators: `std::input_iterator`, `std::forward_iterator`,
- comparisons: `std::totally_ordered`, `std::equality_comparable`,
- callables: `std::invocable`, `std::regular_invocable`,
- type properties: `std::movable`, `std::copyable`, `std::regular`.

66.5.3 Require only what you need (T.41)

Over-constraining reduces usability. Prefer the minimal correct contract.

```
#include <concepts>

// Minimal: needs only equality to find a value
template<class T>
concept Findable =
requires(const T& a, const T& b) {
    { a == b } -> std::convertible_to<bool>;
};

template<class It, class T>
requires Findable<std::iter_value_t<It>> && Findable<T>
It find_value(It first, It last, const T& v)
{
    for (; first != last; ++first) {
        if (*first == v) return first;
    }
    return last;
}
```

66.5.4 Prefer meaningful concepts with semantics (T.20–T.22)

Do not define concepts as random syntactic checks. A concept should represent a coherent idea and, when needed, document semantic laws (axioms) such as:

- equivalence relation for equality,
- strict weak ordering for sorting predicates,
- consistency between hashing and equality.

```
#include <concepts>

template<class T>
concept Equality =
requires(const T& a, const T& b) {
    { a == b } -> std::convertible_to<bool>;
    { a != b } -> std::convertible_to<bool>;
};
// Semantic note (axiom): == is an equivalence relation.
```

66.5.5 Avoid negation and disjunction-heavy constraints (T.30–T.31)

Negation (`!C<T>`) and disjunction (`C1<T> || C2<T>`) are sometimes needed, but heavy use tends to:

- create confusing overload sets,
- make diagnostics harder,
- produce fragile selection behavior as code evolves.

Prefer clearer structure:

- separate functions,

- stronger types,
- tag dispatch or `if constexpr` internally.

66.6 Constraint satisfaction and overload resolution (practical model)

66.6.1 Viability

A constrained candidate participates in overload resolution only if its constraints are satisfied.

66.6.2 More constrained overloads are preferred (subsumption)

When multiple overloads are viable, the compiler prefers the **more constrained** overload when one constraint logically implies another. Design overload sets so the best match is unambiguous.

```
#include <concepts>
#include <string_view>

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

void print(int); // non-template

template<StringLike T>
void print(const T&); // prints text-like things
```

```
template<std::integral I>
void print(I); // more specific for integrals
```

Guideline advice. Prefer sets where:

- there is a clear “best” candidate for every intended call,
- unintended types are rejected by constraints rather than accepted accidentally.

66.7 Common constraint patterns

66.7.1 Pattern 1: Constrain a function template parameter

```
#include <concepts>

template<std::floating_point F>
F lerp(F a, F b, F t)
{
    return a + (b - a) * t;
}
```

66.7.2 Pattern 2: Constrain a class template and its operations

```
#include <concepts>

template<std::movable T>
class buffer {
public:
    void push(T x) { /*...*/ }
private:
    // ...
};
```

66.7.3 Pattern 3: Requires-expression for structural requirements

```
#include <concepts>

template<class T>
concept Streamable =
requires(std::ostream& os, const T& x) {
    { os << x } -> std::same_as<std::ostream&>;
};
```

66.7.4 Pattern 4: Internal selection with `if constexpr`

Constraints keep the interface clean; `if constexpr` chooses internal paths.

```
#include <type_traits>

template<class T>
void serialize(const T& x)
{
    if constexpr (std::is_trivially_copyable_v<T>) {
        // fast path
    } else {
        // general path
    }
}
```

66.8 Migrating from SFINAE to constraints (recommended approach)

SFINAE is often still necessary for older toolchains, but for modern C++:

- replace `enable_if` in signatures with named concepts,

- replace detection idiom checks in overload sets with requires-expressions,
- keep SFINAE for compatibility layers only (T.48).

66.8.1 Example rewrite

SFINAE style:

```
#include <type_traits>

template<class T, std::enable_if_t<std::is_integral_v<T>, int> = 0>
T inc(T x) { return x + 1; }
```

Constraint style:

```
#include <concepts>

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

66.9 Pitfalls and how to avoid them

66.9.1 Pitfall 1: Over-constraining public interfaces

Over-constraining blocks valid types and forces unnecessary adapters. Apply the minimal correct contract (T.41).

66.9.2 Pitfall 2: “syntactic” concepts with no semantic meaning

A concept that checks only that an expression compiles but does not encode meaningful behavior can be misleading (T.20). Prefer concepts that model domain ideas and document axioms (T.22).

66.9.3 Pitfall 3: Negation/disjunction producing fragile overload sets

Avoid large families of overloads that differ only by negated constraints (T.30, T.31). Use clearer structure or internal branching.

66.9.4 Pitfall 4: Accidental ADL customization points

Inside templates, be careful with unqualified non-member calls. If you intend customization, design it as an explicit customization point; otherwise, qualify calls.

66.10 Checklist for constraint reviews

1. Are requirements stated explicitly for template parameters? (T.10)
2. Are standard concepts used where appropriate? (T.11)
3. Are constraints minimal but sufficient? (T.41)
4. Do concepts represent meaningful semantics (not random syntax)? (T.20–T.22)
5. Are overload sets unambiguous and stable as the code evolves?
6. Is negation/disjunction used sparingly and only when it improves clarity? (T.30–T.31)
7. Are public signatures readable (named concepts rather than complex boolean expressions)?
8. Is SFINAE limited to compatibility needs? (T.48)

66.11 Summary

Constraints are the modern, guideline-aligned way to write safe and maintainable templates:

- express template requirements explicitly (concepts and `requires`),
- prefer standard concepts and meaningful semantics,
- require only what you need,
- design overload sets to be unambiguous and stable,
- use SFINAE primarily as a fallback for older compilers and compatibility layers.

Chapter 67

Concepts

67.1 Purpose and scope

Concepts are named compile-time requirements used to constrain templates. They were standardized in C++20 to address long-standing problems in template programming:

- unclear or missing template requirements,
- extremely long and hard-to-understand template errors,
- fragile SFINAE-based interfaces,
- unpredictable overload selection in large overload sets.

The C++ Core Guidelines strongly encourage expressing template requirements via concepts (or equivalent constraints), especially:

- **T.10** specify concepts for all template arguments,
- **T.11** prefer standard concepts,

- **T.12–T.13** prefer concept names and abbreviated templates where they improve clarity,
- **T.20–T.26** design concepts with meaning and semantics (axioms),
- **T.30–T.31** use negation and disjunction sparingly,
- **T.41** require only essential properties,
- **T.48** use SFINAE only when concepts are not available.

This chapter explains concepts precisely, shows best practices for designing and using them, and provides modern, maintainable patterns.

67.2 What a concept is

67.2.1 Definition

A **concept** is a compile-time predicate (a boolean constant expression) used by the language to determine whether template arguments satisfy a requirement. A concept is introduced with the `concept` keyword:

```
#include <concepts>

template<class T>
concept Integral = std::is_integral_v<T>;
```

67.2.2 Concepts are part of the interface

A constrained template is a contract:

- the signature states the required properties,
- invalid types are rejected early,
- diagnostics are typically clearer and more local.

67.3 Core syntax and forms

67.3.1 Constrained template parameters

```
#include <concepts>

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

67.3.2 Abbreviated function templates

```
#include <concepts>

std::integral auto inc(std::integral auto x)
{
    return x + 1;
}
```

67.3.3 Requires-clause using concepts

```
#include <concepts>

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

67.3.4 Requires-expression (defining a concept by use patterns)

A requires-expression checks that certain expressions are well-formed, and can constrain result types.

```
#include <concepts>

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

67.4 Prefer standard concepts (T.11)

The standard library provides widely used concepts that align with standard algorithms and ranges. Common categories:

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

Using standard concepts improves interoperability and reduces the risk of defining incomplete or incorrect requirements.

67.5 Designing good concepts (T.20–T.26)

67.5.1 Concepts should be meaningful (T.20)

A concept should represent a coherent idea (a model), not an arbitrary set of checks.

```
#include <concepts>

// GOOD: coherent requirement for sorting keys
template<class T>
concept SortKey = std::totally_ordered<T>;
```

67.5.2 Require a complete set of operations (T.21)

If your algorithm needs multiple related operations, require them all. Do not “forget” a required operation and let errors appear deep in instantiation.

```
#include <concepts>

template<class T>
concept Equality =
requires(const T& a, const T& b) {
    { a == b } -> std::convertible_to<bool>;
    { a != b } -> std::convertible_to<bool>;
};
```

67.5.3 Specify axioms (semantic laws) when required (T.22)

Some requirements are semantic:

- equality should be an equivalence relation,
- ordering should be consistent and transitive (as required by algorithms),

- hashing must be consistent with equality.

C++ cannot enforce axioms directly; document them and test them.

```
// Axiom (semantic requirement):
// For all a, b, c: if (a == b) then f(a) == f(b) for any observable property f used by the
↪  algorithm.
```

67.5.4 Refinement should add capability (T.23)

A refined concept should enable additional valid usage patterns.

```
#include <concepts>

template<class T>
concept Addable =
requires(T a, T b) {
    { a + b } -> std::same_as<T>;
};

template<class T>
concept AddableAndOrdered = Addable<T> && std::totally_ordered<T>;
```

67.5.5 Separate semantics when operations are identical (T.24)

If two domains have identical operations but different meaning, use tags/strong types.

```
struct meters_tag {};
struct seconds_tag {};

template<class Rep, class Tag>
struct quantity {
    Rep value{};
};
```

```
using meters  = quantity<double, meters_tag>;
using seconds = quantity<double, seconds_tag>;
```

67.5.6 Avoid complementary concepts as primary design tools (T.25)

Avoid designing with `C` and `not_C` as two equally central categories. This often creates brittle overload sets and unclear intent.

67.5.7 Prefer use-pattern concepts (T.26)

Define requirements by how types are used, not by member names.

```
#include <concepts>

template<class T>
concept Addable =
requires(T a, T b) {
    { a + b } -> std::same_as<T>;
};
```

67.6 Using concepts in template interfaces (T.10, T.12, T.13, T.41)

67.6.1 T.10: Specify concepts for all template parameters

This turns a template into a readable contract.

```
#include <concepts>
#include <iterator>
```

```
template<std::input_iterator It>
requires std::equality_comparable<std::iter_value_t<It>>
It find_first(It first, It last, const std::iter_value_t<It>& v)
{
    for (; first != last; ++first) {
        if (*first == v) return first;
    }
    return last;
}
```

67.6.2 T.12: Prefer concept names over `auto` when requirements matter

When the exact type is not important but the requirement is, writing the concept improves readability.

```
#include <concepts>

template<std::integral I>
I twice(I x)
{
    std::integral auto y = x + x; // requirement is explicit
    return y;
}
```

67.6.3 T.13: Prefer abbreviated templates when they are simplest

Use abbreviated syntax for small functions with obvious constraints.

```
#include <concepts>

auto abs_value(std::signed_integral auto x)
{
    return x < 0 ? -x : x;
}
```

67.6.4 T.41: Require only essential properties

Avoid requiring `std::regular` or `std::totally_ordered` when you only need, for example, increment and sentinel comparison. Minimal constraints make code more usable and less brittle.

67.7 Concept composition (T.30–T.31)

67.7.1 Use negation sparingly (T.30)

Negation is sometimes useful but can create confusing overload sets.

```
#include <concepts>

template<class T>
concept NotIntegral = !std::integral<T>;
```

Prefer positive concepts that describe intent.

67.7.2 Use disjunction sparingly (T.31)

Disjunction (`C1<T> || C2<T>`) should represent a real alternative with clear semantics. If the two cases behave differently, consider:

- separate overloads with clear names,
- stronger types,
- internal `if constexpr` branching.

67.8 Overload resolution and subsumption (practical understanding)

When multiple constrained overloads are viable, the language prefers a **more constrained** overload when one constraint implies another. Design overload sets so that:

- for every intended call, there is a clear best overload,
- the ordering is stable under maintenance.

```
#include <concepts>
#include <string_view>

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

void print(const char*);              // best for C strings
template<StringLike T> void print(T); // text-like types
template<std::integral I> void print(I); // integral types
```

67.9 Concepts vs SFINAE (migration guidance)

67.9.1 Concepts are the modern replacement

SFINAE remains relevant for older compilers, but for C++20+:

- constraints are clearer and more maintainable,
- diagnostics are typically better,

- interfaces become self-documenting.

67.9.2 Rewrite example

SFINAE style:

```
#include <type_traits>

template<class T, std::enable_if_t<std::is_integral_v<T>, int> = 0>
T inc(T x) { return x + 1; }
```

Concept style:

```
#include <concepts>

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

67.10 Pitfalls and best practices

67.10.1 Pitfall 1: Concepts that check syntax but not meaning

If a concept does not encode the intended semantics, it can accept types that "compile" but violate expectations. Document axioms and test.

67.10.2 Pitfall 2: Over-constraining breaks interoperability

Requiring too much (e.g., `std::regular` when not necessary) blocks useful types and forces adapters.

67.10.3 Pitfall 3: Unstable overload sets with negation/disjunction

Avoid large overload families based on negated concepts. Prefer clearer design.

67.10.4 Pitfall 4: Accidental customization points

Be careful with unqualified calls inside templates. If you intend customization, design it explicitly; otherwise, qualify calls.

67.11 Checklist for concept-based designs

1. Are requirements stated for template parameters? (T.10)
2. Are standard concepts used where they fit? (T.11)
3. Are concepts meaningful and coherent? (T.20)
4. Are required operations complete and sufficient? (T.21)
5. Are semantic laws (axioms) documented where relevant? (T.22)
6. Is refinement meaningful (adds capabilities)? (T.23)
7. Are semantic differences separated with strong types/tags? (T.24)
8. Are negation and disjunction used sparingly? (T.30, T.31)
9. Are constraints minimal (no over-constraint)? (T.41)
10. Are overload sets stable and unambiguous (subsumption-aware)?

67.12 Summary

Concepts bring templates closer to normal engineering interfaces:

- They express contracts explicitly.

- They improve correctness and diagnostics.
- They integrate with overload resolution in a structured way.
- They encourage good design: meaningful requirements, minimal constraints, and documented semantics.

Chapter 68

Reducing Template Error Complexity

68.1 Purpose and scope

Template errors can be intimidating because:

- templates are instantiated lazily and deeply,
- errors often occur far from the call site,
- overload resolution may explore many candidates,
- failure messages may include long instantiation traces.

Reducing template error complexity is an engineering goal for:

- faster debugging and higher productivity,
- safer code review (less guessing),
- better library usability,

- lower maintenance cost.

The C++ Core Guidelines address this primarily through:

- **T.10–T.13**: express template requirements with concepts and clear constraints,
- **T.20–T.26**: define meaningful concepts with complete requirements and documented semantics,
- **T.41**: require only essential properties (avoid over-constraints),
- **T.47**: avoid highly visible unconstrained templates with common names,
- **T.48**: use SFINAE only as a fallback; do it in a controlled way,
- **T.60**: minimize context dependencies,
- **T.69**: avoid unintended ADL/unqualified calls as accidental customization points.

This chapter presents practical, modern techniques to make template failures short, local, and actionable.

68.2 Root causes of complex template errors

68.2.1 Missing requirements (no contract)

If a template does not state what it needs, the first error appears at the deepest point where an operation is used.

68.2.2 Overly generic overload sets

Unconstrained templates participate in overload resolution for many types, producing long lists of candidate failures.

68.2.3 Accidental substitution and ADL surprises

Unqualified calls inside templates can trigger ADL and unexpected overloads, creating confusing errors and behavior changes.

68.2.4 Type-level metaprogramming without clear boundaries

Heavy SFINAE/type-trait machinery can generate errors that refer to internal helper types rather than the user's code.

68.3 Technique 1: Constrain templates at the interface (T.10)

68.3.1 Use concepts/requires to fail at the call site

A constrained template rejects invalid types before instantiating the body, producing more local diagnostics.

```
#include <concepts>
#include <ostream>

template<class T>
concept Streamable =
requires(std::ostream& os, const T& x) {
    { os << x } -> std::same_as<std::ostream&>;
};

template<Streamable T>
void log_value(const T& x)
{
    std::cout << x << "\n";
}
```

If `T` is not streamable, the error points to the failed constraint rather than a deep instantiation.

68.3.2 Prefer standard concepts (T.11)

Standard concepts provide familiar, well-tested contracts and often yield cleaner diagnostics.

68.4 Technique 2: Name requirements (avoid “giant boolean” constraints)

Long, inline constraints can be as unreadable as SFINAE. Split them into named concepts.

```
#include <concepts>

template<class T>
concept Addable =
requires(T a, T b) {
    { a + b } -> std::same_as<T>;
};

template<class T>
concept AddableAndOrdered = Addable<T> && std::totally_ordered<T>;

template<AddableAndOrdered T>
T min_plus(T a, T b)
{
    return (a < b) ? (a + b) : (b + a);
}
```

68.5 Technique 3: Require only what you need (T.41)

Over-constraining often produces confusing failures for types that could otherwise work. Use minimal correct requirements.

```
#include <concepts>
#include <iterator>

// GOOD: requires only equality on values
template<std::input_iterator It, class V>
requires std::equality_comparable_with<std::iter_value_t<It>, V>
It find_value(It first, It last, const V& v)
{
    for (; first != last; ++first) {
        if (*first == v) return first;
    }
    return last;
}
```

Avoid requiring `std::totally_ordered` or `std::regular` when not needed.

68.6 Technique 4: Keep overload sets small and constrained (T.47)

68.6.1 Do not publish unconstrained templates with common names

An unconstrained `swap`, `begin`, `print`, `to_string`, etc. can hijack overload resolution.

```
// BAD: unconstrained, common name, in a public header
template<class T>
void print(const T&);
```

```
// GOOD: constrain and/or avoid common names
#include <concepts>
#include <string_view>

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

template<StringLike T>
void print_text(const T& t);
```

68.6.2 Use “detail” namespaces for helper templates

Keep heavy machinery out of the user-visible interface.

```
namespace detail {
    template<class T>
    struct helper { /* ... */ };
}
```

68.7 Technique 5: Prefer constraints over SFINAE in interfaces (T.48)

SFINAE-based constraints often generate long and cryptic substitution traces. When concepts are available, prefer them.

68.7.1 If SFINAE is required, isolate it and name it

If you must use SFINAE, keep it behind a readable alias or trait.

```
#include <type_traits>

template<class T>
inline constexpr bool is_int_v = std::is_integral_v<T>;

template<class T, std::enable_if_t<is_int_v<T>, int> = 0>
T inc(T x) { return x + 1; }
```

68.8 Technique 6: Use `static_assert` with a clear message (localize failure)

Even with constraints, some requirements are semantic or cannot be expressed fully in a concept. Use `static_assert` at the boundary to provide an explicit message.

```
#include <type_traits>

template<class T>
void process(T)
{
    static_assert(std::is_trivially_copyable_v<T>,
        "process(T): T must be trivially copyable (required for byte-wise processing).");
}
```

Place such asserts:

- at the start of the template,
- or in a small helper that is instantiated early.

68.9 Technique 7: Prefer `requires`-expressions for diagnostics in concepts

A well-written concept can guide errors to the missing operation.

```
#include <concepts>

template<class T>
concept Reservable =
requires(T& t, std::size_t n) {
    t.reserve(n);
};

template<class T>
void prepare(T& t, std::size_t n)
{
    if constexpr (Reservable<T>) {
        t.reserve(n);
    }
}
```

This pattern avoids hard errors and keeps a single instantiation path.

68.10 Technique 8: Control ADL and unqualified calls (T.69)

Unqualified non-member calls inside templates can produce confusing errors and unstable behavior. Qualify calls unless you are intentionally defining a customization point.

```
#include <cmath>

template<class T>
```

```
T magnitude(T x)
{
    return std::abs(x); // qualified: no accidental ADL
}
```

68.11 Technique 9: Minimize template context dependencies (T.60)

Templates are often defined in headers; accidental dependencies increase error noise and fragility. Keep dependencies explicit:

- include what you use,
- qualify names where appropriate,
- avoid relying on include order.

```
#include <algorithm>
#include <iterator>

template<class C>
void sort_in_place(C& c)
{
    using std::begin;
    using std::end;
    std::sort(begin(c), end(c));
}
```

68.12 Technique 10: Reduce instantiation depth and type noise

68.12.1 Use aliases to shorten types

Long types appear in diagnostics; aliases make both code and errors smaller.

```
#include <vector>
#include <memory>

template<class T>
using vec_uptr = std::vector<std::unique_ptr<T>>;
```

68.12.2 Prefer simple metaprogramming patterns

Prefer:

- `if constexpr`,
- named concepts,
- simple traits,

over multi-layered SFINAE chains when possible.

68.13 Technique 11: Provide readable customization points

If customization is intended, design it explicitly so diagnostics mention a well-named hook.

```
namespace mylib {

template<class T>
void format(const T&); // customization point (document required behavior)

template<class T>
void print(const T& x)
```

```
{
    mylib::format(x); // qualified and intentional
}

} // namespace mylib
```

68.14 A practical “error complexity” checklist

1. Are template requirements stated as concepts/constraints at the interface? (T.10)
2. Are constraints minimal and meaningful, not overly strict? (T.41, T.20)
3. Are constraints named (concepts) rather than huge inline boolean expressions?
4. Are overload sets constrained, avoiding unconstrained public templates with common names? (T.47)
5. Is SFINAE limited to compatibility and isolated from user interfaces? (T.48)
6. Are ADL and unqualified calls controlled unless customization is intended? (T.69)
7. Are dependencies explicit and stable (no include-order surprises)? (T.60)
8. Are `static_assert` messages used where semantic requirements cannot be expressed fully?
9. Are long types reduced via aliases and simpler patterns?

68.15 Summary

You reduce template error complexity by designing templates as clear contracts:

- constrain templates using concepts and requires-clauses (fail early, near the call site),

- name requirements with meaningful concepts,
- require only essential properties,
- avoid large unconstrained overload sets and common-name templates,
- control ADL and context dependencies,
- isolate SFINAE and use it only as a fallback,
- add targeted `static_assert` messages for semantic requirements.

Part XI

Error Handling and Exceptions (E)

Chapter 69

Error Handling and Exceptions (E.1–E.30)

69.1 Overview: what error handling must achieve

Error handling is not only about reporting failure; it must also preserve program correctness and resource safety. The C++ Core Guidelines summarize error handling as:

- detecting an error,
- transmitting information about the error,
- preserving a valid program state,
- avoiding resource leaks.

A good strategy must be simple, consistent, and systematic; complex, rarely-executed error-handling paths are a common source of bugs.

69.2 Rule summary: E.1–E.30 and numbering gaps

The current C++ Core Guidelines define the following rules in the E section:

- E.1, E.2, E.3, E.4, E.5, E.6, E.7, E.8,
- E.12, E.13, E.14, E.15, E.16, E.17, E.18, E.19,
- E.25, E.26, E.27, E.28,
- E.30.

Numbers E.9–E.11, E.20–E.24, and E.29 are not currently defined in the rule summary (they are effectively unused/absent in the published guideline revision referenced in this book).

69.3 E.1: Develop an error-handling strategy early in a design

Rule

Develop an error-handling strategy early in a design.

Reason

A consistent and complete strategy for handling errors and resource leaks is hard to retrofit into a system.

Practical guidance

Decide early and document:

- Which failures are *recoverable* vs *fatal*.
- Whether the codebase uses exceptions, error codes, or a mixed strategy (and where).
- How resources are managed (RAII by default).
- How invariants are maintained during failure paths (E.4).

69.4 E.2: Throw an exception to signal that a function can't perform its assigned task

Rule

Throw an exception to signal that a function can't perform its assigned task.

Reason

To make error handling systematic, robust, and non-repetitive.

Key idea

Use exceptions for **failure to perform the function's contract**, such as:

- failing to meet a required precondition at runtime,
- failing to establish a constructor's invariant,
- inability to acquire a required resource (file, memory, lock, network),
- out-of-range access.

Do not use exceptions as an alternative to normal returns.

Example: resource acquisition failure

```
#include <cstdio>
#include <stdexcept>
#include <string>

class File_handle {
```

```
public:
    File_handle(const std::string& name, const char* mode)
        : f_{std::fopen(name.c_str(), mode)}
    {
        if (!f_) {
            throw std::runtime_error{"File_handle: could not open file"};
        }
    }

    ~File_handle() noexcept
    {
        if (f_) std::fclose(f_);
    }

    std::FILE* get() const noexcept { return f_; }

    File_handle(const File_handle&) = delete;
    File_handle& operator=(const File_handle&) = delete;

    File_handle(File_handle&& other) noexcept : f_{other.f_} { other.f_ = nullptr; }
    File_handle& operator=(File_handle&& other) noexcept
    {
        if (this != &other) {
            if (f_) std::fclose(f_);
            f_ = other.f_;
            other.f_ = nullptr;
        }
        return *this;
    }

private:
    std::FILE* f_{nullptr};
};
```

69.5 E.3: Use exceptions for error handling only

Rule

Use exceptions for error handling only.

Reason

To keep error handling separated from ordinary code; implementations tend to be optimized assuming exceptions are rare.

Anti-pattern

Throwing to return "normal" results (e.g., throwing an index when a value is found) mixes logic and error handling and usually harms performance and clarity.

69.6 E.4: Design your error-handling strategy around invariants

Rule

Design your error-handling strategy around invariants.

Reason

To use an object it must be in a valid state (as defined by its invariant). To recover from an error, every object that survives must still be valid.

Practical guidance

- Define class invariants clearly (in comments and tests).
- Ensure operations provide at least the basic guarantee: after failure, objects remain valid and destructible.
- Prefer designs that naturally provide strong guarantees via RAII and value semantics.

69.7 E.5: Let a constructor establish an invariant, and throw if it cannot

Rule

Let a constructor establish an invariant, and throw if it cannot.

Reason

Leaving an object without its invariant established is asking for trouble; not all member functions can be called safely.

Guidance

- If an object cannot be made valid, do not create it.
- Favor types that are either fully constructed and valid, or not constructed at all.

Example: constructor validates and throws

```
#include <stdexcept>
#include <string>
```

```
class Port {
public:
    explicit Port(int p)
    {
        if (p <= 0 || p > 65535) {
            throw std::out_of_range{"Port: invalid port number"};
        }
        port_ = p;
    }

    int value() const noexcept { return port_; }

private:
    int port_{0}; // invariant: 1..65535
};
```

69.8 E.6: Use RAII to prevent leaks

Rule

Use RAII to prevent leaks.

Reason

RAII ties resource lifetime to object lifetime, ensuring cleanup even on exceptional exits.

Guidance

Prefer standard RAII handles:

- memory: `std::unique_ptr`, `std::shared_ptr`, containers

- locks: `std::lock_guard`, `std::unique_lock`
- threads: `std::jthread` (C++20)
- files/sockets/OS handles: a small owning wrapper type

Example: lock safety

```
#include <mutex>

std::mutex m;

void update()
{
    std::lock_guard<std::mutex> lock{m};
    // ... do work ...
    // lock released automatically, even if an exception is thrown
}
```

69.9 E.7: State your preconditions

Rule

State your preconditions.

Purpose

A precondition communicates what must be true before a function is called. It is essential to decide:

- what is a caller bug (precondition violated),
- what is a runtime failure (cannot complete task).

How to express

- Document preconditions in comments and in the interface (types, concepts, `gsl::span`, etc.).
- When a violated precondition is detected at runtime, choose a consistent policy: throw (if recoverable) or fail fast (if it indicates a programming error).

69.10 E.8: State your postconditions

Rule

State your postconditions.

Purpose

A postcondition documents what the function guarantees after successful completion and helps reason about error paths and invariants.

How to express

- Document postconditions; validate with tests.
- When possible, design APIs so postconditions are enforced structurally (RAII, returned values, strong types).

69.11 E.9–E.11: (Not defined in the current guideline revision)

These rule numbers are not listed in the E section rule summary for the referenced guideline revision.

69.12 E.12: Use `noexcept` when exiting a function because of a throw is impossible or unacceptable

Rule

Use `noexcept` when exiting a function because of a throw is impossible or unacceptable.

Practical meaning

Mark functions `noexcept` when:

- they must not throw by design (destructors, swap-like operations, many move operations),
- throwing would break required guarantees (e.g., during stack unwinding).

Example: `noexcept` move and swap

```
#include <utility>

struct S {
    S() = default;
    S(S&&) noexcept = default;
    S& operator=(S&&) noexcept = default;

    friend void swap(S& a, S& b) noexcept
    {
        using std::swap;
        swap(a.x, b.x);
    }
```

```
    int x{};
};
```

69.13 E.13: Never throw while being the direct owner of an object

Rule

Never throw while being the direct owner of an object.

Meaning

If you own resources directly (raw handles, raw pointers, manual cleanup), a throw can easily cause leaks and partial cleanup. The remedy is to **transfer ownership into an RAII handle immediately** (E.6), so exceptions do not bypass cleanup.

Guidance

- Acquire into an RAII wrapper in the same expression or immediately after.
- Avoid "owning raw pointer variables" whose cleanup depends on later code.

69.14 E.14: Use purpose-designed user-defined types as exceptions (not built-in types)

Rule

Use purpose-designed user-defined types as exceptions (not built-in types).

Why

Typed exceptions communicate meaning and allow structured handling.

Example

```
#include <stdexcept>
#include <string>

class parse_error : public std::runtime_error {
public:
    explicit parse_error(const std::string& msg)
        : std::runtime_error{msg} {}
};
```

69.15 E.15: Throw by value, catch exceptions from a hierarchy by reference

Rule

Throw by value, catch exceptions from a hierarchy by reference.

Why

- Throwing by value preserves the exception object.
- Catching by (const) reference avoids slicing and unnecessary copying.

Example

```
#include <stdexcept>
```

```
void f()
{
    throw std::runtime_error{"failure"};
}

void g()
{
    try {
        f();
    }
    catch (const std::runtime_error& e) { /* handle */ }
    catch (const std::exception& e)     { /* generic handle */ }
}
```

69.16 E.16: Destructors, deallocation, swap, and exception type copy/move construction must never fail

Rule

Destructors, deallocation, `swap`, and exception type copy/move construction must never fail.

Why

Failures in these contexts can lead to termination during stack unwinding or break fundamental guarantees. This rule supports reliable cleanup and exception safety.

Practical guidance

- Destructors should be `noexcept` by default.

- `swap` should be `noexcept` whenever feasible.
- Exception objects should be safe to copy/move without throwing.

69.17 E.17: Don't try to catch every exception in every function

Rule

Don't try to catch every exception in every function.

Why

Catching everywhere causes repetition, hides real failure points, and often prevents correct recovery. Catch at boundaries where you can:

- add context/logging,
- translate to an error-reporting mechanism,
- safely terminate or recover.

69.18 E.18: Minimize the use of explicit `try`/`catch`

Rule

Minimize the use of explicit `try`/`catch`.

Meaning

Most cleanup should be automatic via RAII (E.6), not manual via catch blocks. Use `try/catch` mainly to:

- handle at an architectural boundary,
- add context and rethrow,
- translate exceptions to error codes when required.

Example: boundary handling with context

```
#include <stdexcept>
#include <string>

void low_level();

void api_entry()
{
    try {
        low_level();
    }
    catch (const std::exception& e) {
        throw std::runtime_error{std::string{"api_entry failed: "} + e.what()};
    }
}
```

69.19 E.19: Use a `final_action` object to express cleanup if no suitable resource handle is available

Rule

Use a `final_action` object to express cleanup if no suitable resource handle is available.

Purpose

Sometimes you must call a cleanup function even though no standard RAII type exists. A scope guard expresses "do this on scope exit" safely and locally.

Example: minimal scope guard

```
#include <utility>

template<class F>
class final_action {
public:
    explicit final_action(F f) noexcept : f_{std::move(f)} {}
    final_action(final_action&& other) noexcept
        : f_{std::move(other.f_)}, active_{other.active_}
    {
        other.active_ = false;
    }
    final_action(const final_action&) = delete;
    final_action& operator=(const final_action&) = delete;

    ~final_action() noexcept
    {
        if (active_) f_();
    }

private:
    F f_;
```

```
    bool active_{true};
};

template<class F>
final_action<F> finally(F f) noexcept { return final_action<F>{std::move(f)}; }
```

```
#include <cstdio>

void example(std::FILE* f)
{
    auto cleanup = finally([&] noexcept { if (f) std::fclose(f); });

    // ... do work that may throw ...
}
```

69.20 E.20–E.24: (Not defined in the current guideline revision)

These rule numbers are not listed in the E section rule summary for the referenced guideline revision.

69.21 E.25: If you can’t throw exceptions, simulate RAII for resource management

Rule

If you can’t throw exceptions, simulate RAII for resource management.

Meaning

Even without exceptions, resource safety must be automatic. Use RAII objects so that early returns and error codes still trigger cleanup.

Example: return-based error handling still uses RAII

```
#include <cstdio>

class File {
public:
    explicit File(const char* name, const char* mode) noexcept
        : f_{std::fopen(name, mode)} {}

    ~File() noexcept { if (f_) std::fclose(f_); }

    std::FILE* get() const noexcept { return f_; }
    explicit operator bool() const noexcept { return f_ != nullptr; }

private:
    std::FILE* f_{nullptr};
};

bool do_work_no_exceptions()
{
    File f{"data.txt", "r"};
    if (!f) return false;

    // ... work, return false on error ...
    return true;
}
```

69.22 E.26: If you can't throw exceptions, consider failing fast

Rule

If you can't throw exceptions, consider failing fast.

Meaning

If you cannot propagate detailed error information reliably (no exceptions, weak error-code discipline), a fail-fast strategy can be safer than continuing in a potentially corrupted or unknown state.

69.23 E.27: If you can't throw exceptions, use error codes systematically

Rule

If you can't throw exceptions, use error codes systematically.

Guidance

- Define a uniform error type (enum/struct) and a consistent convention.
- Do not mix "sometimes returns error code" with "sometimes throws" arbitrarily.
- Ensure every call site handles or propagates errors consistently.

Example: structured error code

```
#include <cstdint>
```

```
enum class errc : std::uint8_t {
    ok = 0,
    io_error,
    parse_error,
    out_of_memory
};

struct result {
    errc ec;
};

result run() noexcept
{
    // ...
    return {errc::ok};
}
```

69.24 E.28: Avoid error handling based on global state (e.g., `errno`)

Rule

Avoid error handling based on global state (e.g., `errno`).

Reason

Global state is hard to manage and easy to forget to check.

Practical guidance

- Prefer APIs that return explicit error objects or throw.
- If you must call C APIs that use `errno`, translate immediately into an exception or a local error object.

69.25 E.29: (Not defined in the current guideline revision)

This rule number is not listed in the E section rule summary for the referenced guideline revision.

69.26 E.30: Don’t use exception specifications

Rule

Don’t use exception specifications.

Reason

Exception specifications make error handling brittle, impose runtime cost, and have been removed from the C++ standard.

Modern guidance

- Do not write dynamic exception specifications such as `void f() throw(X,Y)`.
- If no exception can be thrown, use `noexcept` (see E.12).

69.27 Chapter checklist (engineering practice)

1. Do we have a documented project-wide error-handling strategy (exceptions vs error codes vs mixed)? (E.1)
2. Do functions throw only to signal inability to complete their contract? (E.2, E.3)
3. Are invariants defined and preserved on all failure paths? (E.4, E.5)
4. Are all resources managed via RAII or scope guards? (E.6, E.19)
5. Are preconditions and postconditions stated and tested? (E.7, E.8)
6. Are `noexcept` decisions intentional and consistent for destructors/moves/swaps? (E.12, E.16)
7. Are exceptions typed, thrown by value, and caught by reference? (E.14, E.15)
8. Are `try`/`catch` blocks concentrated at boundaries rather than everywhere? (E.17, E.18)
9. In no-exception builds, is error-code discipline systematic and resource-safe? (E.25–E.27)
10. Are we avoiding global-state error handling and translating C-style errors promptly? (E.28)

Chapter 70

When to Use Exceptions

70.1 Purpose and scope

This chapter explains **when exceptions are the right tool** in C++ and when they are not. It is aligned with the C++ Core Guidelines error-handling rules, especially:

- E.1 (design an error-handling strategy early),
- E.2 (throw to signal inability to perform a task),
- E.3 (use exceptions for error handling only),
- E.4–E.6 (invariants, constructors, RAII),
- E.12, E.16 (noexcept and operations that must not fail),
- E.17–E.18 (catch at boundaries; minimize try/catch),
- E.25–E.27 (what to do when exceptions cannot be used).

70.2 Key definition: what an exception should mean

In disciplined C++ design, **throwing an exception means**:

> The function could not complete its assigned task while preserving its contract.

This is not a "control flow feature" and not a replacement for normal return values. It is the mechanism for reporting failure while letting normal code remain clean.

70.3 Use exceptions when these conditions hold

70.3.1 1) The failure is not part of normal expected control flow (E.2, E.3)

Exceptions are intended for **rare or exceptional** failures. If failure is a normal and frequent outcome, prefer a return-based design (e.g., returning a value that can represent "not found").

70.3.2 2) The caller cannot reasonably continue without handling the failure

If continuing would violate invariants or create undefined behavior, the error must be handled explicitly. Exceptions force the caller to either handle or propagate the failure.

70.3.3 3) You need to propagate failure across multiple layers cleanly

Exceptions avoid repetitive boilerplate after every call. They propagate automatically until a boundary that can handle them.

70.3.4 4) A constructor cannot establish its invariant (E.5)

If an object cannot be constructed into a valid state, it should not exist. Throw from constructors to prevent invalid objects from escaping.

```
#include <stdexcept>

class socket_handle {
public:
    socket_handle()  // establishes invariant: valid OS handle
    {
        // if OS allocation fails:
        throw std::runtime_error{"socket_handle: could not acquire socket"};
    }
};
```

70.3.5 5) Resource acquisition fails, but RAII ensures cleanup (E.6)

Exceptions and RAII are designed to work together. If a function acquires resources using RAII handles, then an exception cannot leak resources.

```
#include <fstream>
#include <stdexcept>
#include <string>

std::string read_all(const std::string& path)
{
    std::ifstream in(path);
    if (!in) {
        throw std::runtime_error{"read_all: cannot open file"};
    }

    std::string s{std::istreambuf_iterator<char>(in), std::istreambuf_iterator<char>()};
    return s; // resources released automatically
}
```

70.4 Do not use exceptions in these cases

70.4.1 1) To signal ordinary outcomes (E.3)

If “failure” is expected and frequent, throwing is usually the wrong model.

```
// BAD: throwing for a normal "not found" outcome is usually wrong
int find_index_throwing(/*...*/); // avoid this design
```

Prefer designs like:

- return an iterator or pointer (nullable observer),
- return `bool` plus an out-parameter,
- return `std::optional<T>` when absence is normal,
- return a result type or error code when exceptions are not used.

70.4.2 2) In low-level code where exceptions are forbidden by policy

Some environments disable exceptions (kernel code, certain embedded profiles, safety-certified codebases, or strict ABI boundaries). In these cases follow the Core Guidelines:

- use RAII anyway (E.25),
- use error codes systematically (E.27),
- consider failing fast when recovery is impossible or unsafe (E.26),
- avoid global state error handling such as `errno` as the primary mechanism (E.28).

70.4.3 3) In destructors and during stack unwinding (E.16)

Destructors must not fail. Throwing from a destructor during stack unwinding can terminate the program. Design destructors to be `noexcept` and handle failures internally (log, terminate safely, or store error state for later).

```
#include <cstdio>

class File {
public:
    ~File() noexcept
    {
        // Never throw here.
        // If fclose fails, handle internally (log, set flag, etc.).
        if (f_) std::fclose(f_);
    }
private:
    std::FILE* f_{nullptr};
};
```

70.4.4 4) To replace precondition checking strategy

Violating a precondition is a bug at the call site. Your project must choose a consistent approach:

- fail fast (terminate) for contract violations, or
- throw an exception if you treat it as a recoverable runtime error.

The important part is consistency and clarity of the contract (E.7, E.8).

70.4.5 5) In performance-critical inner loops when failures are frequent

Even when exceptions are enabled, designing frequent failure as exceptions can lead to poor performance and complex behavior. If “failure” is expected, model it as a normal return path.

70.5 A clear practical rule: exceptions vs normal returns

70.5.1 Use exceptions when

- the function **cannot do its job** (E.2),
- the failure is **rare** (E.3),
- callers benefit from **automatic propagation** across layers,
- RAII guarantees cleanup (E.6),
- constructors cannot establish invariants (E.5).

70.5.2 Prefer return-based results when

- the outcome is **expected and common** (e.g., search misses),
- you need a **fast branch** rather than an exception path,
- the environment forbids exceptions by policy (E.25–E.27),
- the API boundary is C-only or requires stable ABI without exceptions crossing.

70.6 How to write exception-based APIs correctly

70.6.1 Throw types that communicate meaning (E.14)

Throw purpose-designed exception types (often derived from `std::exception`) rather than built-in types.

70.6.2 Throw by value, catch by reference (E.15)

```
#include <stdexcept>

void parse()
{
    throw std::runtime_error{"parse failed"};
}

void run()
{
    try {
        parse();
    }
    catch (const std::exception& e) {
        // handle or translate
    }
}
```

70.6.3 Mark functions `noexcept` when throwing is impossible or unacceptable (E.12)

Common candidates:

- destructors,

- move operations when required for strong exception-safety and optimization,
- swap,
- cleanup operations.

70.7 Where to catch exceptions (E.17, E.18)

Catch exceptions at **architectural boundaries**:

- thread entry points,
- the top level of a request handler,
- UI/event loop boundaries,
- service boundaries where you translate to error codes or error responses.

Do not litter code with `try`/`catch`; rely on RAII for cleanup and let exceptions propagate until a meaningful boundary can handle them.

```
#include <exception>
#include <iostream>

int main()
{
    try {
        // application logic
    }
    catch (const std::exception& e) {
        std::cerr << "fatal error: " << e.what() << "\n";
        return 1;
    }
}
```

70.8 Exception safety and resource safety (E.4–E.6)

Exceptions must not compromise program state:

- maintain class invariants (E.4),
- let constructors establish invariants or throw (E.5),
- prevent leaks using RAII (E.6).

A simple and effective rule:

> If your code uses RAII for all resources, you can usually let exceptions propagate safely.

70.9 No-exception builds: what to do instead (E.25–E.27)

If exceptions are disabled:

- still use RAII for resource management (E.25),
- use error codes systematically (E.27),
- consider failing fast if recovery is not reliable (E.26),
- avoid relying on global state like `errno` as the main strategy (E.28).

70.10 Checklist: deciding "exceptions or not" for an API

1. Is failure rare and truly exceptional for typical use? (E.3)
2. Does failure mean the function cannot complete its contract? (E.2)

3. Will exceptions cross boundaries where they are forbidden (C ABI, OS callbacks, plugins)? If yes, translate at the boundary.
4. Are all acquired resources managed by RAII so throwing cannot leak? (E.6)
5. Can the type's invariant be preserved on failure paths? (E.4)
6. Is the failure a normal outcome (e.g., search miss)? If yes, prefer return-based results.
7. Must any involved operations be `noexcept` (destructors, swap, moves)? (E.12, E.16)

70.11 Summary

Use exceptions when a function cannot do its job and the failure is not part of normal control flow. Design exception handling around invariants and RAII:

- throw for failure to meet the function contract (E.2),
- use exceptions for error handling only (E.3),
- preserve invariants (E.4) and throw from constructors if invariants cannot be established (E.5),
- prevent leaks with RAII (E.6),
- use `noexcept` where throwing is impossible or unacceptable (E.12, E.16),
- catch at boundaries, not everywhere (E.17, E.18),
- if exceptions are disabled, use RAII and systematic error codes or fail fast (E.25–E.27).

Chapter 71

Exception Guarantees

71.1 Purpose and scope

Exception guarantees describe what a function promises about program state *if it throws*. They are the engineering foundation for writing robust C++:

- callers know what remains valid after failure,
- invariants are preserved,
- resources are not leaked,
- recovery logic is possible and safe.

The C++ Core Guidelines emphasize designing error handling around invariants and RAII (E.4–E.6), using `noexcept` where throwing is impossible or unacceptable (E.12), and ensuring critical operations that participate in error handling (destructors, deallocation, swap) do not fail (E.16).

This chapter defines the standard exception safety guarantees, shows how to design APIs that provide them, and connects them to modern C++ practice.

71.2 Why guarantees matter: invariants and recovery

71.2.1 Invariants (E.4)

An **invariant** is the set of conditions that must always be true for an object to be usable. Exception guarantees are meaningful only relative to invariants: after an exception, every surviving object must still satisfy its invariant.

71.2.2 RAII (E.6)

RAII ensures resources are released during stack unwinding. Without RAII, exception guarantees are often meaningless because leaks and partial cleanup can invalidate program state.

71.3 The four commonly used exception safety guarantees

71.3.1 1) No-throw guarantee

Promise: the operation does not throw exceptions.
In C++ this is often expressed with `noexcept`. No-throw operations are critical building blocks for strong exception safety, performance optimizations, and predictable cleanup.

```
#include <utility>

struct S {
    S() = default;
    S(S&&) noexcept = default;
    S& operator=(S&&) noexcept = default;

    friend void swap(S& a, S& b) noexcept
```

```
    {
        using std::swap;
        swap(a.x, b.x);
    }

    int x{};
};
```

71.3.2 2) Strong guarantee (commit-or-rollback)

Promise: if the operation throws, *no observable state changes* occur (as if the operation never started). Objects remain exactly as they were before the call.
This guarantee is ideal for "transaction-like" operations and is often achievable using:

- copy-and-swap,
- build-then-commit,
- temporary objects and RAII,
- non-throwing swap/move operations.

71.3.3 3) Basic guarantee

Promise: if the operation throws, the program remains in a *valid state*:

- invariants are preserved,
- no resources are leaked,
- objects are destructible,
- but values may have changed (the operation may have partially completed).

The basic guarantee is the minimum acceptable guarantee for most library-quality code.

71.3.4 4) No guarantee

Promise: none. If the operation throws, state may be corrupted or invariants violated. This is generally unacceptable except in very constrained low-level code where failure is treated as fatal.

71.4 How `noexcept` affects guarantees

71.4.1 E.12: Use `noexcept` when throwing is impossible or unacceptable

`noexcept` is a contract:

- If a `noexcept` function throws, the program calls `std::terminate`.
- Therefore, only mark `noexcept` when you mean it.

71.4.2 E.16: Operations that must never fail

Destructors, deallocation, swap, and exception-type copy/move construction must never fail. This supports reliable cleanup and predictable unwinding.

71.5 Design patterns to achieve strong or basic guarantees

71.5.1 Pattern 1: Copy-and-swap (strong guarantee)

If copying a value and swapping are safe, an assignment can be strongly exception safe.

```
#include <utility>
#include <vector>

class VecBox {
```

```
public:
    VecBox() = default;

    void assign(std::vector<int> v) // take by value (copy/move happens before commit)
    {
        data_.swap(v); // commit; if swap is noexcept, strong guarantee
    }

private:
    std::vector<int> data_;
};
```

Explanation:

- Any exception happens while creating the parameter `v` (before touching `data_`).
- Then the commit step is `swap`, which is designed to be non-throwing for many standard containers.
- If an exception occurs before `swap`, the object is unchanged (strong guarantee).

71.5.2 Pattern 2: Build-then-commit (strong guarantee)

Construct new state in temporaries, then commit in a no-throw step.

```
#include <string>
#include <utility>

class NameList {
public:
    void set_name(std::string s) // by value
    {
        name_.swap(s);           // commit
```

```
    }

    const std::string& name() const noexcept { return name_; }

private:
    std::string name_;
};
```

71.5.3 Pattern 3: Maintain invariants and provide at least the basic guarantee

When strong guarantee is expensive or impossible, still preserve invariants and prevent leaks.

```
#include <vector>

class Counter {
public:
    void push_count(int x)
    {
        // If vector growth throws (allocation), vector remains valid.
        // Invariant: counts_ is a valid vector.
        counts_.push_back(x);
    }

private:
    std::vector<int> counts_;
};
```

71.6 Exception guarantees in standard library components (practical view)

The standard library generally aims for:

- **basic guarantee** widely,
- **strong guarantee** for many operations when allocation/element operations permit,
- **no-throw** for destructors and many swaps/moves where feasible.

However, the effective guarantee often depends on:

- whether element move/copy operations can throw,
- allocator behavior,
- whether operations allocate memory.

Therefore, when you design generic code, specify guarantees in terms of requirements such as "strong guarantee if `T` is nothrow-movable and swap is `noexcept`".

71.7 Guarantees for constructors and destructors

71.7.1 Constructors

A constructor should establish the invariant or throw (E.5). If it throws, the object is not created, and already-constructed subobjects are destroyed. This is an important form of safety: a partially constructed object never becomes usable.

71.7.2 Destructors

Destructors must not throw (E.16). If cleanup can fail (e.g., flushing a file), choose a strategy that does not throw from the destructor:

- provide an explicit `close()` or `flush()` that can report failure,
- store error status for later retrieval,
- log errors and continue,
- treat failure as fatal and terminate in a controlled way (project policy).

71.8 Specifying exception guarantees in API design

71.8.1 Document guarantees explicitly

For each public function, document:

- does it throw? under what conditions?
- if it throws, what remains valid and what may change?
- are invariants preserved?
- is it `noexcept`?

71.8.2 Use types to support guarantees

- RAII wrappers prevent leaks automatically.
- Value types and immutability often make strong guarantees easier.
- `noexcept` move and swap enable commit steps.

71.8.3 Prefer narrow, composable operations

Complex functions that do many steps are harder to make strongly exception safe. Break operations into stages:

- validate inputs,
- build new state,
- commit.

71.9 A worked example: strong guarantee for a mutating operation

Goal: insert an element while providing strong guarantee for the container state.

```
#include <vector>
#include <utility>

class IntSet {
public:
    void add(int x)
    {
        if (contains(x)) return;

        // Build new state first (may allocate and throw)
        std::vector<int> tmp = data_;
        tmp.push_back(x);

        // Commit (swap is non-throwing for std::vector in typical implementations)
        data_.swap(tmp);
    }
```

```
private:
    bool contains(int x) const
    {
        for (int v : data_) if (v == x) return true;
        return false;
    }

    std::vector<int> data_;
};
```

If allocation fails while building `tmp`, `data_` is unchanged (strong guarantee). This approach may cost extra copying; choose it when strong guarantee is worth the cost.

71.10 Common pitfalls that break guarantees

71.10.1 Throwing with “direct ownership” (E.13)

Throwing while holding raw owning pointers/handles risks leaks. Acquire into RAII immediately and never let ownership be manual across throwing code paths.

71.10.2 Catching too early and continuing in an unknown state

Catching exceptions without restoring invariants often breaks guarantees. Catch only when you can:

- recover to a known valid state,
- translate or log and rethrow,
- or terminate cleanly.

71.10.3 Throwing from destructors

This can terminate the program during unwinding and is forbidden by rule E.16.

71.10.4 Marking `noexcept` incorrectly

A wrong `noexcept` can turn a recoverable error into termination. Use `noexcept` only when throwing is truly impossible or unacceptable by design (E.12).

71.11 Checklist: selecting a guarantee

1. Can the operation be `noexcept`? If yes, prefer no-throw. (E.12)
2. If it can throw, can you provide the strong guarantee using build-then-commit or copy-and-swap?
3. If strong guarantee is too costly, can you provide the basic guarantee by preserving invariants and using RAII?
4. Are destructors and cleanup operations guaranteed not to throw? (E.16)
5. Are ownership and lifetimes always managed with RAII so unwinding cannot leak? (E.6)
6. Are invariants explicit and preserved across all failure paths? (E.4)

71.12 Summary

Exception guarantees are the contract for program state under failure:

- **No-throw guarantee:** operation never throws (`noexcept`).
- **Strong guarantee:** commit-or-rollback; no observable change if an exception is thrown.

- **Basic guarantee:** invariants preserved and no leaks, but state may change.
- **No guarantee:** generally unacceptable.

Design for guarantees using invariants (E.4), constructors that establish invariants or throw (E.5), RAII to prevent leaks (E.6), careful use of `noexcept` (E.12), and non-throwing cleanup/swap operations (E.16).

Chapter 72

No Exceptions from Destructors

72.1 Purpose and scope

This chapter explains why destructors must never let exceptions escape, how this rule interacts with **stack unwinding**, and how to design types that manage resources safely even when cleanup can fail.
This guidance is grounded in the C++ Core Guidelines, especially:

- E.12 (use `noexcept` when throwing is impossible or unacceptable),
- E.16 (destructors, deallocation, swap, and exception type copy/move construction must never fail),
- E.6 (use RAII to prevent leaks),
- E.4–E.5 (design around invariants; constructors establish invariants or throw).

72.2 The rule

Rule statement (E.16)

Destructors must never fail. In practice, a destructor must not throw (it should be `noexcept` and must not allow exceptions to escape).

72.3 Why throwing from a destructor is dangerous

72.3.1 1) Termination during stack unwinding

If an exception is already active and stack unwinding is in progress, and then a destructor throws a second exception, the C++ runtime **terminates the program**.
This is why allowing exceptions to propagate from destructors is considered fundamentally unsafe for robust systems.

72.3.2 2) Destructors run in many critical contexts

Destructors run:

- at normal scope exit,
- during exception propagation (unwinding),
- for partially constructed objects when a constructor throws,
- during destruction of static objects at program termination.

In these contexts, throwing is either unacceptable by design or can lead to termination.

72.3.3 3) Cleanup must be reliable

A destructor is part of the **resource safety contract** (RAII). If cleanup can throw, RAII can no longer be trusted, and resource leaks or corrupted state become much more likely.

72.4 What the language guarantees about destructors

72.4.1 Destructors are implicitly `noexcept`

In modern C++, destructors are implicitly treated as non-throwing unless explicitly declared otherwise. Regardless of syntax, your design goal should be:

- do not throw from destructors,
- do not call throwing operations from destructors unless you catch internally.

72.5 Practical rule: destructor code must be "failure-free"

72.5.1 Allowed destructor actions

A destructor may:

- release resources (close file, unlock mutex, free memory),
- perform non-throwing cleanup,
- swallow/record cleanup errors without throwing.

72.5.2 Forbidden destructor actions

A destructor must not:

- throw exceptions,
- call code that can throw unless exceptions are fully handled inside the destructor,
- depend on callers to remember cleanup steps (violates RAII spirit).

72.6 How to design with “cleanup can fail”

Some cleanup operations can fail in reality (e.g., flushing buffers, committing transactions, network shutdown, close with error reporting). The design pattern is:

72.6.1 Pattern A: Provide an explicit operation that can report failure

Use an explicit `close()`, `flush()`, `commit()`, or `finish()` member function that:

- can throw (exception-enabled builds), or
- returns an error code / status (no-exception builds).

The destructor then performs *best-effort* cleanup without throwing.

```
#include <cstdio>
#include <stdexcept>

class OutFile {
public:
    explicit OutFile(const char* path)
        : f_{std::fopen(path, "wb")}
    {
        if (!f_) throw std::runtime_error{"OutFile: open failed"};
    }

    // Explicit close that can report failure
    void close()
    {
        if (!f_) return;
        if (std::fclose(f_) != 0) {
            f_ = nullptr;
            throw std::runtime_error{"OutFile: close failed"};
```

```
        }
        f_ = nullptr;
    }

    ~OutFile() noexcept
    {
        // Destructors must not throw.
        // Best-effort cleanup only:
        if (f_) (void)std::fclose(f_);
    }

    OutFile(const OutFile&) = delete;
    OutFile& operator=(const OutFile&) = delete;

private:
    std::FILE* f_{nullptr};
};
```

72.6.2 Pattern B: Catch and handle exceptions inside the destructor

If a destructor must call potentially throwing code (often a design smell), catch everything and handle locally.

```
#include <exception>

struct Logger {
    ~Logger() noexcept
    {
        try {
            flush(); // may throw
        }
        catch (...) {
            // Never throw from destructor.
```

```
            // Policy options: log to a non-throwing sink, set a flag, or terminate.
        }
    }

    void flush(); // potentially throwing
};
```

Important. Catching in destructors should be rare. A better design is to ensure `flush()` is explicit and called before destruction if errors must be reported.

72.6.3 Pattern C: "Two-phase" completion (explicit finalize)

For operations where correctness requires confirming success (e.g., writing a file completely, committing a transaction), use a two-phase pattern:

- the destructor rolls back or closes best-effort,
- the user must call `commit()` / `finalize()` to obtain error reporting.

```
#include <stdexcept>

class Transaction {
public:
    void commit()
    {
        // perform commit; throw/report if it fails
        committed_ = true;
    }

    ~Transaction() noexcept
    {
        if (!committed_) {
            // rollback best-effort; must not throw
```

```
            rollback_noexcept();
        }
    }

private:
    void rollback_noexcept() noexcept
    {
        // non-throwing rollback
    }

    bool committed_{false};
};
```

72.7 Connection to RAII and invariants

72.7.1 E.6: RAII provides cleanup during exceptions

RAII ensures that when an exception leaves a scope, all local objects are destroyed, releasing resources safely. This works only if destructors do not throw.

72.7.2 E.5: constructors establish invariants or throw

A common and safe pattern is:

- constructors may throw to prevent invalid objects from existing,
- destructors must never throw to ensure cleanup is always safe.

72.8 Interaction with `noexcept` (E.12)

Mark destructors `noexcept` (explicitly or implicitly) and treat it as a hard rule:

- if a destructor throws, the program will terminate,
- therefore any failure reporting must happen outside the destructor.

72.9 Common mistakes and correct alternatives

72.9.1 Mistake 1: throwing on destructor failure

```
// BAD: destructor that throws
struct Bad {
    ~Bad() { throw 1; }
};
```

Fix: move failure reporting to an explicit function; destructor does best-effort cleanup.

72.9.2 Mistake 2: calling a throwing function from destructor without handling

```
// BAD: if flush() throws, destructor throws
struct BadLogger {
    ~BadLogger() noexcept(false) { flush(); }
    void flush();
};
```

Fix: make `flush()` explicit and require callers to handle errors, or catch internally and use a policy.

72.9.3 Mistake 3: hiding important failures

Swallowing exceptions silently can hide data loss (e.g., failed flush/close). Use a policy:

- explicit `close()/commit()` for reliable error reporting,

- on destructor cleanup failure, log to a non-throwing sink or store a status for later retrieval.

72.10 Checklist

1. Does every destructor guarantee no exceptions escape? (E.16)
2. Are destructors effectively `noexcept` and do they avoid throwing operations? (E.12)
3. If cleanup can fail, is there an explicit operation that reports failure (`close/flush/commit`)?
4. Are RAII wrappers used for all resources so unwinding cannot leak? (E.6)
5. Are invariants preserved in all failure paths (constructors throw if they cannot establish invariants)? (E.4, E.5)
6. If exceptions are caught in destructors, is the handling policy explicit and safe (no silent corruption)?

72.11 Summary

Destructors must never let exceptions escape:

- During stack unwinding, a throwing destructor can cause immediate program termination.
- Robust RAII depends on non-throwing destructors.
- If cleanup can fail, design an explicit operation (`close/flush/commit`) to report failure; destructors perform best-effort cleanup only.

- Use `noexcept` intentionally and treat destructor non-throwing behavior as a hard contract.

Chapter 73

Alternatives to Exceptions

73.1 Purpose and scope

Exceptions are an effective error-handling mechanism in C++ when used as intended (E.2, E.3), but not every project can use them. Some codebases disable exceptions for policy, ABI, footprint, or certification reasons (common in embedded and kernel-like environments). The C++ Core Guidelines explicitly address this situation:

- E.25: If you can't throw exceptions, simulate RAII for resource management.
- E.26: If you can't throw exceptions, consider failing fast.
- E.27: If you can't throw exceptions, use error codes systematically.
- E.28: Avoid error handling based on global state (e.g., `errno`).

This chapter presents practical alternatives to exceptions, how to choose between them, and how to preserve the key properties that exceptions normally provide:

- clear propagation of failure,

- resource safety (no leaks),
- preservation of invariants,
- readable, non-repetitive code.

73.2 First principle: alternatives must still be safe

Disabling exceptions does not remove the need for:

- invariants (E.4),
- constructors that establish invariants (or alternative construction patterns),
- RAII resource management (E.25),
- consistent error propagation (E.27).

The most common failure mode in no-exception code is not the lack of `throw`; it is **inconsistent handling** and **resource leaks**.

73.3 Alternative 1: Return status (`bool` / error enums) and out-parameters

73.3.1 When it fits

Use return status when:

- failure is expected and common (e.g., search miss),
- the error space is small and easy to encode,
- you can keep a strict, systematic convention across the project (E.27).

73.3.2 Example: status + out-parameter

```
#include <cstdint>

enum class errc : std::uint8_t { ok, invalid_input, io_error };

errc parse_port(const char* s, int& out_port) noexcept
{
    if (!s) return errc::invalid_input;

    int p = 0;
    for (const char* c = s; *c; ++c) {
        if (*c < '0' || *c > '9') return errc::invalid_input;
        p = p * 10 + (*c - '0');
        if (p > 65535) return errc::invalid_input;
    }

    out_port = p;
    return errc::ok;
}
```

73.3.3 Pros and cons

Pros: simple, fast, explicit.
Cons: easy to ignore return values; propagation can become repetitive and error-prone unless disciplined.

73.3.4 Guideline advice (E.27)

If you use error codes, do it systematically:

- define a single convention (return type, naming, and propagation),

- require checking (code review, tooling, `[[nodiscard]]`),
- avoid mixing unrelated styles.

73.4 Alternative 2: `std::optional` for "absence is not an error"

73.4.1 When it fits

Use `std::optional<T>` when:

- returning "no value" is a normal outcome,
- you do not need a detailed error reason,
- you want a value-based interface with clear absence semantics.

```
#include <optional>

std::optional<int> find_first_positive(const int* a, int n) noexcept
{
    if (!a || n < 0) return std::nullopt; // treat invalid inputs as "no result" by policy
    for (int i = 0; i < n; ++i) {
        if (a[i] > 0) return a[i];
    }
    return std::nullopt;
}
```

73.4.2 Important limitation

`optional` cannot carry an error message. If you need an error reason, use a result type.

73.5 Alternative 3: Return a “result” type (value-or-error)

73.5.1 When it fits

Use a dedicated result type when:

- you need to return either a value or an error,
- you want explicit propagation without exceptions,
- you want consistent project-wide handling (E.27).

73.5.2 Minimal result type example

```
#include <utility>
#include <string>

enum class errc { ok, io_error, parse_error };

template<class T>
class result {
public:
    static result success(T v) { return result{std::move(v), errc::ok, {}}; }
    static result failure(errc e, std::string msg = {}) { return result{T{}, e,
    ↪  std::move(msg)}; }

    bool ok() const noexcept { return ec_ == errc::ok; }
    errc error() const noexcept { return ec_; }
    const std::string& message() const noexcept { return msg_; }

    T& value() & { return val_; }
    const T& value() const& { return val_; }
```

```
private:
    result(T v, errc e, std::string msg) : val_{std::move(v)}, ec_{e}, msg_{std::move(msg)}
    ↪ {}

    T val_{};
    errc ec_{errc::ok};
    std::string msg_{};
};

result<int> parse_int(const char* s) noexcept
{
    if (!s) return result<int>::failure(errc::parse_error, "null input");
    int v = 0;
    bool any = false;
    for (const char* c = s; *c; ++c) {
        if (*c < '0' || *c > '9') return result<int>::failure(errc::parse_error,
        ↪ "non-digit");
        any = true;
        v = v * 10 + (*c - '0');
    }
    if (!any) return result<int>::failure(errc::parse_error, "empty");
    return result<int>::success(v);
}
```

73.5.3 Discipline requirement

A result type is only effective if:

- every caller checks `ok()` (or equivalent),
- the project agrees on propagation style,
- ignoring results is prevented (consider `[[nodiscard]]` on result types).

73.6 Alternative 4: Fail fast (terminate) when recovery is unsafe or impossible

73.6.1 When it fits (E.26)

Fail fast is appropriate when:

- continuing would be unsafe (corrupt state, violated invariants),
- there is no meaningful recovery strategy,
- exceptions are disabled and error-code discipline is unreliable.

Fail fast is not "error handling" in the recovery sense; it is a policy decision: *do not continue in an unknown state.*

73.6.2 Example: fail fast on contract violation

```
#include <cstdlib>

[[noreturn]] void fail_fast() noexcept
{
    std::abort();
}

int at(const int* a, int n, int i) noexcept
{
    if (!a || n < 0 || i < 0 || i >= n) fail_fast();
    return a[i];
}
```

73.7 Alternative 5: Error propagation boundaries (translation layers)

Even in exception-enabled projects, you often translate errors at boundaries:

- C ABI boundaries (cannot let exceptions cross),
- OS callbacks,
- plugin boundaries,
- thread entry functions.

In a no-exception environment, design explicit boundary functions that:

- convert internal errors to a stable outward representation (error codes, status structs),
- centralize logging/context augmentation.

73.8 Alternative 6: Using `errno` and global state (generally avoid) (E.28)

73.8.1 Guideline position

Avoid error handling based on global state such as `errno`.

73.8.2 If you must use C APIs that set `errno`

Translate immediately to a local error object:

- capture `errno` into a value,

- return a result/error code based on it,
- do not let global state be the main interface.

```
#include <cerrno>
#include <cstdio>

enum class errc { ok, io_error };

struct io_status {
    errc ec;
    int sys_errno;
};

io_status write_raw(const char* path) noexcept
{
    std::FILE* f = std::fopen(path, "wb");
    if (!f) {
        return {errc::io_error, errno};
    }
    std::fclose(f);
    return {errc::ok, 0};
}
```

73.9 Resource safety without exceptions (E.25)

73.9.1 RAII still works

RAII is not dependent on exceptions; it also protects against early returns and multiple error exits. Even in a no-exception codebase, RAII should be the default resource management technique.

73.9.2 Example: RAII + error codes

```
#include <cstdio>

class File {
public:
    File(const char* path, const char* mode) noexcept
        : f_{std::fopen(path, mode)} {}

    ~File() noexcept { if (f_) std::fclose(f_); }

    std::FILE* get() const noexcept { return f_; }
    explicit operator bool() const noexcept { return f_ != nullptr; }

private:
    std::FILE* f_{nullptr};
};

enum class errc { ok, io_error };

errc process_file(const char* path) noexcept
{
    File f{path, "rb"};
    if (!f) return errc::io_error;

    // work; return on errors as needed
    return errc::ok;
}
```

73.10 How to choose an alternative: decision guide

73.10.1 If failure is normal and expected

Use:

- `std::optional` when you only need “present/absent”,
- status codes or error enums when a small reason set is sufficient.

73.10.2 If you need value *or* a reason

Use a project-wide result type (value-or-error).

73.10.3 If recovery is not safe

Fail fast (E.26).

73.10.4 If you are wrapping C APIs

Translate global-state failures immediately (E.28), and expose explicit error objects.

73.11 How to keep alternative approaches maintainable

73.11.1 Enforce checking

- Use `[[nodiscard]]` on result/status types.
- Use code review and static analysis to enforce propagation rules.
- Avoid designs where error checking is optional.

73.11.2 Centralize error representation

Define:

- a small error enum or category system,
- a consistent status/result structure,
- a single policy for adding context (messages, codes).

73.11.3 Avoid mixing styles casually

Mixing "sometimes throw" with "sometimes return codes" without clear boundaries causes unpredictable behavior. If you must mix, define strict translation boundaries.

73.12 Checklist

1. Is there a documented, project-wide error strategy? (E.1)
2. If exceptions are disabled, is RAII still used everywhere? (E.25)
3. Are error codes/results used systematically with a single convention? (E.27)
4. Are return values enforced as checked (e.g., `[[nodiscard]]`)?
5. Are "absence" outcomes modeled with `optional` rather than exceptions?
6. Is fail fast used intentionally when recovery is unsafe? (E.26)
7. Are C-style global-state errors (`errno`) translated immediately? (E.28)

73.13 Summary

Alternatives to exceptions can be safe and maintainable if they preserve the core properties that exceptions normally provide:

- use RAII for resources even without exceptions (E.25),
- use error codes systematically and consistently (E.27),
- consider failing fast when recovery is unsafe (E.26),
- avoid global-state error handling such as `errno` as a primary strategy (E.28),
- model normal outcomes with return-based types (`optional`, status enums, result types),
- design clear boundaries where errors are translated and logged.

Part XII

Concurrency and Parallelism (CP)

Chapter 74

CP.1 – CP.61: Concurrency and Parallelism (CP)

74.1 Scope and intent

This chapter summarizes and explains the **Concurrency and Parallelism (CP)** rules of the *ISO C++ Core Guidelines*. The goal is to help you write C++ code that remains correct when used in multi-threaded programs, and to use standard C++ concurrency facilities safely and efficiently.

74.1.1 What this chapter covers (and what it does not)

- It covers the CP rules that are currently defined in the Core Guidelines within the numeric range **CP.1–CP.61**.
- Some CP numbers in this range are **not defined** by the Core Guidelines (reserved or absent). This chapter explicitly marks such gaps to remain faithful to the original rule set.

- It focuses on correctness first (data races, deadlocks, lifetime), then performance (contention, context switching, thread creation).

- It does not attempt to replace dedicated concurrency textbooks; instead it gives practical engineering guidance and safe patterns.

74.2 Essential definitions (precision matters)

74.2.1 Concurrency vs. parallelism

- **Concurrency** means multiple tasks are in progress during overlapping time periods. They may or may not run simultaneously.

- **Parallelism** means multiple computations run *at the same time* (e.g., on multiple cores or vector units).

74.2.2 Data race (the one rule you cannot negotiate)

A **data race** occurs when:

- two or more threads access the same memory location concurrently, and

- at least one access is a write, and

- there is no synchronization that establishes a **happens-before** relationship.

A data race in C++ yields **undefined behavior**. This is not "just a bug"; the optimizer may assume it never happens and transform code in surprising ways.

74.2.3 Happens-before (why locks and atomics exist)

A **happens-before** relationship is the formal ordering that makes one thread's writes visible to another thread in a defined way. Common ways to establish it:

- Locking and unlocking the same mutex around shared data accesses.
- Atomic operations with appropriate memory ordering.
- Thread start/join relationships (synchronization at thread completion).
- Condition variables with correct locking discipline.

74.3 Rule map (CP.1–CP.61)

74.3.1 Defined rules in this numeric range

- **General:** CP.1, CP.2, CP.3, CP.4, CP.8, CP.9
- **Concurrency (shared-state):** CP.20, CP.21, CP.22, CP.23, CP.24, CP.25, CP.26, CP.31, CP.32, CP.40, CP.41, CP.42, CP.43, CP.44, CP.50
- **Coroutines:** CP.51, CP.52, CP.53
- **Message passing / task results:** CP.60, CP.61

74.3.2 CP numbers not defined by the Core Guidelines (in CP.1–CP.61)

The following rule numbers in the range **CP.1–CP.61** are not currently defined as CP rules in the Core Guidelines:

Not defined	CP.5–CP.7, CP.10–CP.19, CP.27–CP.30, CP.33–CP.39, CP.45–CP.49, CP.54–CP.59

When these numbers are absent, you should not invent "rules" for them in a formal rule reference manual; treat them as **reserved/unused** in the original document.

74.4 CP.1 – CP.9: Baseline rules for all C++ code

74.4.1 CP.1: Assume that your code will run as part of a multi-threaded program

Rule. Write code so it remains correct even if called concurrently, especially library code.
Reason. Code gets reused. Today's single-threaded utility becomes tomorrow's shared library.
Common pitfall: shared function-local statics used as a cache.
Bad example (shared mutable state without synchronization):

```
double cached_computation(int x)
{
    static int cached_x = 0;
    static double cached_result = 0.0;

    if (cached_x != x) {
        cached_x = x;
        cached_result = expensive(x);
    }
    return cached_result; // data race if called concurrently
}
```

Good example (make caching an explicit object; caller chooses sharing model):

```
struct ComputationCache {
    int cached_x = 0;
    double cached_result = 0.0;
```

```
    double compute(int x) {
        if (cached_x != x) {
            cached_x = x;
            cached_result = expensive(x);
        }
        return cached_result;
    }
};

// Each thread can have its own cache instance, or share one with proper locking.
```

Notes.

- If you must have per-thread caching, consider `thread_local` state, but document it clearly.
- If you must share, encapsulate the lock with the data (see CP.50).

74.4.2 CP.2: Avoid data races

Rule. Ensure that no two threads access the same object concurrently when at least one access writes, unless you use correct synchronization.
Reason. Data races are undefined behavior; correctness collapses.
Bad example (non-atomic increment of shared state):

```
int get_id()
{
    static int id = 0;
    return ++id; // data race if called concurrently
}
```

Better (use an atomic for the shared counter):

```
#include <atomic>

int get_id()
{
    static std::atomic<int> id{0};
    return id.fetch_add(1, std::memory_order_relaxed) + 1;
}
```

Important note. Atomics solve *atomicity and ordering* for the atomic object itself. They do not automatically make surrounding non-atomic data safe.

74.4.3 CP.3: Minimize explicit sharing of writable data

Rule. Prefer designs where threads do not share writable state.
Reason. If you do not share writable data, you cannot have a data race on that data. Less sharing also typically means less locking and better performance.
Preferred approaches.

- Share **immutable** data freely (const data, persistent structures).
- Prefer **task decomposition** where each task owns its working set.
- Pass data by **value** or **move** rather than by shared references (see CP.31).

Example (share read-only input; compute independent results):

```
auto f1 = std::async([&] { return compute_a(read_only_input); });
auto f2 = std::async([&] { return compute_b(read_only_input); });

auto r1 = f1.get();
auto r2 = f2.get();
```

74.4.4 CP.4: Think in terms of tasks, rather than threads

Rule. Express concurrency in terms of **tasks** (units of work), not raw threads.
Reason. Threads are a low-level execution resource. Tasks match application intent and are easier to reason about. Task-based designs also map better to thread pools and work-stealing schedulers.
Guidance.

- Prefer task abstractions (`std::async`, thread pools, parallel algorithms) over manual thread management.
- Treat a task like a function call that may run concurrently.

74.4.5 CP.8: Don't try to use `volatile` for synchronization

Rule. Do not use `volatile` to make multi-threaded code "safe".
Reason. In ISO C++, `volatile` is about special memory (e.g., memory-mapped I/O) and preventing certain optimizations on accesses to that object. It does **not** establish inter-thread ordering, mutual exclusion, or happens-before.
Bad example (volatile does not fix data races):

```
volatile int free_slots = 100;

int acquire_slot()
{
    return free_slots--; // still a data race; volatile does not help
}
```

Better (use atomic or a lock):

```
std::atomic<int> free_slots{100};
```

```
int acquire_slot()
{
    int old = free_slots.fetch_sub(1, std::memory_order_relaxed);
    return old - 1;
}
```

74.4.6 CP.9: Whenever feasible use tools to validate your concurrent code

Rule. Use dynamic analysis and testing tools designed to detect concurrency bugs.
Reason. Concurrency bugs are often timing-dependent and may not reproduce reliably. Specialized tools can detect data races, deadlocks, and other hazards that normal testing misses.
Practical tool categories (non-exhaustive).

- Data race detectors (e.g., thread sanitizers).
- Deadlock detection (runtime diagnostics, lock-order checking).
- Stress testing with randomized scheduling or high load.

74.5 CP.20 – CP.50: Shared-state concurrency rules (CP.con)

This group covers the safe use of locks, threads, and shared ownership.

74.5.1 CP.20: Use RAII, never plain `lock()`/`unlock()`

Rule. Acquire locks using RAII types (`std::lock_guard`, `std::unique_lock`, `std::scoped_lock`) rather than manual `lock()` and `unlock()`.
Reason. Prevents leaked locks on exceptions, early returns, and maintenance edits.
Bad:

```
std::mutex m;

void f()
{
    m.lock();
    do_work();
    if (error()) return;   // lock leak: undefined behavior later
    m.unlock();
}
```

Good:

```
std::mutex m;

void f()
{
    std::lock_guard<std::mutex> lock(m);
    do_work();
    if (error()) return;   // safe: lock released automatically
}
```

74.5.2 CP.21: Use `std::lock()` or `std::scoped_lock` to acquire multiple mutexes

Rule. When locking multiple mutexes, use deadlock-avoiding facilities.

Reason. Locking multiple mutexes in inconsistent order is a classic deadlock source.

Good (C++17 and later):

```
std::mutex m1, m2;

void f()
```

```
{
    std::scoped_lock lock(m1, m2); // deadlock-safe acquisition
    // use protected resources
}
```

Alternative (C++11/14):

```
std::mutex m1, m2;

void f()
{
    std::lock(m1, m2);
    std::lock_guard<std::mutex> l1(m1, std::adopt_lock);
    std::lock_guard<std::mutex> l2(m2, std::adopt_lock);
    // use protected resources
}
```

74.5.3 CP.22: Never call unknown code while holding a lock (e.g., a callback)

Rule. Do not invoke callbacks, virtual functions, or any "unknown" code while holding a mutex.

Reason. Unknown code may try to acquire the same lock (or another lock in a conflicting order), causing deadlock or severe contention.

Bad:

```
std::mutex m;
std::vector<int> data;

void for_each_and_call(const std::function<void(int)>& cb)
{
    std::lock_guard<std::mutex> lock(m);
```

```
    for (int x : data) {
        cb(x); // unknown code while holding lock
    }
}
```

Good (copy data under lock, then call outside):

```
std::mutex m;
std::vector<int> data;

void for_each_and_call(const std::function<void(int)>& cb)
{
    std::vector<int> snapshot;
    {
        std::lock_guard<std::mutex> lock(m);
        snapshot = data; // copy while protected
    }
    for (int x : snapshot) {
        cb(x); // safe: no lock held
    }
}
```

74.5.4 CP.23: Think of a joining thread as a scoped container

Rule. A thread that is always joined acts like a scoped object: it must finish before scope exit.
Reason. This matches RAII thinking and prevents background execution from outliving required resources.
Modern C++ guidance. In C++20 and later, prefer `std::jthread` because it joins automatically on destruction.

```
#include <thread>
```

```
void work();

void f()
{
    std::jthread t(work); // joins automatically in destructor
    // ...
} // safe: join on scope exit
```

74.5.5 CP.24: Think of a thread as a global container

Rule. A thread that can outlive the creating scope behaves like a global object: it can observe destroyed state unless you design ownership and lifetime carefully.
Reason. Once a thread outlives the scope that created it, it can easily use dangling references or invalid pointers.
Engineering implication. If a thread may outlive the creating scope, you must:

- use explicit shared ownership (`shared_ptr`) for heap objects that must remain alive (see CP.32), and
- define clean shutdown (stop tokens, message passing, or joining discipline).

74.5.6 CP.25: Prefer a joining thread over `std::thread`

Rule. Prefer a thread type that joins in its destructor (Core Guidelines historically refer to `gsl::joining_thread`).
Up-to-date C++ note. In standard C++20 and later, `std::jthread` directly provides this RAII joining behavior.
Example:

```
#include <thread>

void worker();

void run()
{
    std::jthread t(worker); // RAII join
    // ...
}
```

74.5.7 CP.26: Don't `detach()` a thread

Rule. Avoid `detach()`.
Reason. Detached threads are hard to control: lifetime, shutdown, error propagation, resource cleanup, and process exit behavior become fragile. They commonly cause use-after-free and shutdown races.
If you think you need detach. Prefer one of:

- a long-lived service thread with explicit ownership and a clear stop protocol,
- a thread pool,
- task-based concurrency (`std::async` or a scheduler),
- message passing with explicit shutdown.

74.5.8 CP.31: Pass small amounts of data between threads by value

Rule. Prefer passing small data by value, rather than by reference/pointer, when transferring data to another thread or task.
Reason. Values avoid shared mutable state and lifetime bugs.

Bad (dangling reference risk):

```
void start()
{
    std::string s = "job";
    std::thread t([&] { use(s); }); // may outlive scope: dangling reference
    t.detach();
}
```

Good (copy or move the value into the task):

```
void start()
{
    std::string s = "job";
    std::jthread t([s = std::move(s)] { use(s); }); // safe ownership
}
```

74.5.9 CP.32: To share ownership between unrelated threads use `shared_ptr`

Rule. If multiple threads need to share ownership of a heap object and their lifetimes are not nested, use `std::shared_ptr` (or an equivalent shared-ownership mechanism).
Reason. It provides safe, automatic deletion when the last owner releases the object.
Example:

```
#include <memory>
#include <thread>

struct State { /*...*/ };

void worker(std::shared_ptr<State> st);
```

```
void run()
{
    auto st = std::make_shared<State>();
    std::jthread t1(worker, st);
    std::jthread t2(worker, st);
} // safe: State lives until both threads finish and release their shared_ptr
```

Important note. `shared_ptr` manages lifetime, not thread-safety of the pointed-to object. You may still need locks/atomics for the object's internal state.

74.5.10 CP.40: Minimize context switching

Rule. Avoid designs that force frequent context switches.

Reason. Context switches add overhead (scheduler work, cache disruption), and can destroy scalability.

Practical guidance.

- Prefer batching work to reduce lock/unlock and wake/sleep churn.
- Avoid excessive fine-grained task splitting when overhead dominates.
- Use wait-free reads where safe (immutable snapshots), rather than constantly contended locks.

74.5.11 CP.41: Minimize thread creation and destruction

Rule. Avoid creating a new thread per small unit of work.

Reason. Thread creation is expensive and may exhaust resources.

Bad (thread-per-message):

```
void dispatcher(std::istream& is)
{
    for (Message m; is >> m; ) {
        std::thread(worker, m).detach(); // expensive and unsafe shutdown
    }
}
```

Better (queue + fixed worker threads, or a thread pool):

```
Sync_queue<Message> q;

void dispatcher(std::istream& is)
{
    for (Message m; is >> m; ) {
        q.put(m);
    }
}

void worker()
{
    for (;;) {
        Message m = q.get();
        process(m);
    }
}
```

74.5.12 CP.42: Don't `wait` without a condition

Rule. Do not wait on a condition variable without checking a predicate.

Reason. Spurious wakeups are allowed, and missed notifications can occur if you do not use the correct predicate-based pattern.

Correct pattern:

```
std::mutex m;
std::condition_variable cv;
bool ready = false;

void consumer()
{
    std::unique_lock<std::mutex> lock(m);
    cv.wait(lock, [&]{ return ready; }); // predicate protects correctness
    use_resource();
}
```

74.5.13 CP.43: Minimize time spent in a critical section

Rule. Keep locked regions as small as possible.

Reason. Long critical sections increase contention, reduce parallelism, and raise tail latency.

Bad (lock held during slow work):

```
void f()
{
    std::lock_guard<std::mutex> lock(m);
    update_shared_state();
    slow_io(); // do not hold lock across slow I/O
}
```

Good (separate protected update from slow work):

```
void f()
{
    {
```

```
        std::lock_guard<std::mutex> lock(m);
        update_shared_state();
    }
    slow_io();
}
```

74.5.14 CP.44: Remember to name your `lock_guard`s and `unique_lock`s

Rule. Give lock objects a name.

Reason. A named lock makes scope explicit and prevents accidental immediate destruction (and thus accidental unlock).

Bad (temporary guard unlocks immediately):

```
std::lock_guard<std::mutex>(m); // locked and immediately unlocked
critical_update();              // not protected
```

Good:

```
std::lock_guard<std::mutex> lock(m);
critical_update(); // protected
```

74.5.15 CP.50: Define a mutex together with the data it guards. Use `synchronized_value<T>` where possible

Rule. Encapsulate the mutex and the protected data in the same type.

Reason. This reduces the chance of accessing the data without holding the correct lock, and makes invariants easier to maintain.

Good (encapsulation):

```
#include <mutex>
#include <vector>
```

```
class ThreadSafeQueue {
public:
    void push(int x)
    {
        std::lock_guard<std::mutex> lock(m_);
        q_.push_back(x);
    }

    bool try_pop(int& out)
    {
        std::lock_guard<std::mutex> lock(m_);
        if (q_.empty()) return false;
        out = q_.back();
        q_.pop_back();
        return true;
    }

private:
    std::mutex m_;
    std::vector<int> q_;
};
```

Note. The Guidelines Support Library (GSL) provides higher-level helpers in some environments (e.g., a `synchronized_value<T>` pattern). If you use such a facility, treat it as a strict improvement only if its semantics and performance properties are well understood and well tested in your toolchain.

74.6 CP.51 – CP.53: Coroutines (CP.coro)

These rules apply when using C++ coroutines (language support introduced in C++20, with libraries evolving across C++20/23/26).

74.6.1 CP.51: Do not use capturing lambdas that are coroutines

Rule. Avoid coroutine lambdas that capture state.
Reason. Coroutine frames can outlive the scope where captures were created, making lifetime and ownership easy to get wrong. Captures can also produce surprising object lifetimes and dangling references.
Preferred approach.

- Use explicit coroutine types/functions with well-defined parameter ownership.
- Pass required state by value (copy/move) into the coroutine.

74.6.2 CP.52: Do not hold locks or other synchronization primitives across suspension points

Rule. Never suspend a coroutine while holding a mutex (or similar synchronization resource).
Reason. Suspending while holding a lock can freeze progress for an unbounded time, causing deadlocks, priority inversions, and severe contention.
Engineering rule of thumb.

- Acquire lock, update shared state, release lock.
- Only then `co_await` (or otherwise suspend).

74.6.3 CP.53: Parameters to coroutines should not be passed by reference

Rule. Do not pass coroutine parameters by reference (including output parameters).
Reason. A coroutine may resume after the caller's referenced objects have gone out of scope. References become dangling silently.
Preferred approach.

- Pass parameters by value and move when appropriate.
- Return results as values (or via futures, see CP.60).

74.7 CP.60 – CP.61: Message passing and task results (CP.mess)

74.7.1 CP.60: Use a `future` to return a value from a concurrent task

Rule. Use `std::future` to represent the result of asynchronous work, including error propagation via exceptions.
Reason. A future preserves normal "return a value or throw" semantics without manual locking. It also makes the synchronization point explicit at `get()`.
Example:

```
#include <future>
#include <string>

int compute();

void f()
{
    std::future<int> fut = std::async(std::launch::async, compute);
```

```
    // do other work...
    int value = fut.get(); // waits and retrieves or rethrows
    use(value);
}
```

Note. Always understand the launch behavior you rely on. If you require concurrency, specify `std::launch::async`; otherwise an implementation may choose deferred execution.

74.7.2 CP.61: Use `async()` to spawn concurrent tasks

Rule. Prefer spawning concurrent tasks with `std::async` rather than managing raw `std::thread` and `std::promise` directly.

Reason. `std::async` packages execution + result delivery, and reduces error-prone manual thread lifecycle management.

Example:

```
#include <future>
#include <fstream>
#include <iostream>
#include <string>

int read_value(const std::string& filename)
{
    std::ifstream in(filename);
    in.exceptions(std::ifstream::failbit | std::ifstream::badbit);

    int value = 0;
    in >> value;
    return value;
}
```

```
void async_example()
{
    try {
        auto f1 = std::async(std::launch::async, read_value, "v1.txt");
        auto f2 = std::async(std::launch::async, read_value, "v2.txt");

        std::cout << (f1.get() + f2.get()) << '\n';
    } catch (const std::ios_base::failure&) {
        // handle I/O error
    }
}
```

Important limitations (engineering reality).

- `std::async` does not guarantee a thread pool; it may create threads aggressively and can fail by resource exhaustion.
- For high-throughput systems, a well-designed thread pool or scheduler is often preferable, but keep the `future`-based interface idea.

74.8 Practical enforcement checklist (what to review in code)

- **Race safety:** Any shared writable object must have a clear synchronization strategy (mutex, atomic, message passing, or confinement).
- **Ownership/lifetime:** No thread or task may access an object unless its lifetime is guaranteed (value passing, `shared_ptr`, joining discipline).
- **Lock discipline:** RAII locks only; no manual `lock/unlock`; no callbacks while holding locks; minimal critical sections.

- **Condition variables:** Always wait with a predicate; ensure the predicate is protected by the same mutex.
- **Thread management:** Avoid detach; prefer `std::jthread` (C++20+) or an equivalent joining thread type.
- **Performance hygiene:** Watch contention, context switching, and thread churn; prefer batching and pools where appropriate.
- **Tooling:** Use concurrency analyzers and stress testing as part of CI for multi-threaded components.

74.9 Common misconceptions corrected

74.9.1 "If it works in tests, it is thread-safe"

False. Many concurrency bugs are schedule-dependent and may disappear under debugging, logging, or different timing.

74.9.2 "Using `shared_ptr` makes my object thread-safe"

False. `shared_ptr` makes *lifetime* safe. The pointed-to object may still need locking/atomics.

74.9.3 "`volatile` fixes reordering and visibility"

False in ISO C++ concurrency. Use atomics and mutexes to express ordering and synchronization.

Chapter 75

Data Races

75.1 Purpose

A **data race** is one of the most destructive classes of concurrency bugs in C++. In the C++ memory model, a data race does not merely produce a wrong result: it makes the program's behavior **undefined**. The goal of this chapter is to define data races precisely, explain why they happen, show common real-world patterns that trigger them, and present reliable ways to prevent them using standard C++ facilities and Core Guidelines rules.

75.2 Precise definition (C++ memory model)

75.2.1 Conflicting actions and potential concurrency

Two evaluations (actions) **conflict** if they access the same memory location and at least one of them is a write. If those conflicting actions are **potentially concurrent** (for example, performed by different threads) and there is **no happens-before relationship** between them, then the execution contains a **data race**. In C++, the presence of such a data race results in

undefined behavior. This is the central rule to remember.

75.2.2 What counts as “the same memory location”

At the abstract-machine level, data races are defined per **memory location**. Practically:

- Two threads writing different *fields* of the same object may still race if the implementation treats those fields as overlapping storage or if they share the same scalar object.
- Reads/writes to the same scalar object (e.g., `int`, pointer, `bool`) from different threads without synchronization are the most common case.
- Accesses to different elements of a container can be safe or unsafe depending on the container and the operation; you must follow the container’s thread-safety rules.

75.2.3 Why “undefined behavior” is stronger than “unpredictable”

Undefined behavior means the C++ standard imposes **no requirements** on what the program does. The compiler may assume the data race never happens and optimize accordingly. Therefore:

- You cannot safely “detect” a data race by observing output.
- You cannot “handle” it with `try/catch`.
- You cannot “fix” it with `volatile`.

The correct response to a data race is to **eliminate it by design**.

75.3 Data race vs. race condition

75.3.1 Race condition (general)

A **race condition** is broader: the program's correctness depends on timing or interleaving. You can have a race condition without data races (for example, all shared accesses are atomic but the algorithm is logically wrong).

75.3.2 Data race (C++ formal meaning)

A **data race** is a specific violation of the C++ memory model: conflicting non-atomic accesses without happens-before. Every data race is a serious bug; not every race condition is a data race.

75.4 The Core Guidelines position

75.4.1 CP.2: Avoid data races

The C++ Core Guidelines explicitly require avoiding data races and recommend minimizing shared writable state, preferring tasks over raw threads, and using proper synchronization primitives rather than `volatile`.

75.5 How data races happen in real code

75.5.1 Shared mutable state without a lock or atomic

The most common cause is a shared variable that is read and written from multiple threads without synchronization.

75.5.2 Publishing an object without synchronization (unsafe publication)

A thread constructs or modifies an object and makes a pointer/reference to it visible to other threads without a synchronization step (lock, atomic release/acquire, etc.). Other threads may observe partially initialized state.

75.5.3 Lifetime bugs masquerading as data races

Sometimes the underlying issue is not merely concurrent access but **dangling references** (use-after-free) due to detached threads, coroutine suspension, or escaping references. These often appear in the same codebases as data races and must be fixed with ownership and lifetime discipline.

75.6 Canonical examples

75.6.1 Example 1: Non-atomic increment

Problem. `++x` is a read-modify-write sequence. Without synchronization, two threads can conflict.

```
#include <thread>
#include <vector>
#include <iostream>

int x = 0; // shared

void worker()
{
    for (int i = 0; i < 1'000'000; ++i) {
        ++x; // data race: unsynchronized conflicting accesses
```

```
    }
}

int main()
{
    std::thread t1(worker);
    std::thread t2(worker);
    t1.join();
    t2.join();
    std::cout << x << '\n';
}
```

Why it is a data race. Two threads write the same memory location (`x`) concurrently with no happens-before relationship. This triggers undefined behavior.

75.6.2 Example 2: "Read-only" that is not actually read-only

A function looks like it only reads a global, but another thread writes it.

```
#include <thread>
#include <string>

std::string mode = "fast"; // shared mutable state

void reader()
{
    // data race if writer updates mode concurrently
    if (mode == "fast") { /* ... */ }
}
```

```
void writer()
{
    mode = "safe"; // write concurrent with reader
}
```

Key insight. “I only read” does not matter if someone else writes concurrently.

75.6.3 Example 3: Volatile does not fix it

```
#include <thread>

volatile int ready = 0;
int data = 0;

void producer()
{
    data = 42;    // write
    ready = 1;    // volatile write does not establish inter-thread ordering in C++
}

void consumer()
{
    while (ready == 0) { } // volatile read
    int x = data;          // still a data race / not safely synchronized
}
```

Why it is wrong. `volatile` does not create the required happens-before relationship for thread synchronization in ISO C++.

75.7 How to prevent data races

75.7.1 Strategy A: Eliminate sharing (preferred)

The most robust approach is to avoid shared writable state:

- Give each thread/task its own state (thread confinement).
- Share immutable data (const objects, read-only snapshots).
- Use message passing: send work and results instead of sharing structures.

75.7.2 Strategy B: Protect shared state with mutexes

75.7.2.1 The fundamental pattern

If multiple threads access the same object and at least one access writes, protect all accesses with the same mutex.

```
#include <mutex>
#include <string>

class Config {
public:
    void set_mode(std::string m)
    {
        std::lock_guard<std::mutex> lock(mx_);
        mode_ = std::move(m);
    }

    std::string get_mode() const
```

```
    {
        std::lock_guard<std::mutex> lock(mx_);
        return mode_;
    }

private:
    mutable std::mutex mx_;
    std::string mode_;
};
```

75.7.2.2 RAII locking is mandatory

Always acquire locks via RAII (`std::lock_guard`, `std::unique_lock`, `std::scoped_lock`) to avoid leaked locks and to keep correctness stable under maintenance.

75.7.2.3 Rule: keep the mutex with the data

Encapsulate the mutex and the guarded data in the same type to reduce accidental unprotected access. This aligns with the Core Guidelines concurrency design approach.

75.7.3 Strategy C: Use atomics for atomic objects and simple coordination

75.7.3.1 Atomicity and ordering

`std::atomic<T>` provides atomic operations for the atomic object and can establish ordering (happens-before) when used with the correct memory order.

Example: atomic counter (no data race on the counter itself).

```
#include <atomic>

std::atomic<int> counter{0};
```

```
int next_id()
{
    return counter.fetch_add(1, std::memory_order_relaxed) + 1;
}
```

75.7.3.2 Release-acquire for safe publication (conceptual pattern)

A common safe publication technique is:

- Producer writes data, then performs an atomic store with `memory_order_release`.
- Consumer performs an atomic load with `memory_order_acquire`, then reads the data.

This creates a happens-before relationship between the producer's writes (before release) and the consumer's reads (after acquire), preventing data races *when correctly applied*.

75.7.3.3 Important limitation

Atomics do not automatically make a whole object graph thread-safe. They only guarantee atomicity and ordering for the atomic operations themselves. Protect compound invariants with mutexes or redesign to avoid sharing.

75.7.4 Strategy D: Use futures and tasks to avoid shared writable state

Instead of sharing writable state, prefer task-based concurrency that returns values:

- `std::async` + `std::future` to compute results asynchronously.
- Parallel algorithms (when appropriate) that operate on independent ranges.

This reduces the need for shared mutable data and therefore reduces data-race risk.

75.8 Common misconceptions and their corrections

75.8.1 `const` makes it thread-safe

False. `const` only prevents modification through that reference; the object may still be modified elsewhere. Also, internal mutation (`mutable`) can still race.

75.8.2 "It is just one machine instruction"

False. Even if a load or store maps to one instruction on a given CPU, without the C++ happens-before guarantees you still have a data race and undefined behavior. The language rules define correctness.

75.8.3 "Data races only cause wrong values"

False. They can cause crashes, infinite loops, corrupted invariants, and optimizer-induced transformations that make debugging misleading. Formally, the behavior is undefined.

75.9 Detecting data races (practice)

75.9.1 Use dynamic analysis tools

Use data-race detectors (thread sanitizers) and stress tests. These tools do not prove the absence of races, but they are extremely effective at finding real bugs.

75.9.2 Write concurrency-focused tests

- Run tests under high load and on different core counts.
- Prefer long-running stress tests for critical concurrency components.

- Test shutdown paths (they are frequent sources of races).

75.10 Summary

- A data race in C++ is **conflicting potentially concurrent accesses** with **no happens-before**, where at least one access is non-atomic.
- Any data race causes **undefined behavior**; you must eliminate it.
- The safest approach is to **avoid shared writable state** (immutability, confinement, message passing).
- If sharing is necessary, use **mutexes with RAII** or **atomics with correct ordering**.
- `volatile` is **not** a synchronization mechanism in ISO C++.

Chapter 76

Mutexes

76.1 Purpose

A **mutex** is the primary standard C++ mechanism for **mutual exclusion**: ensuring that only one thread at a time executes a critical section that reads or writes shared state. Used correctly, mutexes prevent **data races** and help preserve invariants of shared objects. Used incorrectly, they create deadlocks, poor scalability, and subtle correctness bugs.

This chapter explains how to select a mutex type, how to lock safely with RAII, how to avoid deadlocks, how to combine mutexes with condition variables, and how these practices map to the ISO C++ Core Guidelines rules in Part 12 (CP).

76.2 Core rules from the ISO C++ Core Guidelines (CP)

76.2.1 The minimum set you must follow

- **CP.20:** Use RAII; never manually `lock()`/`unlock()`

- **CP.21:** Use `std::lock()` or `std::scoped_lock` to acquire multiple mutexes
- **CP.22:** Never call unknown code while holding a lock
- **CP.43:** Minimize time spent in a critical section
- **CP.44:** Name your `lock_guard`s and `unique_lock`s
- **CP.50:** Define a mutex together with the data it guards

76.3 When to use a mutex

76.3.1 The main use case

Use a mutex when:

- multiple threads access the same object (or group of related objects), and
- at least one access modifies state, and
- you must preserve an invariant spanning multiple reads/writes.

76.3.2 Prefer not sharing instead

Mutexes are necessary in many systems, but the safest design is to avoid shared mutable state:

- thread confinement (each thread owns its data),
- immutability (share read-only objects),
- message passing (queues, futures).

When you do share, mutex discipline must be explicit and consistent.

76.4 Standard mutex types in C++

76.4.1 `std::mutex`: the default choice

`std::mutex` provides exclusive, non-recursive ownership. It is the default mutex type and should be your first choice unless you have a clear reason to use a different type.

76.4.2 `std::recursive_mutex`: avoid unless you must

`std::recursive_mutex` allows the same thread to lock the mutex multiple times (recursively), requiring the same number of unlocks.
Why it is usually a bad sign. Recursive locking often hides a design problem:

- unclear ownership of invariants,
- re-entrant call chains caused by poor separation of concerns,
- increased risk of deadlocks when multiple locks exist.

If you think you need a recursive mutex, first try to refactor so that each logical operation locks once at the boundary.

76.4.3 `std::timed_mutex` and `std::recursive_timed_mutex`

Timed mutexes allow bounded waits (e.g., `try_lock_for`, `try_lock_until`). They can be useful for:

- avoiding unbounded blocking in non-critical paths,
- implementing timeouts and fallback behavior.

Do not use timeouts to “fix” deadlocks; fix deadlocks by design.

76.4.4 `std::shared_mutex`: many readers, few writers

A shared mutex supports:

- **shared** (read) locking by multiple threads, and
- **exclusive** (write) locking by one thread.

Use it only when you have measured that read-heavy contention is a bottleneck and your design fits the model.

76.5 RAII locking (CP.20)

76.5.1 Never manually `lock()` and `unlock()`

Manual locking is fragile under:

- early returns,
- exceptions,
- future maintenance edits.

The correct pattern is RAII: lock in a constructor, unlock in a destructor.

76.5.2 `std::lock_guard`: simplest and safest

Use `std::lock_guard` when you need to lock exactly one mutex for the whole scope.

```
#include <mutex>
#include <vector>

class Counter {
```

```
public:
    void inc()
    {
        std::lock_guard<std::mutex> lock(m_); // CP.20 + CP.44
        ++value_;
    }

    int get() const
    {
        std::lock_guard<std::mutex> lock(m_);
        return value_;
    }

private:
    mutable std::mutex m_;
    int value_ = 0;
};
```

76.5.3 `std::unique_lock`: for unlock/relock and condition variables

Use `std::unique_lock` when you need:

- to unlock before scope end,
- to pass the lock to a condition variable wait,
- deferred locking or try-lock logic.

A condition variable wait requires a `std::unique_lock<std::mutex>` because the wait operation must temporarily release and then re-acquire the mutex.

76.5.4 `std::scoped_lock`: one or many mutexes (C++17+)

`std::scoped_lock` is an RAII lock that can own zero or more mutexes and is designed to help avoid deadlocks when locking multiple mutexes at once.

76.6 Naming lock objects (CP.44)

76.6.1 Avoid accidental immediate unlock

An unnamed temporary lock is destroyed immediately, so it does not protect the following code.
Bad (lock is a temporary and unlocks immediately):

```
std::mutex m;

void f()
{
    std::lock_guard<std::mutex>(m); // wrong: temporary
    critical_update();              // unprotected
}
```

Good (named lock with clear lifetime):

```
std::mutex m;

void f()
{
    std::lock_guard<std::mutex> lock(m); // CP.44
    critical_update();
}
```

This is exactly why CP.44 exists.

76.7 Critical sections and invariants (CP.43)

76.7.1 Keep the locked region small

Mutexes serialize execution. Your goal is to protect invariants, not to make the thread “safe” by locking everything.

Bad (holds lock during slow work):

```
void update_and_log()
{
    std::lock_guard<std::mutex> lock(m_);
    update_shared_state();
    slow_io(); // do not block other threads on I/O
}
```

Good (lock only around invariant updates):

```
void update_and_log()
{
    {
        std::lock_guard<std::mutex> lock(m_);
        update_shared_state();
    }
    slow_io();
}
```

Minimizing time in the critical section improves throughput and reduces contention.

76.8 Deadlocks and how to avoid them

76.8.1 What is a deadlock

A deadlock occurs when threads wait on each other in a cycle, so none can progress. The classic cause is inconsistent lock ordering.

76.8.2 Lock ordering rule

If you must lock more than one mutex, enforce a global ordering rule (for example, lock `A` before `B` everywhere). This is effective but requires discipline and documentation.

76.8.3 Use `std::scoped_lock` or `std::lock` (CP.21)

When you need to lock multiple mutexes, use standard deadlock-avoiding locking utilities.
Good (C++17+):

```
#include <mutex>

std::mutex m1, m2;

void transfer(Account& a, Account& b, int amount)
{
    std::scoped_lock lock(m1, m2); // CP.21: acquire together
    a.debit(amount);
    b.credit(amount);
}
```

Alternative (C++11/14):

```
#include <mutex>

std::mutex m1, m2;
```

```
void transfer(Account& a, Account& b, int amount)
{
    std::lock(m1, m2);
    std::lock_guard<std::mutex> l1(m1, std::adopt_lock);
    std::lock_guard<std::mutex> l2(m2, std::adopt_lock);

    a.debit(amount);
    b.credit(amount);
}
```

76.8.4 Never call unknown code while holding a lock (CP.22)

Unknown code includes:

- callbacks,
- virtual function calls,
- user-provided function objects,
- logging frameworks that may acquire locks.

Why. Unknown code may try to take the same lock (self-deadlock) or another lock in the wrong order (deadlock), or block for long periods, destroying performance.
Safer pattern (copy under lock, then call outside):

```
#include <mutex>
#include <vector>
#include <functional>

class Publisher {
```

```
public:
    void add(int x)
    {
        std::lock_guard<std::mutex> lock(m_);
        data_.push_back(x);
    }

    void for_each(const std::function<void(int)>& cb) const
    {
        std::vector<int> snapshot;
        {
            std::lock_guard<std::mutex> lock(m_);
            snapshot = data_;
        }
        for (int x : snapshot) cb(x); // no lock held
    }

private:
    mutable std::mutex m_;
    std::vector<int> data_;
};
```

76.9 Mutex + condition variable: correct waiting

76.9.1 The rule: always wait with a predicate

A condition variable wait may wake up:

- due to `notify_one()` / `notify_all()`,

- due to spurious wakeups (allowed),
- due to timeout (for timed waits).

Therefore, waiting must always be done in a loop or using the predicate overload, which is specified as a loop internally.

76.9.2 Correct producer/consumer pattern

```
#include <condition_variable>
#include <mutex>
#include <queue>

class Queue {
public:
    void push(int x)
    {
        {
            std::lock_guard<std::mutex> lock(m_);
            q_.push(x);
        }
        cv_.notify_one();
    }

    int pop()
    {
        std::unique_lock<std::mutex> lock(m_);
        cv_.wait(lock, [&]{ return !q_.empty(); }); // predicate form
        int x = q_.front();
        q_.pop();
```

```
        return x;
    }

private:
    std::mutex m_;
    std::condition_variable cv_;
    std::queue<int> q_;
};
```

76.9.3 Why the predicate must be protected by the same mutex

The predicate typically checks shared state (e.g., `q_.empty()`). That state must be read and written under the same mutex to avoid data races and to ensure correct visibility to waiting threads.

76.10 Encapsulate the mutex with the data (CP.50)

76.10.1 The strongest practical rule for mutex correctness

If the mutex and the protected data live far apart, people will eventually access the data without locking, especially under maintenance pressure. Encapsulation makes the correct usage the easiest usage.

76.10.2 Example: invariant-preserving class

```
#include <mutex>
#include <stdexcept>

class BankAccount {
```

```
public:
    void deposit(int amount)
    {
        if (amount < 0) throw std::invalid_argument("negative");
        std::lock_guard<std::mutex> lock(m_);
        balance_ += amount;
    }

    void withdraw(int amount)
    {
        if (amount < 0) throw std::invalid_argument("negative");
        std::lock_guard<std::mutex> lock(m_);
        if (balance_ < amount) throw std::runtime_error("insufficient funds");
        balance_ -= amount;
    }

    int balance() const
    {
        std::lock_guard<std::mutex> lock(m_);
        return balance_;
    }

private:
    mutable std::mutex m_; // guards balance_ invariant
    int balance_ = 0;
};
```

76.11 Try-lock and timeouts (careful use)

76.11.1 `try_lock()` is not a deadlock solution

`try_lock()` can be useful for:

- opportunistic work,
- avoiding blocking in low-priority paths,
- implementing backoff strategies in performance-tuned systems.

But using try-lock to "avoid deadlocks" usually replaces a clear deadlock with unclear starvation or livelock. Prefer CP.21 and consistent lock ordering.

76.11.2 Timed locks for bounded waiting

Timed locks can be useful when you want a bounded wait and a fallback path. Keep this logic simple and well documented, and treat timeouts as normal control flow (not as correctness mechanisms).

76.12 Performance notes (what matters in practice)

76.12.1 Contention is your real cost

The raw cost of locking is often less important than contention:

- High contention serializes the program.
- Lock handoff can increase context switching and cache misses.

Therefore:

- reduce shared mutable state,
- keep critical sections minimal (CP.43),
- consider sharding (multiple independent locks) when appropriate,
- prefer read-only sharing and message passing when possible.

76.12.2 Avoid “lock around everything” designs

A single global mutex makes code easy to write but often destroys scalability. If correctness requires it, accept the cost, but do not assume it will scale.

76.13 Common mistakes and how to prevent them

76.13.1 Mistake: guarding data with multiple mutexes

If the same data is sometimes protected by `m1` and sometimes by `m2`, the code is broken. Each piece of shared state must have one clear synchronization strategy.

76.13.2 Mistake: returning references into protected data

Returning a reference or pointer to internal data often escapes the lock’s protection.
Bad (caller uses reference without holding lock):

```
class Bad {
public:
    int& value()
    {
        std::lock_guard<std::mutex> lock(m_);
        return value_; // reference escapes
```

```
    }
private:
    std::mutex m_;
    int value_ = 0;
};
```

Better (return by value, or provide locked operations):

```
class Good {
public:
    int value() const
    {
        std::lock_guard<std::mutex> lock(m_);
        return value_;
    }

    void set_value(int v)
    {
        std::lock_guard<std::mutex> lock(m_);
        value_ = v;
    }
private:
    mutable std::mutex m_;
    int value_ = 0;
};
```

76.13.3 Mistake: locking and then calling user code

This violates CP.22. Move the call outside the lock or redesign to avoid callbacks in locked contexts.

76.13.4 Mistake: using `volatile` for synchronization

`volatile` does not provide inter-thread synchronization in ISO C++. Use mutexes or atomics.

76.14 Summary

- Use `std::mutex` by default; use other mutex types only with a clear justification.
- Always lock with RAII (CP.20) and name the lock objects (CP.44).
- Keep critical sections short (CP.43).
- Avoid deadlocks by locking multiple mutexes with `std::scoped_lock` or `std::lock` (CP.21), and never call unknown code while holding locks (CP.22).
- Encapsulate the mutex with the data it guards (CP.50).
- For waiting, combine mutexes with condition variables using the predicate wait pattern.

Chapter 77

Atomics

77.1 Purpose

Atomic operations are the C++ language and library mechanism for performing concurrent reads and writes to a variable *without a data race*. Atomics are also the foundation for defining **inter-thread synchronization** (visibility and ordering) through the C++ memory model.
This chapter explains:

- what `std::atomic` guarantees and what it does not,
- when to choose atomics instead of mutexes (and vice versa),
- the meaning of memory ordering (`std::memory_order`),
- safe publication patterns (release/acquire),
- atomic waiting and notification (`wait`/`notify_*`, since C++20),
- `std::atomic_ref` (since C++20).

77.2 Core Guidelines alignment (Part 12, CP)

The ISO C++ Core Guidelines emphasize:

- **Avoid data races** (CP.2) and minimize shared writable state (CP.3) .
- **Do not use** `volatile` **for synchronization** (CP.8) .
- Prefer designs that communicate via tasks, values, and well-defined synchronization rather than ad-hoc shared state (CP.4, CP.60, CP.61) .

Atomics are one of the main correct tools when you truly need shared state.

77.3 What `std::atomic` guarantees

77.3.1 No data race on the atomic object itself

For an object of type `std::atomic<T>`, concurrent atomic reads/writes (and read-modify-write operations) on that atomic object are **well-defined**.

77.3.2 Visibility and ordering (when you ask for it)

Atomic operations can establish **inter-thread synchronization** and can order surrounding non-atomic operations as specified by `std::memory_order`.

77.3.3 What `std::atomic` does *not* guarantee

- Atomics do **not** automatically make other (non-atomic) shared data safe.
- Atomics do **not** automatically preserve complex invariants across multiple variables.
- Atomics do **not** eliminate logical race conditions (algorithmic mistakes).

77.4 When to use atomics vs. mutexes

77.4.1 Use atomics when

- The shared state is a **single variable** (or a small independent set) and operations are simple.
- You need **lock-free** or low-latency coordination (counters, flags, state machines).
- You want **release/acquire** visibility between threads without a mutex.

77.4.2 Use mutexes when

- You must protect an invariant spanning **multiple variables** or complex structure mutations.
- You need to perform a sequence of operations atomically as a unit.
- The code must remain easy to maintain by most engineers.

77.4.3 A practical engineering rule

If the correct atomic protocol is not obvious and reviewable, a mutex is usually the safer choice. Atomics can be correct and fast, but they are easier to get subtly wrong.

77.5 Atomic operations: the essential set

77.5.1 Load and store

`load()` reads the atomic value, `store()` writes it:

```cpp
#include <atomic>

std::atomic<int> a{0};

int read_a()
{
    return a.load();  // default: memory_order_seq_cst
}

void write_a(int v)
{
    a.store(v);       // default: memory_order_seq_cst
}
```

Defaults are sequentially consistent unless you specify a memory order.

77.5.2 Read-modify-write (RMW) operations

RMW operations update the atomic value as a single atomic step:

```cpp
#include <atomic>

std::atomic<int> counter{0};

int next_id()
{
    // returns old value, then increments
    return counter.fetch_add(1, std::memory_order_relaxed);
}
```

RMW operations are the standard way to implement counters and indices safely.

77.5.3 Compare-and-exchange (CAS)

CAS compares the current value with an expected value and conditionally replaces it:

```
#include <atomic>

std::atomic<int> state{0};

bool try_set_state(int expected, int desired)
{
    return state.compare_exchange_strong(expected, desired,
                                         std::memory_order_acq_rel);
}
```

CAS is the core primitive for many lock-free algorithms, but it must be used with care.

77.6 The memory order model (what it means, simply)

77.6.1 Why memory ordering exists

On modern CPUs, different threads can observe memory operations in different apparent orders unless synchronization constraints are used. C++ exposes this reality through `std::memory_order`.

77.6.2 The six standard orders

C++ provides:

- `memory_order_relaxed`
- `memory_order_consume` (use is discouraged in practice; implementations often treat it as acquire)

- `memory_order_acquire`
- `memory_order_release`
- `memory_order_acq_rel`
- `memory_order_seq_cst`

77.6.3 Relaxed: atomicity without synchronization

Use case. Counters and statistics where you only need atomicity, not ordering.

```
#include <atomic>

std::atomic<unsigned long long> hits{0};

void record_hit()
{
    hits.fetch_add(1, std::memory_order_relaxed);
}
```

Meaning. Other threads will see a coherent modification order for `hits`, but the increment does not synchronize other data.

77.6.4 Release and acquire: publishing and consuming

Idea.

- **Release** on a store: publish prior writes in the same thread.
- **Acquire** on a load: after observing the released value, also observe the publisher's prior writes.

This is the standard pattern for safe publication.

77.6.4.1 Example: safe publication of data

```
#include <atomic>
#include <cassert>

struct Payload {
    int x;
    int y;
};

Payload data;                  // non-atomic data
std::atomic<bool> ready{false};

void producer()
{
    data.x = 10;
    data.y = 20;
    ready.store(true, std::memory_order_release);
}

void consumer()
{
    while (!ready.load(std::memory_order_acquire)) {
        // spin or use wait() (see later)
    }
    // After acquire, reads of data are guaranteed to see producer's writes.
    assert(data.x == 10 && data.y == 20);
}
```

Critical condition. All accesses to `data` that rely on publication must follow the

release/acquire protocol.

77.6.5 Sequential consistency: simplest mental model

`memory_order_seq_cst` provides the strongest ordering and the simplest model: atomic operations appear in a single global order consistent with program order. It is often a safe starting point when correctness is more important than maximum performance.

77.7 Fences (advanced, use sparingly)

77.7.1 What a fence does

`std::atomic_thread_fence(order)` establishes ordering constraints for non-atomic and relaxed atomic operations around it, but it does not by itself "send a signal" to another thread; at least one atomic operation is still needed to form synchronization.

77.7.2 Engineering advice

Prefer release/acquire on the atomic operations themselves. Use fences only when you have a clear, reviewed reason and you can explain the synchronization chain precisely.

77.8 Atomic waiting and notification (C++20)

77.8.1 Why it exists

Busy-waiting (spinning) wastes CPU and can harm performance. C++20 introduced atomic waiting operations that allow a thread to block efficiently until an atomic value changes, similar in intent to condition variables but tied to an atomic object.

77.8.2 wait/notify_one/notify_all

```cpp
#include <atomic>
#include <thread>

std::atomic<int> phase{0};

void worker()
{
    // Wait until phase != 0
    phase.wait(0); // blocks while value equals 0 (since C++20)
    // proceed
}

void start()
{
    std::jthread t(worker);

    // publish: change value then notify
    phase.store(1, std::memory_order_release);
    phase.notify_one();
}
```

Key points.

- `wait(old)` blocks while the atomic equals `old`, and returns when it changes (spurious wakeups are permitted; the correct pattern is still "check in a loop" conceptually).
- `notify_one` unblocks at least one waiting thread (if any).

77.9 `std::atomic_ref` (C++20)

77.9.1 What it is

`std::atomic_ref<T>` allows atomic operations on an existing non-atomic object by creating an atomic reference view to that object.

77.9.2 The strict rule you must follow

While an object is referenced by an `atomic_ref`, it must be accessed **exclusively** through `atomic_ref` operations for the entire period where any `atomic_ref` exists for that object (otherwise you introduce a data race). Object lifetime must also exceed the lifetime of all its `atomic_ref` instances.

77.9.3 Example: atomic updates to a field that you cannot change to `std::atomic`

```
#include <atomic>
#include <cstdint>

struct Node {
    std::uint32_t flags;
};

void set_flag(Node& n, std::uint32_t mask)
{
    std::atomic_ref<std::uint32_t> f(n.flags);
    f.fetch_or(mask, std::memory_order_relaxed);
}
```

77.10 Lock-free does not mean faster

77.10.1 Lock-free and wait-free (terminology)

- **Lock-free** means that *some* thread makes progress system-wide, even if others are delayed.
- **Wait-free** means each thread completes its operation in a bounded number of steps.

Whether `std::atomic<T>` is lock-free is platform- and type-dependent. Use `is_lock_free()` or `is_always_lock_free` to query.

77.10.2 Performance reality

Atomics can be slower than a well-designed mutex under high contention due to cache-line bouncing and expensive RMW operations. Measure, then decide.

77.11 Common mistakes (and correct replacements)

77.11.1 Mistake: using `volatile` instead of atomics

Why it is wrong. `volatile` does not provide inter-thread synchronization in ISO C++. It does not create happens-before and does not prevent data races.
Replacement. Use `std::atomic` with the weakest correct ordering (often relaxed for counters; release/acquire for publication).

77.11.2 Mistake: mixing atomic and non-atomic access to the same object

If one thread writes with atomics and another thread reads/writes the same object non-atomically, you still have undefined behavior. The object must be accessed consistently via

atomic operations (or protected by a mutex).

77.11.3 Mistake: using relaxed when you need visibility

`memory_order_relaxed` is correct only when you need atomicity but not ordering. If you use a flag to publish data, you typically need release/acquire (or stronger).

77.11.4 Mistake: building "clever" lock-free structures without a proof

Lock-free algorithms require careful reasoning about ordering, lifetimes, and ABA-style issues. Unless you can explain and justify the ordering and correctness, prefer simpler designs (mutex + clear invariants, or message passing).

77.12 Recommended patterns (safe and reviewable)

77.12.1 Pattern 1: relaxed counters

Use relaxed for statistics:

```
std::atomic<std::uint64_t> requests{0};
void on_request() { requests.fetch_add(1, std::memory_order_relaxed); }
```

77.12.2 Pattern 2: one-way publication

Producer writes data then releases a flag; consumer acquires the flag then reads the data (publication example earlier).

77.12.3 Pattern 3: atomic state machine

Use an atomic integer/enum for state transitions, often with `compare_exchange` and `acq_rel`:

```cpp
#include <atomic>

enum class Status : int { idle, running, done };
std::atomic<Status> st{Status::idle};

bool start()
{
    Status expected = Status::idle;
    return st.compare_exchange_strong(expected, Status::running,
                                      std::memory_order_acq_rel);
}
```

77.12.4 Pattern 4: wait/notify for efficient blocking (C++20)

Use `wait`/`notify_*` to avoid spinning when appropriate.

77.13 Summary

- Atomics eliminate data races on the atomic object and can provide synchronization and ordering when used with correct memory orders.
- Prefer `relaxed` for pure counters, `release`/`acquire` for publication, and `seq_cst` as a safe baseline when unsure.
- Do not use `volatile` for thread synchronization (Core Guidelines CP.8).
- Use mutexes for multi-variable invariants and complex shared objects; use atomics for simple shared-state coordination.
- C++20 adds atomic waiting/notification and `atomic_ref` with strict access rules.

Chapter 78

Deadlocks

78.1 Purpose

A **deadlock** is a liveness failure: threads stop making progress because each is waiting for something that can never happen. Unlike a data race (undefined behavior), a deadlock often produces a program that *appears* correct but freezes under load or rare timing.
This chapter explains what deadlocks are, why they happen, how they appear in modern C++ code, and how to prevent them using the **ISO C++ Core Guidelines (CP)** and the C++ Standard Library synchronization facilities.

78.2 Definition and the classic necessary conditions

78.2.1 Operational definition

A deadlock occurs when there is a cycle of waiting:

- Thread A holds resource X and waits for resource Y.

- Thread B holds resource Y and waits for resource X.

Neither thread can proceed, so the system makes no progress.

78.2.2 The four necessary conditions (Coffman conditions)

In practice, deadlocks arise when all of the following are present:

1. **Mutual exclusion:** at least one resource is held exclusively (e.g., a mutex).
2. **Hold and wait:** a thread holds one resource while waiting for another.
3. **No preemption:** resources are released only voluntarily (mutexes are not forcibly taken).
4. **Circular wait:** there is a cycle of waiting across threads.

To prevent deadlocks, you typically remove at least one of these conditions by design.

78.3 How deadlocks appear in C++

78.3.1 The most common causes

- **Inconsistent lock order** across code paths.
- **Calling unknown code while holding a lock** (callbacks, virtual functions, logging).
- **Waiting while holding a lock** in the wrong way (condition variables without a correct predicate pattern).
- **Locking multiple mutexes without a deadlock-avoidance algorithm**.
- **Joining or stopping threads while holding locks** that the joined thread might need to complete.

78.4 Core Guidelines rules that directly prevent deadlocks

The ISO C++ Core Guidelines Part 12 (CP) includes several rules that are specifically about preventing deadlocks and lock-induced liveness failures:

- **CP.20:** Use RAII, never plain `lock()`/`unlock()`.
- **CP.21:** Use `std::lock()` or `std::scoped_lock` to acquire multiple mutexes.
- **CP.22:** Never call unknown code while holding a lock.
- **CP.43:** Minimize time spent in a critical section.
- **CP.44:** Remember to name your `lock_guards` and `unique_locks`.
- **CP.50:** Define a mutex together with the data it guards.

78.5 Deadlock pattern 1: inconsistent lock ordering

78.5.1 Example: two mutexes locked in opposite order

Bug. The following code can deadlock when executed concurrently:

```
#include <mutex>

std::mutex m1;
std::mutex m2;

void f()
{
    std::lock_guard<std::mutex> l1(m1);
```

```
    std::lock_guard<std::mutex> l2(m2);
    // ...
}

void g()
{
    std::lock_guard<std::mutex> l2(m2);
    std::lock_guard<std::mutex> l1(m1);
    // ...
}
```

If thread T1 enters `f()` and locks `m1`, while thread T2 enters `g()` and locks `m2`, each thread can block forever trying to acquire the other mutex.

78.5.2 Fix A: use `std::scoped_lock` (C++17+) (CP.21)

`std::scoped_lock` can lock multiple mutexes using a deadlock-avoidance algorithm (as if by `std::lock`).

```
#include <mutex>

std::mutex m1;
std::mutex m2;

void f()
{
    std::scoped_lock lock(m1, m2); // CP.21
    // ...
}
```

```
void g()
{
    std::scoped_lock lock(m1, m2); // CP.21
    // ...
}
```

78.5.3 Fix B: use `std::lock` + `adopt_lock` (C++11/14) (CP.21)

```
#include <mutex>

std::mutex m1;
std::mutex m2;

void f()
{
    std::lock(m1, m2); // deadlock-avoidance algorithm
    std::lock_guard<std::mutex> l1(m1, std::adopt_lock);
    std::lock_guard<std::mutex> l2(m2, std::adopt_lock);
    // ...
}
```

78.5.4 Fix C: impose and document a strict lock hierarchy

Another correct method is a global rule such as “always lock `m1` before `m2`”. This can work well in large systems, but requires rigorous code review, documentation, and tooling support.

78.6 Deadlock pattern 2: calling unknown code while holding a lock (CP.22)

78.6.1 Why this is dangerous

Unknown code may:

- try to lock the same mutex again (self-deadlock with a non-recursive mutex),
- lock other mutexes in an order that conflicts with your code,
- block for a long time (I/O, allocation contention, logging locks),
- call back into your component re-entrantly.

78.6.2 Bad example: callback under lock

```
#include <mutex>
#include <vector>
#include <functional>

class Store {
public:
    void add(int x)
    {
        std::lock_guard<std::mutex> lock(m_);
        v_.push_back(x);
    }

    void for_each(const std::function<void(int)>& cb) const
```

```
    {
        std::lock_guard<std::mutex> lock(m_);
        for (int x : v_) {
            cb(x); // CP.22 violation: unknown code under lock
        }
    }

private:
    mutable std::mutex m_;
    std::vector<int> v_;
};
```

78.6.3 Good example: snapshot under lock, call outside

```
#include <mutex>
#include <vector>
#include <functional>

class Store {
public:
    void add(int x)
    {
        std::lock_guard<std::mutex> lock(m_);
        v_.push_back(x);
    }

    void for_each(const std::function<void(int)>& cb) const
    {
        std::vector<int> snapshot;
```

```
        {
            std::lock_guard<std::mutex> lock(m_);
            snapshot = v_;
        }
        for (int x : snapshot) {
            cb(x); // safe: no lock held
        }
    }

private:
    mutable std::mutex m_;
    std::vector<int> v_;
};
```

78.7 Deadlock pattern 3: condition variables used incorrectly

78.7.1 The correct mental model

A condition variable is used to wait until a shared-state predicate becomes true. Key points:

- Wait may wake up spuriously.
- Wait must release the mutex while sleeping, then re-acquire it before returning.
- The predicate must be checked under the same mutex that protects the shared state.

78.7.2 Correct pattern: predicate wait

```
#include <condition_variable>
#include <mutex>
```

```
#include <queue>

class Queue {
public:
    void push(int x)
    {
        {
            std::lock_guard<std::mutex> lock(m_);
            q_.push(x);
        }
        cv_.notify_one();
    }

    int pop()
    {
        std::unique_lock<std::mutex> lock(m_);
        cv_.wait(lock, [&]{ return !q_.empty(); }); // correct predicate wait
        int x = q_.front();
        q_.pop();
        return x;
    }

private:
    std::mutex m_;
    std::condition_variable cv_;
    std::queue<int> q_;
};
```

78.7.3 Typical deadlock: waiting while holding another lock

A subtle deadlock occurs when a thread holds **two** locks and then waits, preventing the producer thread from acquiring the second lock needed to satisfy the predicate.

Bad design sketch (do not copy):

```
std::mutex m_queue;
std::mutex m_other;
std::condition_variable cv;
bool ready = false;

void consumer()
{
    std::unique_lock<std::mutex> lq(m_queue);
    std::lock_guard<std::mutex> lo(m_other); // holds another lock
    cv.wait(lq, []{ return ready; });        // waits while holding m_other
}

void producer()
{
    std::lock_guard<std::mutex> lo(m_other); // may block forever
    {
        std::lock_guard<std::mutex> lq(m_queue);
        ready = true;
    }
    cv.notify_one();
}
```

Fix. Never wait while holding unrelated locks. Reduce the critical region and hold only the mutex associated with the condition variable predicate during `wait`.

78.8 Deadlock pattern 4: joining threads while holding locks

78.8.1 The problem

If you call `join()` (or let a joining thread destructor run) while holding a lock that the worker thread may need to exit, you can deadlock:

- Main thread holds mutex M and waits for worker to finish (`join`).
- Worker needs mutex M to finish cleanup and exit.

78.8.2 Bad example

```
#include <mutex>
#include <thread>

std::mutex m;

void worker()
{
    std::lock_guard<std::mutex> lock(m);
    // cleanup that requires m
}

void shutdown(std::thread& t)
{
    std::lock_guard<std::mutex> lock(m);
    // ...
    t.join(); // deadlock risk: worker may need m to finish
}
```

78.8.3 Safe rule

Never block on thread completion while holding locks that the thread may need. Release locks first, then join.

78.8.4 Better design with C++20 `std::jthread` and `stop_token`

C++20 `std::jthread` combines RAII joining with cooperative cancellation via `std::stop_token`. This helps structure shutdown so threads can exit without needing hazardous forced termination.

```
#include <mutex>
#include <thread>
#include <stop_token>

class Service {
public:
    Service()
        : thr_([this](std::stop_token st){ run(st); })
    {}

    void request_stop()
    {
        thr_.request_stop();
        // No join here: jthread joins automatically in destructor
    }

    ~Service()
    {
        // Ensure we are not holding locks that run() needs.
```

```
        thr_.request_stop();
    }

private:
    void run(std::stop_token st)
    {
        while (!st.stop_requested()) {
            // do work; keep lock scopes small
            {
                std::lock_guard<std::mutex> lock(m_);
                // update shared state
            }
            // do non-critical work without holding m_
        }
        // cleanup, with minimal lock usage
    }

    std::mutex m_;
    std::jthread thr_;
};
```

This does not magically prevent deadlocks, but it strongly supports a correct **structured shutdown** design.

78.9 Engineering strategies for deadlock prevention

78.9.1 Strategy 1: avoid shared mutable state

The strongest prevention is architectural:

- immutable shared data,
- thread-local ownership,
- message passing (queues, futures),
- task-based design.

78.9.2 Strategy 2: keep mutexes with the data they guard (CP.50)

Encapsulate mutex + data in a single type so access is always mediated correctly. This reduces accidental lock order inversions and missing locks.

78.9.3 Strategy 3: lock multiple mutexes safely (CP.21)

When you must take multiple locks, prefer:

- `std::scoped_lock` (C++17+) for immediate RAII locking,
- `std::lock` + `adopt_lock` (C++11/14) where needed.

78.9.4 Strategy 4: keep critical sections short (CP.43)

Shorter lock hold times reduce the chance of forming deadlock cycles and reduce contention.

78.9.5 Strategy 5: do not call unknown code under lock (CP.22)

If you must call out, make a snapshot under lock and call outside.

78.9.6 Strategy 6: prefer one lock per invariant

A single coherent mutex per invariant is often safer than many fine-grained locks. Fine-grained locking can improve performance but increases the risk of deadlocks and reasoning complexity.

78.10 Detecting and diagnosing deadlocks

78.10.1 Symptoms

- Threads stuck in `mutex::lock`, `condition_variable::wait`, or `join`.
- CPU usage near zero (classic deadlock) or high CPU with no progress (possible livelock/spin).

78.10.2 Practical diagnostics

- Use debugger thread backtraces to identify lock ownership and wait sites.
- Add logging around lock acquisition (carefully) and around waits.
- Use tooling that can detect lock order inversions or deadlocks in tests where available.

78.11 Deadlock vs. livelock vs. starvation

78.11.1 Deadlock

No progress due to a cycle of waiting.

78.11.2 Livelock

Threads keep running (often retrying) but no useful progress occurs (for example, two threads repeatedly backing off and retrying a try-lock protocol).

78.11.3 Starvation

A thread waits indefinitely because it never gets scheduled or never acquires a resource due to unfairness.
Deadlock avoidance techniques (like `std::lock`) address deadlocks but do not automatically solve starvation or livelock; those require separate design and measurement.

78.12 Summary

- Deadlocks are primarily caused by inconsistent lock ordering, waiting under locks, calling unknown code under locks, and unsafe multi-lock acquisition.
- Follow Core Guidelines rules: CP.20, CP.21, CP.22, CP.43, CP.44, CP.50.
- Use `std::scoped_lock` or `std::lock` for multiple mutexes.
- Use condition variables with predicate waits, and never wait while holding unrelated locks.
- Never join (or wait for thread termination) while holding locks that the thread may need to finish.

Chapter 79

Concurrent System Design

79.1 Purpose

Concurrent system design is the discipline of building software that stays **correct**, **responsive**, and **maintainable** when multiple activities overlap in time. In C++, correctness is tightly bound to the language memory model: **data races are undefined behavior**, and many liveness failures (deadlocks, starvation) can be introduced by small design mistakes.
This chapter focuses on architecture-level guidance: how to choose a concurrency model, how to structure ownership and lifetime, how to communicate between activities, how to stop and shut down safely, and how to select the right C++ standard facilities.

79.2 Guiding principles from the ISO C++ Core Guidelines (CP)

The Core Guidelines emphasize a few foundational ideas:

- **Assume multi-threading:** code is reused and may become concurrent later (CP.1).

- **Avoid data races:** they cause undefined behavior (CP.2).
- **Minimize shared writable state:** prefer designs that do not share mutation (CP.3).
- **Think in tasks, not threads:** express intent at a higher level (CP.4).
- **Locks must be disciplined:** RAII locking, safe multi-lock acquisition, no unknown calls under lock, short critical sections (CP.20–CP.22, CP.43–CP.44).
- **Keep the mutex with the data:** make correct use the easiest use (CP.50).
- **Use futures/async for task results when appropriate:** communicate results as values (CP.60–CP.61).

79.3 Concurrency vs. parallelism (design consequences)

79.3.1 Concurrency

Concurrency addresses **overlapping** work, such as:

- handling many I/O events,
- serving many requests,
- running background maintenance.

The main risks are correctness and liveness: data races, deadlocks, lifetime errors, shutdown hazards.

79.3.2 Parallelism

Parallelism uses hardware to run work **simultaneously** (multiple cores). The main risks are scalability problems:

- contention on shared locks/atomics,
- false sharing and cache-line bouncing,
- overhead from thread creation and context switching.

79.4 The three design axes you must decide early

79.4.1 Axis 1: Shared-state vs. message-passing

Shared-state concurrency shares objects across threads and synchronizes access (mutexes, atomics). **Message-passing concurrency** avoids shared mutation by sending values (queues, channels, futures).
Core recommendation. Prefer message passing and immutable sharing where feasible (CP.3, CP.4).

79.4.2 Axis 2: Unstructured vs. structured concurrency

Unstructured concurrency creates background threads that may outlive the scope that created them (common with `detach()`). **Structured concurrency** keeps concurrent work *scoped*: tasks start and finish within a defined lifetime boundary.
In modern C++ practice, `std::jthread` (C++20) supports structured thread lifetimes by joining on destruction and providing cooperative cancellation via `stop_token`.

79.4.3 Axis 3: Blocking vs. non-blocking coordination

Blocking uses mutexes, condition variables, semaphores, and waits. **Non-blocking** uses atomics and lock-free techniques.
Engineering rule:

- Use blocking primitives for clarity and correctness first.
- Use lock-free techniques only when you can precisely explain the correctness and you have a measurable performance need.

79.5 Concurrency safety contract for every component

Every concurrent component should document (and enforce) a safety contract:

- **Thread-safety level:**
  - *Thread-compatible:* safe if externally synchronized.
  - *Thread-safe:* safe for concurrent use (internal synchronization).
  - *Thread-confined:* must be used by one thread only.
- **Ownership and lifetime:** which thread owns what, and what keeps objects alive.
- **Shutdown protocol:** how to stop work and drain pending operations.
- **Error propagation:** where errors go (return values, exceptions, futures).

79.6 Model A: Shared-state design with encapsulated locking

79.6.1 The rule: protect invariants, not lines of code

A mutex should guard an invariant (a relationship between fields), not a random collection of statements. Keep the mutex inside the type that owns the invariant (CP.50).

79.6.2 Example: a thread-safe bounded queue

This example shows several design rules:

- mutex is private and guards the queue invariant,
- waiting uses a predicate to handle spurious wakeups,
- critical sections are minimal.

```
#include <condition_variable>
#include <mutex>
#include <queue>
#include <stdexcept>

template<class T>
class BoundedQueue {
public:
    explicit BoundedQueue(std::size_t capacity) : cap_(capacity) {
        if (cap_ == 0) throw std::invalid_argument("capacity");
    }

    void push(T value)
    {
        std::unique_lock<std::mutex> lock(m_);
        not_full_.wait(lock, [&]{ return q_.size() < cap_; });

        q_.push(std::move(value));

        lock.unlock();
        not_empty_.notify_one();
```

```
    }

    T pop()
    {
        std::unique_lock<std::mutex> lock(m_);
        not_empty_.wait(lock, [&]{ return !q_.empty(); });

        T value = std::move(q_.front());
        q_.pop();

        lock.unlock();
        not_full_.notify_one();
        return value;
    }

    std::size_t size() const
    {
        std::lock_guard<std::mutex> lock(m_);
        return q_.size();
    }

private:
    const std::size_t cap_;
    mutable std::mutex m_;
    std::condition_variable not_empty_;
    std::condition_variable not_full_;
    std::queue<T> q_;
};
```

79.6.3 Avoiding deadlocks in shared-state systems

Deadlocks usually come from:

- inconsistent lock ordering,
- calling unknown code while holding a lock (callbacks, virtual functions) (CP.22),
- waiting while holding unrelated locks.

When you must lock multiple mutexes, use `std::scoped_lock` or `std::lock` (CP.21).

```
#include <mutex>

std::mutex m1, m2;

void update_two_resources()
{
    std::scoped_lock lock(m1, m2); // deadlock-avoiding multi-lock (C++17+)
    // update both resources atomically with respect to other threads
}
```

79.7 Model B: Message-passing design (queues, actors, and values)

79.7.1 Why message passing scales better

Message passing reduces or eliminates shared writable state (CP.3). Benefits:

- fewer data races by construction,
- fewer locks,

- easier reasoning about ownership and lifetime.

79.7.2 Single-owner event loop (actor-style) skeleton

One of the simplest and most reliable concurrency models is:

- each component owns its state,
- other threads communicate by sending messages to its queue,
- the owner thread processes messages sequentially.

```
#include <mutex>
#include <condition_variable>
#include <queue>
#include <variant>
#include <thread>
#include <stop_token>

struct Add { int x; };
struct Get { int* out; }; // for illustration only; prefer futures in real code
using Msg = std::variant<Add, Get>;

class Actor {
public:
    Actor() : thr_([this](std::stop_token st){ run(st); }) {}

    void send(Msg m)
    {
        {
            std::lock_guard<std::mutex> lock(m_);
```

```
            q_.push(std::move(m));
        }
        cv_.notify_one();
    }

    void stop() { thr_.request_stop(); cv_.notify_all(); }

private:
    void run(std::stop_token st)
    {
        int sum = 0; // owned state: accessed by this thread only
        while (!st.stop_requested()) {
            Msg msg;
            {
                std::unique_lock<std::mutex> lock(m_);
                cv_.wait(lock, [&]{ return st.stop_requested() || !q_.empty(); });
                if (st.stop_requested()) break;
                msg = std::move(q_.front());
                q_.pop();
            }

            if (auto* a = std::get_if<Add>(&msg)) {
                sum += a->x;
            } else if (auto* g = std::get_if<Get>(&msg)) {
                *g->out = sum; // demonstration: prefer std::promise/std::future
            }
        }
    }
```

```
    std::mutex m_;
    std::condition_variable cv_;
    std::queue<Msg> q_;
    std::jthread thr_; // joins automatically
};
```

Note. The message queue still uses a mutex, but the *actor state* is not shared, which is the major simplification.

79.8 Structured shutdown and cancellation (C++20)

79.8.1 Why shutdown is a first-class design problem

Many production deadlocks and data races appear during shutdown:

- threads outlive objects they use,
- a destructor blocks while holding a lock,
- cancellation is forced rather than cooperative.

79.8.2 Use `std::jthread` and `std::stop_token` for cooperative cancellation

C++20 introduced `std::jthread` (joining on destruction) and `std::stop_token` for cooperative stop requests.

```
#include <thread>
#include <stop_token>
#include <chrono>
#include <iostream>
```

```
void worker(std::stop_token st)
{
    while (!st.stop_requested()) {
        // do a unit of work
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
    // cleanup here
}

int main()
{
    std::jthread t(worker); // auto-join
    std::this_thread::sleep_for(std::chrono::milliseconds(50));
    t.request_stop();       // cooperative cancellation
}
```

79.8.3 Rule: never join while holding locks the worker might need

If the worker needs a lock to finish, joining while holding that lock can deadlock. Design shutdown so:

- signal stop,
- release locks,
- then wait for completion.

79.9 Selecting the right synchronization primitive

79.9.1 Mutexes and condition variables

Use when protecting invariants or waiting for state transitions. Follow CP.20–CP.22 and predicate-based waiting.

79.9.2 Atomics

Use for simple shared variables (counters, flags, state machines) where a mutex would be excessive. Atomics require correct memory ordering to publish and observe other data.

79.9.3 Atomic waiting (`wait`/`notify`) (C++20)

C++20 adds blocking waits on atomics, enabling efficient coordination without spinning.

```
#include <atomic>
#include <thread>

std::atomic<int> phase{0};

void consumer()
{
    phase.wait(0);      // blocks while value equals 0 (C++20)
    // proceed after phase changes
}

void producer()
{
    phase.store(1, std::memory_order_release);
    phase.notify_one(); // wake a waiter
}
```

79.9.4 Latches and barriers (C++20)

`std::latch` is a one-shot countdown gate; `std::barrier` is reusable for repeated phases. They are useful for organizing phases in a parallel pipeline.

```
#include <barrier>
#include <thread>
#include <vector>

void phase_work(int id);

int main()
{
    const int n = 4;
    std::barrier sync(n);

    std::vector<std::jthread> ts;
    ts.reserve(n);
    for (int i = 0; i < n; ++i) {
        ts.emplace_back([&, i](std::stop_token){
            phase_work(i);
            sync.arrive_and_wait(); // end of phase 1
            phase_work(i);
            sync.arrive_and_wait(); // end of phase 2
        });
    }
}
```

79.9.5 Semaphores (C++20)

Semaphores are useful for bounding concurrency (resource permits) and producer/consumer designs. Prefer higher-level designs when possible; use semaphores when they express the problem directly.

79.10 Task-based parallelism and standard algorithms

79.10.1 Parallel algorithms

C++ provides parallel execution policies for certain algorithms, allowing the implementation to parallelize work while preserving a defined interface. This approach aligns with CP.4 (think in tasks).

79.10.2 Practical guidance

- Ensure work per element is large enough to amortize overhead.
- Avoid shared writes inside parallel loops; if necessary, use reduction patterns.
- Measure; parallelism can be slower for small workloads or memory-bound tasks.

79.11 Thread pools and execution resources

79.11.1 Why you usually want a pool

Creating and destroying threads for short tasks is expensive and can harm performance. A thread pool:

- amortizes thread creation cost,

- provides back-pressure (bounded queue),
- can improve locality and throughput.

79.11.2 A minimal pool interface (conceptual)

This is a design sketch focused on contracts rather than a full production pool:

- tasks are values (callables),
- shutdown is structured (stop + drain),
- no detaching.

```
#include <functional>
#include <mutex>
#include <condition_variable>
#include <queue>
#include <thread>
#include <stop_token>
#include <vector>

class ThreadPool {
public:
    explicit ThreadPool(std::size_t n)
    {
        workers_.reserve(n);
        for (std::size_t i = 0; i < n; ++i) {
            workers_.emplace_back([this](std::stop_token st){ worker_loop(st); });
        }
    }
```

```
    void submit(std::function<void()> task)
    {
        {
            std::lock_guard<std::mutex> lock(m_);
            q_.push(std::move(task));
        }
        cv_.notify_one();
    }

    void stop()
    {
        for (auto& t : workers_) t.request_stop();
        cv_.notify_all();
    }

    ~ThreadPool()
    {
        stop(); // jthreads join automatically
    }

private:
    void worker_loop(std::stop_token st)
    {
        while (!st.stop_requested()) {
            std::function<void()> task;
            {
                std::unique_lock<std::mutex> lock(m_);
                cv_.wait(lock, [&]{ return st.stop_requested() || !q_.empty(); });
                if (st.stop_requested()) break;
```

```
                task = std::move(q_.front());
                q_.pop();
            }
            task();
        }
    }

    std::mutex m_;
    std::condition_variable cv_;
    std::queue<std::function<void()>> q_;
    std::vector<std::jthread> workers_;
};
```

79.12 Modern direction: `std::execution` and senders/receivers (C++26 working draft)

There has been major standardization work toward a unified model for asynchronous execution and composition of tasks using **senders/receivers**, published as WG21 paper **P2300** and merged into the **C++26 draft** per committee approval announcements.
Design relevance (even before full ecosystem adoption):

- It encourages **composable tasks** rather than manual threads.
- It makes **execution resources** explicit (where work runs).
- It treats **cancellation and completion** as part of the model.

For system design, this aligns strongly with CP.4 (tasks over threads) and CP.60/CP.61 (value-based results and task spawning).

79.13 Performance and scalability rules for architects

79.13.1 Rule 1: reduce contention first

Contention (many threads fighting over the same lock/atomic) usually dominates overhead. Use:

- sharding (multiple independent locks),
- per-thread buffers with periodic merge,
- message passing and ownership.

79.13.2 Rule 2: avoid false sharing

If independent thread-local counters share the same cache line, performance collapses. Architect data layouts to separate frequently-written thread-local variables.

79.13.3 Rule 3: control parallelism, do not assume "more threads is faster"

Bound concurrency by the number of cores and by the nature of the workload (CPU-bound vs I/O-bound). Use pools and queues to avoid thread explosion.

79.13.4 Rule 4: prefer clear correctness over cleverness

A correct mutex-based design that is easy to review often beats a fragile lock-free design. Use lock-free only when:

- correctness can be explained precisely,
- you can test and analyze it thoroughly,
- performance measurements justify it.

79.14 Design review checklist (practical)

- **No data races:** every shared writable object has exactly one synchronization strategy.
- **Ownership is explicit:** no dangling references into asynchronous work.
- **No detach:** background work is scoped or owned and stoppable.
- **Lock discipline:** RAII locks only, safe multi-lock acquisition, no unknown calls under lock, short critical sections.
- **Shutdown is designed:** stop request path, draining policy, and destructor behavior are all defined.
- **Error propagation exists:** futures/promises or explicit channels; no silent thread failures.
- **Scalability is measured:** contention hot spots identified, not guessed.

79.15 Summary

A concurrent system is correct and maintainable when it is built around:

- **minimal shared mutable state** (CP.3),
- **task-oriented structure** (CP.4),
- **structured lifetimes and shutdown** (`std::jthread`, `stop_token`),
- **disciplined synchronization** (CP.20–CP.22, CP.43–CP.44, CP.50),
- **value-based communication** (queues, futures, and evolving `std::execution` models).

Chapter 80

Message Passing

80.1 Purpose

Message passing is a concurrency design approach where threads (or tasks) **do not share writable state**. Instead, they exchange **messages** (values) through well-defined channels such as queues, futures, or other synchronization objects.
In C++, this approach is strongly recommended because:

- it reduces the chance of **data races** (which are undefined behavior),
- it reduces **deadlock** risk by minimizing lock coupling,
- it improves maintainability by making ownership and communication explicit.

This chapter explains message-passing design, shows correct standard C++ patterns, and maps them to the ISO C++ Core Guidelines (CP.mess).

80.2 Core Guidelines view: CP.mess (Message passing)

The C++ Core Guidelines explicitly present message passing as a higher-level alternative to low-level shared-state synchronization, with the summary rules:

- **CP.60:** Use a `future` to return a value from a concurrent task.
- **CP.61:** Use `async()` to spawn concurrent tasks (when appropriate).

They also reinforce the broader principles:

- **CP.2:** Avoid data races.
- **CP.3:** Minimize explicit sharing of writable data.
- **CP.4:** Think in terms of tasks, rather than threads.

80.3 What message passing means in practice

80.3.1 Key idea: ownership stays local

Each concurrent activity owns its internal state. To cooperate, it sends or receives:

- **requests** (commands),
- **data** (payloads),
- **results** (values or errors),
- **signals** (stop, wake-up, completion).

80.3.2 Benefits over shared-state designs

Compared with “multiple threads access the same object guarded by a mutex,” message passing offers:

- fewer shared invariants,
- fewer locks held at once,
- fewer lock-order constraints,
- clearer lifetime boundaries.

80.3.3 Costs and trade-offs

Message passing is not free:

- copies/moves of messages (often mitigated by move semantics),
- queueing overhead and potential back-pressure needs,
- latency from scheduling and blocking,
- need to design shutdown and draining policies.

80.4 Standard C++ building blocks for message passing

80.4.1 `std::future` and `std::promise` (CP.60)

`std::promise<T>` stores a result (or an exception) into a shared state, and `std::future<T>` retrieves it. This provides a natural message-passing channel for *one result*.

80.4.1.1 One-way result passing

```
#include <future>
#include <thread>
#include <stdexcept>

int compute()
{
    // ... do work ...
    return 42;
}

int main()
{
    std::promise<int> p;
    std::future<int> f = p.get_future();

    std::thread t([p = std::move(p)]() mutable {
        try {
            p.set_value(compute());
        } catch (...) {
            p.set_exception(std::current_exception());
        }
    });

    // Message received here:
    int v = f.get(); // blocks until set_value/set_exception
    t.join();
}
```

Properties.

- The value (or exception) is the message.
- `future::get()` is the synchronization point.
- This avoids sharing writable state between threads for the returned value.

80.4.1.2 Rules and limitations you must know

- A `std::promise` is intended for **one-time** fulfillment (set once).
- A `std::future` is **move-only** and **single-consumer** by design. If multiple consumers must observe a result, use `std::shared_future`.

80.4.2 `std::async` (CP.61)

`std::async` packages task execution and result delivery into a `std::future`.

80.4.2.1 Force asynchronous execution when you require concurrency

If you need the work to run concurrently, specify `std::launch::async`:

```
#include <future>
#include <string>

int parse_and_compute(std::string input);

int main()
{
    auto fut = std::async(std::launch::async, parse_and_compute,
    ↪  std::string("job"));
    // do other work...
```

```
    int result = fut.get(); // message received (value or exception)
}
```

Important note. Without `std::launch::async`, the implementation may choose deferred execution, meaning the work might run only when `get()` is called. This is not wrong, but it changes the concurrency model.

80.4.3 `std::packaged_task`

`std::packaged_task` wraps a callable and connects it to a future, enabling message passing through a future while giving you control over where/how execution happens (for example in a thread pool).

```
#include <future>
#include <thread>
#include <utility>

int work(int x);

int main()
{
    std::packaged_task<int(int)> task(work);
    std::future<int> fut = task.get_future();

    std::thread t(std::move(task), 7); // run task on a thread
    int r = fut.get();                 // receive result message
    t.join();
}
```

80.5 Message passing with queues (channels)

80.5.1 The producer/consumer queue pattern

A queue is a multi-message channel:

- producers send messages by pushing values,
- consumers receive messages by popping values.

To be correct, the queue must:

- protect internal state with a mutex (or use atomics correctly),
- use condition variables with predicate waits (no waiting without a condition),
- define shutdown behavior (stop/drain).

80.5.2 A correct blocking queue (mutex + condition variable)

This queue is a practical message-passing channel. It:

- keeps mutex with the data,
- uses predicate waiting to handle spurious wakeups,
- supports cooperative shutdown.

```
#include <condition_variable>
#include <mutex>
#include <queue>
#include <optional>
```

```
template<class T>
class BlockingQueue {
public:
    void push(T v)
    {
        {
            std::lock_guard<std::mutex> lock(m_);
            q_.push(std::move(v));
        }
        cv_.notify_one();
    }

    // Returns std::nullopt when closed and empty.
    std::optional<T> pop()
    {
        std::unique_lock<std::mutex> lock(m_);
        cv_.wait(lock, [&]{ return closed_ || !q_.empty(); });

        if (q_.empty()) return std::nullopt; // closed and drained
        T v = std::move(q_.front());
        q_.pop();
        return v;
    }

    void close()
    {
        {
            std::lock_guard<std::mutex> lock(m_);
            closed_ = true;
```

```
        }
        cv_.notify_all();
    }

private:
    std::mutex m_;
    std::condition_variable cv_;
    std::queue<T> q_;
    bool closed_ = false;
};
```

80.5.3 Design rule: messages should be values

Prefer messages that are:

- movable and cheap to move,
- self-contained (no dangling pointers),
- explicit about ownership (use `unique_ptr` when transferring ownership).

80.6 Actor-style message passing (single-owner state)

80.6.1 Why actors simplify correctness

An actor owns its state and processes one message at a time. This ensures:

- actor state is not shared across threads,
- there is no need to lock actor state,
- communication is only via message passing.

This design aligns strongly with CP.3 and CP.4.

80.6.2 Minimal actor skeleton with `std::jthread` (C++20)

```
#include <condition_variable>
#include <mutex>
#include <queue>
#include <thread>
#include <stop_token>
#include <variant>
#include <future>

struct Add { int x; };
struct Get { std::promise<int> reply; };
using Msg = std::variant<Add, Get>;

class SumActor {
public:
    SumActor()
        : thr_([this](std::stop_token st){ loop(st); })
    {}

    void send(Msg m)
    {
        {
            std::lock_guard<std::mutex> lock(m_);
            q_.push(std::move(m));
        }
        cv_.notify_one();
    }
```

```
    std::future<int> request_sum()
    {
        std::promise<int> p;
        auto f = p.get_future();
        send(Get{std::move(p)});
        return f;
    }

    void stop()
    {
        thr_.request_stop();
        cv_.notify_all();
    }

    ~SumActor()
    {
        stop(); // jthread joins automatically
    }

private:
    void loop(std::stop_token st)
    {
        int sum = 0; // owned state
        while (!st.stop_requested()) {
            Msg msg;
            {
                std::unique_lock<std::mutex> lock(m_);
                cv_.wait(lock, [&]{ return st.stop_requested() || !q_.empty(); });
```

```
                if (st.stop_requested()) break;
                msg = std::move(q_.front());
                q_.pop();
            }

            if (auto* a = std::get_if<Add>(&msg)) {
                sum += a->x;
            } else if (auto* g = std::get_if<Get>(&msg)) {
                g->reply.set_value(sum); // reply message via future/promise
            }
        }
    }

    std::mutex m_;
    std::condition_variable cv_;
    std::queue<Msg> q_;
    std::jthread thr_;
};
```

Key property. The actor's state (`sum`) is accessed by one thread only.

80.7 Atomic wait/notify as a message-passing aid (C++20)

80.7.1 What it is

C++20 introduced atomic waiting and notification operations (e.g., `atomic::wait`, `notify_one`, `notify_all`). These can be used to build efficient message channels without spinning.

80.7.2 A simple “event” message

```
#include <atomic>
#include <thread>

std::atomic<int> event_count{0};

void sender()
{
    event_count.fetch_add(1, std::memory_order_release);
    event_count.notify_one();
}

void receiver()
{
    int old = event_count.load(std::memory_order_acquire);
    event_count.wait(old); // blocks until event_count != old
    // received at least one event
}
```

Note. This models a simple signal channel. For payload-carrying channels, use queues or futures unless you have a proven lock-free design.

80.8 Error propagation and failure semantics

80.8.1 Prefer “value or exception” channels

Futures naturally propagate exceptions from the producing task to the consuming thread at `get()`. This is a strong reason to prefer futures/promises for request/response message passing.

80.8.2 Queues need an explicit error strategy

If you use queues:

- define whether errors are messages (e.g., `variant<Payload, Error>`),
- define whether shutdown is graceful (drain) or immediate (drop),
- define what `close()` means (no new messages vs. cancel everything).

80.9 Shutdown and cancellation in message-passing systems

80.9.1 Why shutdown must be designed

Many concurrency failures occur during shutdown: dangling references, joining under locks, or workers blocked forever waiting for input.

80.9.2 Cooperative stop with `std::stop_token` (C++20)

`std::jthread` provides RAII joining and a built-in stop mechanism. This supports structured lifetimes in message-passing systems.

```
#include <thread>
#include <stop_token>
#include <chrono>

void run(std::stop_token st)
{
    while (!st.stop_requested()) {
        // process messages, or block waiting for them
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
```

```
    }
}

int main()
{
    std::jthread t(run);
    t.request_stop();
}
```

80.9.3 Queue close + stop request: a robust combination

A common robust design uses both:

- a stop request (global cancellation intent),
- closing the input channel (unblock receivers and allow draining/exit).

80.10 Modern direction: composable asynchronous message passing (`std::execution`)

WG21 P2300 proposes a standard framework for asynchronous execution using **senders/receivers** (`std::execution`), which is expected to be part of C++26 library direction and widely discussed as a major upcoming concurrency facility. Architecturally, senders/receivers generalize message passing:

- values flow through composed operations,
- completion and errors are explicit channels,
- cancellation is part of the model.

Even if your codebase does not yet use `std::execution`, adopting message-passing principles today reduces migration friction later.

80.11 Design checklist for message-passing components

- **No shared writable state** across threads except inside well-defined channels.
- **Messages are values** (move into the channel; avoid borrowed references).
- **Channel contract is explicit:** ordering, capacity/back-pressure, close semantics.
- **Error strategy is explicit:** exceptions via futures or error objects via messages.
- **Shutdown is explicit:** stop request + channel close; no detached threads.
- **No callbacks under locks:** if a channel uses locks internally, do not call unknown code while holding them (CP.22).

80.12 Summary

- Message passing is a primary technique to reduce data races and deadlocks by minimizing shared mutable state.
- Standard C++ supports message passing through `future`/`promise` (CP.60), `async` (CP.61), `packaged_task`, and queue-based channels.
- Correct channels require explicit contracts: ordering, shutdown, and error semantics.
- C++20 adds `jthread`/`stop_token` and atomic wait/notify, improving structured shutdown and efficient signaling.

- The direction of modern C++ concurrency continues toward composable asynchronous message passing (senders/receivers).

Part XIII

Performance (Per)

Chapter 81

Per.1 – Per.14

81.1 Scope and intent of the Performance rules

The Performance (Per) rules of the ISO C++ Core Guidelines target **high performance** and **low latency** code. They are intentionally more restrictive than what many applications need. Follow them when performance is a *real requirement*, and when you can *measure* and *validate* the impact.
Two principles apply throughout:

- **Correctness comes first.** In C++, data races are undefined behavior and can destroy performance measurements and program validity.
- **Performance is a feature you must justify and measure.** "Feels faster" is not a reliable engineering argument.

81.2 Per.1: Don't optimize without reason

81.2.1 Rule

Do not optimize unless you have a reason grounded in requirements, observations, or constraints.

81.2.2 Reason

Optimizing without need usually increases complexity, bugs, and maintenance cost while delivering no user-visible benefit.

81.2.3 Practical guidance

- Establish a performance goal (latency, throughput, memory, CPU, energy).
- Identify the critical operations and their acceptable limits.
- Keep code simple until measurement proves it is insufficient.

81.3 Per.2: Don't optimize prematurely

81.3.1 Rule

Avoid early micro-optimizations that make code larger, harder to change, or harder to reason about.

81.3.2 Reason

Heavily optimized code is often brittle: future feature changes can invalidate the optimization and preserve the complexity.

81.3.3 Practical guidance

- Prefer clear algorithms and good data structures first.
- Optimize only after you can reproduce a performance problem.
- Use compiler optimization reports and profiling before rewriting code.

81.4 Per.3: Don't optimize something that's not performance critical

81.4.1 Rule

Do not spend effort optimizing code that is not on the performance-critical path.

81.4.2 Reason

Improving non-critical code does not significantly improve overall system performance (Amdahl-style reality).

81.4.3 Practical guidance

- Use a profiler to identify hotspots (CPU time, allocations, contention).
- Focus on the top contributors first.
- Re-measure after each change; hotspots move.

81.5 Per.4: Don’t assume that complicated code is necessarily faster than simple code

81.5.1 Rule

Do not assume that a complex, low-level rewrite is faster than the straightforward version.

81.5.2 Reason

Modern compilers optimize simple, well-structured code extremely well. Complex tricks can inhibit optimization, introduce undefined behavior, or worsen memory behavior.

81.5.3 Example

```
#include <cstdint>
#include <vector>

void invert_bytes(std::vector<std::uint8_t>& v)
{
    for (auto& x : v) {
        x = static_cast<std::uint8_t>(~x);
    }
}
```

81.5.4 Guidance

- Prefer idiomatic loops and standard algorithms.
- Avoid type-punning and misaligned access tricks unless you can prove correctness and wins.

81.6 Per.5: Don't assume that low-level code is necessarily faster than high-level code

81.6.1 Rule

Prefer clear, type-safe, higher-level constructs when they express intent and enable optimization.

81.6.2 Reason

Low-level code can hide intent and remove information the compiler needs. High-level code can expose invariants and enable vectorization, inlining, and devirtualization.

81.6.3 Practical examples

- Prefer `std::sort` over `qsort` because it preserves type information, iterator category, and comparison semantics.
- Prefer `std::span` or iterators over raw pointer pairs when it improves clarity and bounds reasoning.

81.7 Per.6: Don't make claims about performance without measurements

81.7.1 Rule

Never assert performance improvements without measuring under realistic conditions.

81.7.2 Reason

Performance folklore is often wrong. Hardware, caches, branch predictors, allocators, and compilers can make results counterintuitive.

81.7.3 Measurement essentials

- Use `std::chrono::steady_clock` for timing.
- Warm up caches and avoid measuring one-time initialization unless that is the target.
- Separate microbenchmarks (tight loops) from system benchmarks (real workload).
- Measure variability (min/median/max), not a single number.

81.7.4 A minimal timing harness

```
#include <chrono>
#include <cstddef>
#include <iostream>

template<class F>
auto time_it(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();
    }
    auto stop = clock::now();
    return std::chrono::duration_cast<std::chrono::nanoseconds>(stop - start);
```

```
}

int main()
{
    volatile int sink = 0;
    auto d = time_it([&] { sink += 1; }, 10'000'000);
    std::cout << "ns: " << d.count() << "\n";
}
```

81.8 Per.7: Design to enable optimization

81.8.1 Rule

Design interfaces and data structures so future optimization is possible without breaking users.

81.8.2 Reason

A design that discards type information, shape information, or ownership constraints becomes hard to optimize later.

81.8.3 Guidance

- Prefer typed interfaces over `void*`.
- Expose enough information to allow efficient implementations (sizes, ranges, ordering).
- Keep representation details private (allow changing from list to vector later).
- Avoid “designing for everything” (overgeneralization often blocks optimization).

81.8.4 Example: avoid type-erasing C interfaces

```
// C-style: throws away information
void qsort(void* base, std::size_t n, std::size_t sz,
           int (*cmp)(const void*, const void*));

// C++ style: preserves element type and ordering
#include <algorithm>
#include <vector>

void better(std::vector<double>& v)
{
    std::sort(v.begin(), v.end());
}
```

81.9 Per.10: Rely on the static type system

81.9.1 Rule

Prefer statically typed code over weakly typed techniques (e.g., `void*`, byte-wise manipulation) when performance matters.

81.9.2 Reason

Weak typing increases the optimizer's uncertainty and can force pessimistic assumptions. Statically typed code enables inlining, specialization, and better alias analysis.

81.9.3 Guidance

- Prefer templates, `std::variant`, and concepts-based overloads to manual type tags plus `void*`.
- Prefer typed containers and algorithms to byte buffers unless you are implementing a low-level boundary.
- Keep invariants explicit (e.g., `enum class` for states instead of raw integers).

81.10 Per.11: Move computation from run time to compile time

81.10.1 Rule

When it improves performance and correctness, compute at compile time using `constexpr`, `consteval`, templates, and constant initialization.

81.10.2 Reason

Compile-time computation can reduce runtime work, improve safety (no races on constants), and move error detection to compilation.

81.10.3 Example: prefer compile-time initialization

```
constexpr double ntimes(double d, int n)   // assume n >= 0
{
    double m = 1.0;
    while (n--) m *= d;
    return m;
```

```
}

constexpr double s3 = ntimes(2.0, 3); // computed at compile time
```

81.10.4 Important trade-offs

- Excessive compile-time computation can increase build time and complicate debugging.
- Not everything belongs at compile time; use it where it provides real value.

81.11 Per.12: Eliminate redundant aliases

81.11.1 Rule

Avoid designs where the same mutable object is reachable through many independent aliases, especially across call boundaries.

81.11.2 Why it matters

Aliasing is not automatically "bad," but **unnecessary aliasing** makes reasoning harder and can restrict optimization. When the compiler must assume that two pointers/references may refer to the same object, it may reorder fewer operations safely.

81.11.3 Guidance (safe, portable)

- Prefer passing `span<T>` or iterators that clearly define the range being modified.
- Prefer returning values rather than exposing internal pointers/references.
- Keep mutation localized: minimize the number of simultaneously-live references to the same mutable state.

- When a function mutates one region and reads another, keep those regions distinct by design.

81.11.4 Example: clarify which range is modified

```
#include <span>
#include <cstddef>

void scale(std::span<float> out, std::span<const float> in, float k)
{
    // Design intent: out is written, in is read
    // Keep caller responsibility: out and in should not overlap.
    for (std::size_t i = 0; i < out.size(); ++i) {
        out[i] = in[i] * k;
    }
}
```

81.11.5 Note

If overlap is allowed, document it and implement accordingly. If overlap is forbidden, state it as a precondition and consider runtime checks in debug builds.

81.12 Per.13: Eliminate redundant indirections

81.12.1 Rule

Avoid unnecessary layers of pointer chasing, dynamic dispatch, and heap indirection on performance-critical paths.

81.12.2 Why it matters

Each indirection can cost:

- extra loads and cache misses,
- harder branch prediction (virtual calls),
- less inlining and less constant propagation.

81.12.3 Guidance

- Prefer contiguous storage (`std::vector`, `std::string`) for hot data.
- Prefer values and small objects by value when appropriate.
- Use polymorphism where it expresses real variation, but avoid virtual dispatch in tight inner loops when alternatives exist (e.g., templates, `std::variant`, function objects).

81.12.4 Example: replace pointer chasing with contiguous storage when possible

```
#include <vector>
#include <cstdint>

struct Node {
    std::uint32_t value;
    std::uint32_t next; // index into a vector, not a pointer
};

std::uint64_t sum_list(const std::vector<Node>& nodes, std::uint32_t head)
{
```

```
    std::uint64_t sum = 0;
    for (auto i = head; i != UINT32_MAX; i = nodes[i].next) {
        sum += nodes[i].value;
    }
    return sum;
}
```

81.13 Per.14: Minimize the number of allocations and deallocations

81.13.1 Rule

Reduce dynamic memory allocation on performance-critical paths.

81.13.2 Why it matters

Allocations and deallocations can be expensive and unpredictable due to:

- allocator bookkeeping,
- synchronization in multi-threaded allocators,
- fragmentation and cache effects,
- latency spikes.

81.13.3 Guidance

- Prefer stack allocation for short-lived objects.
- Use `std::vector::reserve` when the final size is known or can be estimated.

- Reuse buffers and objects where it does not complicate correctness.
- Prefer `std::string_view` to avoid temporary string allocations when you only need a view.
- For allocation-heavy subsystems, consider polymorphic allocators (`std::pmr`) to control allocation strategy centrally.

81.13.4 Example: avoid repeated reallocation

```
#include <vector>
#include <string>

std::vector<std::string> build(std::size_t n)
{
    std::vector<std::string> v;
    v.reserve(n); // Per.14: reduce allocations
    for (std::size_t i = 0; i < n; ++i) {
        v.emplace_back("item");
    }
    return v;
}
```

81.13.5 Example: `std::pmr` for controlled allocation

```
#include <memory_resource>
#include <string>
#include <vector>

std::pmr::vector<std::pmr::string> build_pmr(std::pmr::memory_resource* mr,
↪  std::size_t n)
```

```
{
    std::pmr::vector<std::pmr::string> v{mr};
    v.reserve(n);
    for (std::size_t i = 0; i < n; ++i) {
        v.emplace_back("item", mr);
    }
    return v;
}
```

81.14 Putting Per.1–Per.14 into a single workflow

A reliable workflow consistent with these rules is:

1. Define what performance means for the system (latency/throughput/memory).
2. Measure baseline (Per.6).
3. Identify hotspots (Per.3).
4. Improve design to enable optimization (Per.7), keep code simple (Per.4, Per.5).
5. Apply typed, optimizer-friendly solutions (Per.10).
6. Reduce runtime work with compile-time techniques when appropriate (Per.11).
7. Reduce aliasing and indirection where it blocks performance (Per.12, Per.13).
8. Reduce allocation pressure (Per.14).
9. Re-measure, validate, and keep changes only if they help (Per.6).

Chapter 82

When to Optimize

82.1 Purpose

Optimization is not a random activity or a habit. In the ISO C++ Core Guidelines, performance work is treated as an **engineering process** driven by requirements and validated by measurement. The goal is to improve user-visible outcomes (latency, throughput, memory, energy) *without* damaging correctness, maintainability, or long-term evolution.
This chapter explains when optimization is justified, how to decide *what* to optimize, and how to optimize safely and verifiably in modern C++.

82.2 Core message from the Performance rules (Per)

The first performance rules define the policy:

- **Per.1:** Don't optimize without reason.
- **Per.2:** Don't optimize prematurely.

- **Per.3:** Don't optimize something that's not performance critical.
- **Per.6:** Don't make claims about performance without measurements.

If you follow these four consistently, most "performance work" becomes straightforward and defensible.

82.3 When optimization is *actually* needed

82.3.1 Trigger 1: a defined performance requirement exists

Optimization is justified when you have a requirement that can be stated and tested, such as:

- P99 latency below a threshold (e.g., `P99 < 10 ms`),
- throughput above a threshold (e.g., `> 200k ops/s`),
- memory usage below a limit,
- startup time below a limit,
- power/energy budget constraints (embedded and mobile).

Without such a requirement, optimization is usually just complexity. This is the practical meaning of Per.1 and Per.2.

82.3.2 Trigger 2: measurements show a real problem

Sometimes you do not begin with a strict numeric target, but telemetry or profiling proves a real issue:

- timeouts under load,

- CPU pegged in production,
- high allocation rate causing stalls,
- lock contention causing throughput collapse,
- cache-miss dominated hot loops.

This still counts as a “reason” (Per.1), but it must be supported by measurement (Per.6).

82.3.3 Trigger 3: a scalability bottleneck is proven

A typical scalability rule: improving a part of the program matters only in proportion to how often it is used. This is the reasoning behind focusing on hotspots (Per.3) and is commonly summarized by Amdahl’s law.

82.4 When optimization is *not* needed

82.4.1 No measurable bottleneck

If profiling shows your code is not on a critical path, optimizing it rarely matters (Per.3).

82.4.2 The system is I/O bound or externally limited

If the dominant time is network I/O, disk I/O, or waiting on an external service, micro-optimizing CPU code usually has little impact. Improve the dominant constraint first (e.g., batching, caching, reducing round-trips).

82.4.3 You cannot test or reproduce performance

If the workload cannot be reproduced or measured reliably, optimization claims are not engineering (Per.6). Stabilize the benchmark or create instrumentation first.

82.5 A practical decision workflow

82.5.1 Step 1: State the performance goal

Write the goal in a measurable form:

- metric (latency/throughput/memory),
- percentile if relevant (P50/P95/P99),
- environment (CPU model, core count, OS, compiler),
- workload definition (input sizes, distributions).

82.5.2 Step 2: Establish a baseline

Measure the current system and record:

- build configuration (debug vs release),
- compiler version and flags,
- input dataset or generation seed,
- run-to-run variability.

Without a baseline, you cannot prove improvement (Per.6).

82.5.3 Step 3: Find the bottleneck (Per.3)

Use profiling and counters to locate the top contributors:

- CPU time hotspots,

- allocation hotspots,
- contention hotspots (locks/atomics),
- cache misses / branch mispredictions (when appropriate).

Then choose the smallest change that can improve the dominant bottleneck.

82.5.4 Step 4: Optimize with minimal complexity (Per.4, Per.5)

The guidelines explicitly warn against assuming that:

- complicated code is faster than simple code (Per.4),
- low-level tricks are faster than high-level code (Per.5).

Start with algorithm and data-structure improvements, because they often deliver the biggest gains with the least risk.

82.5.5 Step 5: Re-measure and keep only wins (Per.6)

After each change:

- re-run the same benchmark,
- compare against baseline,
- check regression risk (correctness tests, sanitizer builds),
- keep the change only if it improves the goal.

This is the operational meaning of Per.6.

82.6 Measurement fundamentals you must respect (Per.6)

82.6.1 Use a monotonic clock for timing

For interval measurement, prefer `std::chrono::steady_clock`, which is monotonic and suitable for measuring durations.

82.6.2 A minimal, reproducible timing harness

This example is intentionally simple and demonstrates a safe clock choice and repeated iterations.

```
#include <chrono>
#include <cstddef>
#include <iostream>

template<class F>
std::chrono::nanoseconds time_it(F&& f, std::size_t iters)
{
    using clock = std::chrono::steady_clock; // monotonic for intervals
    const auto start = clock::now();
    for (std::size_t i = 0; i < iters; ++i) {
        f();
    }
    const auto stop = clock::now();
    return std::chrono::duration_cast<std::chrono::nanoseconds>(stop - start);
}

int main()
{
```

```
    volatile int sink = 0; // prevents trivial removal in this small example
    auto d = time_it([&]{ sink += 1; }, 10'000'000);
    std::cout << "ns: " << d.count() << "\n";
}
```

82.6.3 Avoid common benchmarking mistakes

- Measuring debug builds when release performance is the target.
- Measuring one-time initialization when steady-state performance is the target.
- Measuring too few iterations (noise dominates).
- Comparing two versions with different compiler flags or different input distributions.
- Ignoring variability (always run multiple trials).

Per.6 exists because these mistakes are extremely common.

82.7 What to optimize first (highest return on effort)

82.7.1 1) Algorithmic complexity

Changing $O(n^2)$ to $O(n \log n)$ often dwarfs micro-optimizations. This also tends to preserve maintainability.

82.7.2 2) Data structures and memory layout

In performance-critical code, memory behavior can dominate:

- contiguous storage improves locality,

- fewer allocations reduces latency spikes,
- smaller working set fits caches better.

This aligns with Per.14 (minimize allocations) and the general Per theme of enabling optimization by design.

82.7.3 3) Allocation rate and object lifetime

If profiling shows heavy allocation pressure, reduce allocations using:

- `reserve()` and reuse,
- value types and move semantics,
- pooling or polymorphic allocators (`std::pmr`) when justified.

82.7.4 4) Concurrency contention

If the bottleneck is contention:

- reduce shared writable state,
- shorten critical sections,
- shard locks or use per-thread aggregation.

82.8 How to decide between readability and performance

82.8.1 Default rule

Prefer the simplest correct implementation until measurement proves it is insufficient (Per.1, Per.2, Per.6).

82.8.2 When a faster version is acceptable

A more complex optimization is justified when:

- it improves a declared performance goal,
- it is covered by tests (including regression tests),
- it is localized (does not infect the whole design),
- it is documented (what was measured, what improved, and why the technique is safe).

82.8.3 Keep optimized code maintainable

When optimized code is needed:

- keep the optimized path behind a clean interface,
- keep a readable reference implementation for clarity where appropriate,
- add comments describing invariants and why the change works.

82.9 Checklist: “Should we optimize now?”

You should optimize now only if all of these are true:

1. There is a measurable performance requirement or a measured production bottleneck. (Per.1)
2. You can reproduce the workload and measure reliably. (Per.6)
3. You have identified the performance-critical path via profiling. (Per.3)
4. You can describe the expected win and validate it after the change. (Per.6)

5. The change preserves correctness and stays maintainable. (Per.2, Per.4, Per.5)

Otherwise, postpone optimization and invest in clarity, tests, and observability.

82.10 Summary

- Optimize only with a real reason (Per.1) and not prematurely (Per.2).
- Focus on what is performance critical (Per.3).
- Validate every claim with measurement (Per.6).
- Use `std::chrono::steady_clock` for timing intervals in benchmarks.
- Prefer high-return improvements first: algorithms, data structures, memory behavior, and contention reduction.

Chapter 83

Measure Before Optimizing

83.1 Purpose

Optimization without measurement is speculation. Real performance work must be **driven by evidence**: you must quantify current behavior, identify bottlenecks, and only then apply changes that measurably improve the metric you care about. The ISO C++ Core Guidelines emphasize this principle because unmeasured optimization often wastes time and introduces complexity without delivering benefits.

83.2 Why Measurement Matters

83.2.1 Avoid wasted effort

Without measurement you cannot know whether:

- a piece of code is actually on the critical path,
- a change improves or degrades overall performance,

- a bottleneck is caused by CPU, memory, I/O, or contention,
- your optimization technique is effective on real workloads.

83.2.2 Prevent interfering effects

Modern systems have many overlapping costs:

- compiler optimizations,
- caches and branch predictors,
- library implementations,
- operating system scheduling,
- hardware variability.

Only measurement under controlled conditions reveals true behavior.

83.3 The Guiding Rule: Per.6

83.3.1 Statement of the rule

Per.6. Do not make claims about performance without measurements.

83.3.2 What this means in practice

- Always quantify performance before and after optimization.
- Avoid relying on intuition, anecdote, or language folklore.
- Use metrics that match your performance goals (latency, throughput, memory, energy).

83.4 Types of Measurement

83.4.1 Profiling

Profilers tell you where time or resources are spent:

- **Sampling profilers** periodically record the instruction pointer to find hotspots without intrusive instrumentation.
- **Instrumenting profilers** record every function call or event, useful for detailed path analysis but more intrusive.

Use profiling to locate the true hotspot before optimization.

83.4.2 Benchmarking

Benchmarking quantifies performance of a particular function or workload under defined conditions:

- microbenchmarks measure a small piece of code,
- macrobenchmarks validate performance at the subsystem or system level.

Benchmarks must be repeatable, isolated, and measurable.

83.5 Principles of Good Measurement

83.5.1 Define the metric clearly

Before measuring, specify:

- what quantity you measure (e.g., latency, throughput, memory),

- units (e.g., milliseconds, operations per second),
- workload characteristics (size, distribution, concurrency),
- environment (hardware, OS, compiler flags).

83.5.2 Use a stable clock source

When timing code, use a monotonic clock that does not jump due to system clock adjustments. In C++, the appropriate choice is `std::chrono::steady_clock`, which is guaranteed not to go backwards or jump.

83.5.3 Isolate the code under test

To measure a specific component:

- avoid unrelated work in the measurement loop,
- isolate variability sources (other processes, network noise),
- measure in conditions representative of real use when possible.

83.5.4 Warm up caches and environment

CPU caches, branch predictors, dynamic memory allocators, and JIT-like behaviors require stabilization before measurement:

- run the workload multiple times before recording metrics,
- discard first-run anomalies where appropriate.

83.5.5 Run multiple iterations

A single measurement rarely represents typical performance. Collect multiple samples and analyze:

- minimum (best-case),
- median (typical),
- maximum (worst-case),
- variance or standard deviation.

83.5.6 Control for measurement noise

Measurement noise can come from:

- background OS activity,
- interrupts,
- CPU frequency scaling,
- other concurrently running applications.

Use a quiet environment and repeat runs to reduce noise influence.

83.6 A Simple Measurement Framework in C++

```
#include <chrono>
#include <cstddef>
#include <vector>
```

```cpp
template<typename F>
auto measure_repeated(F&& work, std::size_t iterations)
{
    using clock = std::chrono::steady_clock;
    std::vector<std::chrono::nanoseconds> results;
    results.reserve(iterations);

    for (std::size_t i = 0; i < iterations; ++i) {
        auto start = clock::now();
        work();
        auto end = clock::now();
        results.push_back(end - start);
    }
    return results; // analyze outside: min/median/max/variance
}
```

83.6.1 Analyzing the results

After collecting timing samples:

- sort the results,
- report relevant statistics (median, percentiles),
- compare before and after changes on the **same metric** and under the **same conditions**.

83.7 Microbenchmarks vs. System Benchmarks

83.7.1 Microbenchmarks

Microbenchmarks focus on a small piece of code. They are useful for understanding low-level behaviors and comparing alternatives. Their limitations:

- they may not reflect real workloads,
- they may remove context that matters at system level,
- they can mislead if not designed carefully.

83.7.2 System Benchmarks

System benchmarks measure performance in conditions closer to production. They are harder to control and isolate but provide meaningful insight into end-to-end behavior.

83.8 Common Measurement Pitfalls

83.8.1 Measuring debug builds

Debug builds include additional checks and disabled optimizations. Always benchmark release builds compiled with `-O2` or equivalent optimization settings.

83.8.2 Measuring initialization rather than steady state

Many systems have a warm-up phase (e.g., JIT-like caching or dynamic data structure growth). Decide whether your metric targets cold-start or steady-state performance, and measure accordingly.

83.8.3 Changing code between runs

Even small changes (including adding logging or I/O) can invalidate measurement. Keep the code under test as consistent as possible across runs.

83.8.4 Misinterpreting variability

High variance in results usually indicates:

- insufficient iteration count,
- interference from the environment,
- unstable workload conditions.

Use statistical approaches (median, percentiles) rather than single samples.

83.9 Profile Before You Optimize

83.9.1 Why profiling matters

Profiling reveals where the program actually spends time or resources:

- function-level hotspots,
- call graph bottlenecks,
- memory allocation hot paths,
- contention on locks or atomics.

83.9.2 Types of profiling useful for C++

- CPU sampling profilers for time-dominant code paths,
- allocation profilers for heap pressure,
- concurrency profilers for lock contention and thread stalls,
- hardware counter-based profilers for cache and branch behavior.

83.10 Using Profilers to Guide Optimization

83.10.1 Interpret results carefully

Profilers approximate where time is spent; avoid overinterpreting minor differences. Focus on:

- the largest contributors to total time,
- patterns that persist over many runs,
- hotspots that dominate the performance-critical path.

83.10.2 Iterative approach

Optimization should be iterative:

- profile the baseline,
- identify a candidate hotspot,
- optimize and re-profile,
- repeat until requirements are met or returns diminish.

83.11 Validating the Impact of Optimization

83.11.1 Re-measure after each change

Always compare new measurements to your established baseline. Use the same harness, workload, and environment to ensure comparability.

83.11.2 Check for unintended regressions

Performance changes can improve one metric but degrade others (e.g., faster CPU time but higher memory use). Track multiple metrics when relevant.

83.11.3 Maintain correctness under optimization

Performance changes must preserve correctness. Regression tests and sanitizers (memory, thread) help ensure that optimization does not introduce bugs.

83.12 When Optimized Code Should Be Reviewed Differently

Optimized code often carries:

- more complex logic,
- lower-level constructs,
- hardware-sensitive assumptions.

Such code deserves:

- dedicated performance-focused code review,
- clear documentation of why the change matters,

- linkage to measured goals and quantifiable impact.

83.13 Summary

- Measurement must precede optimization. Optimization without measurement is guesswork.
- Use stable clocks (`std::chrono::steady_clock`) and repeated runs to collect meaningful data.
- Distinguish microbenchmarks from system benchmarks; use both as appropriate.
- Profile to find hotspots before attempting optimization.
- Re-measure after each optimization and validate impact against the baseline.
- Avoid common pitfalls such as measuring debug builds or noisy environments.

Chapter 84

Cost Models

84.1 Purpose

A **cost model** is a practical way to predict where time and memory go before you run a profiler. It is not a replacement for measurement; it is a disciplined method to:

- choose algorithms and data structures with the right asymptotic behavior,
- avoid accidental expensive operations (allocations, cache misses, contention),
- design APIs whose performance properties are predictable and stable,
- decide what to measure first (hot paths and high-frequency operations).

In the ISO C++ Core Guidelines, the Performance rules start with the principle that performance claims must be backed by measurement (not intuition). A good cost model is what you use *before* measuring to form correct hypotheses and avoid wasting time.

84.2 Cost model foundations for modern C++

84.2.1 Two levels of cost

A useful cost model has two layers:

1. **Algorithmic cost:** how work scales with input size (Big-O, amortized complexity).
2. **Machine cost:** constant factors driven by memory hierarchy, allocations, branches, and synchronization.

Big-O selects the right family of solutions; machine costs explain why two solutions with the same Big-O can differ by 5–50x in practice.

84.2.2 The "trust but verify" rule (Per)

Cost models guide decisions, but you must confirm with measurements:

- **Per.1–Per.3:** optimize only with reason and only on critical paths,
- **Per.6:** do not make performance claims without measurements.

84.3 Algorithmic cost models you must use

84.3.1 Big-O as a design filter

Big-O is most valuable when it prevents catastrophic growth:

- prefer $O(n \log n)$ sorting over $O(n^2)$ approaches for large n,
- avoid repeated linear searches in a growing container when a hash map or sorting would be appropriate.

84.3.2 Standard library complexity guarantees are part of the model

The standard library provides complexity guarantees that should be part of your mental cost model:

- `std::sort` provides $O(N \log N)$ in the worst case.
- `std::vector` provides amortized constant-time insertion/removal at the end, and linear-time insertion/removal in the middle.
- `std::unordered_map` offers average constant-time search/insert/erase (with important caveats in worst cases and rehashing behavior).

84.3.3 Amortized complexity is real cost, not a trick

Many operations are cheap most of the time, but occasionally expensive:

- `vector::push_back` can trigger reallocation and invalidate references/iterators when capacity grows.
- appropriate use of `reserve()` can eliminate many reallocations, but misusing `reserve()` can *harm* complexity and performance.

84.4 Machine-level cost models that dominate real performance

84.4.1 Memory hierarchy and locality

On modern CPUs, memory access patterns can dominate execution time:

- contiguous storage (like `std::vector`) improves cache locality and prefetching,

- pointer chasing in node-based structures increases cache misses and latency.

This is why “compact data” and “predictable access” themes appear repeatedly in performance guidance, and why Per rules often favor removing indirections and allocations.

84.4.2 Allocation cost and variability

Dynamic allocation is expensive and can be unpredictable due to:

- allocator bookkeeping,
- fragmentation,
- synchronization in multi-threaded allocators,
- cache and TLB effects.

The Core Guidelines explicitly recommend minimizing allocations (Per.14).

84.4.3 Branching and predictability

Branches that are hard to predict cause pipeline stalls. Cost models should treat:

- tight loops with unpredictable branches as expensive,
- branchless or predictable-branch alternatives as potentially faster,

but only after measurement confirms the bottleneck (Per.6).

84.4.4 Synchronization and contention

In concurrent code, the cost model must include:

- lock acquisition and wakeups,

- cache-line bouncing caused by contended atomics,
- context switches and scheduler behavior.

A small critical section can still be expensive if many threads contend on it.

84.5 API-level cost models (what your interfaces promise)

A strong performance design makes costs visible and predictable at the interface.

84.5.1 Cost transparency: cheap operations should look cheap

Prefer APIs where the caller can see likely costs:

- **Non-owning views** for read-only access to existing data (e.g., `string_view`, `span`),
- **Value return** when move semantics and NRVO make it cheap and safe,
- **Explicit ownership transfer** using `unique_ptr` for heap objects.

84.5.2 Example: `std::string` to `std::string_view` is a cheap view

C++17 provides conversion from `std::basic_string` to `std::basic_string_view` that constructs a view from `data()` and `size()`, which is typically constant-time.

```
#include <string>
#include <string_view>

void consume(std::string_view sv);

int main()
{
```

```
    std::string s = "hello";
    consume(s); // passes a view (no copy of characters)
}
```

Cost model note. The view is cheap, but it does not own the data. The lifetime of the referenced characters must exceed the view's use.

84.5.3 Value vs. reference: the cost model depends on size, ownership, and lifetime

There is no universal rule that references are always cheaper:

- small trivially-copyable types are often best passed by value,
- large objects may be best passed by reference or view,
- returning by value is often efficient due to move semantics and copy elision.

The correct decision is "cost model + measurement," not habit.

84.6 Container and algorithm cost model examples

84.6.1 Example 1: reserve to control reallocation cost

```
#include <vector>
#include <cstddef>

std::vector<int> build(std::size_t n)
{
    std::vector<int> v;
    v.reserve(n); // reduces reallocations when n is known
```

```
    for (std::size_t i = 0; i < n; ++i) {
        v.push_back(static_cast<int>(i));
    }
    return v;
}
```

This leverages the cost model that reallocation is costly and avoidable when the final size is predictable.

84.6.2 Example 2: beware of reference/iterator invalidation on growth

```
#include <vector>

int main()
{
    std::vector<int> v;
    v.push_back(1);
    int& r = v[0];

    // A later push_back may reallocate and invalidate r:
    v.push_back(2);

    // Using r here can be wrong if reallocation happened.
}
```

Cost model note. Growth may cause reallocation; treat references/iterators as invalid unless capacity is stable.

84.6.3 Example 3: choosing between `unordered_map` and sorting

If you need many lookups:

- `unordered_map`: average constant-time lookup, but more allocations and worse locality,
- sorting once: $O(N \log N)$ cost, then efficient binary searches with good locality.

Your cost model must include both complexity and machine costs.

84.7 Building a cost model checklist for performance work

When you analyze a piece of code, answer these questions *before* you optimize:

1. **What is the unit of work?** (per request, per element, per frame)
2. **What scales with input size?** Identify $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$.
3. **Where are allocations?** How many per unit of work? Can they be eliminated? (Per.14)
4. **What is the memory access pattern?** Contiguous vs. pointer-chasing; predictable vs. random.
5. **What is the synchronization pattern?** Any contended locks/atomics? Any waits?
6. **What does the interface promise?** Does it hide expensive work (copying, formatting, allocation)?
7. **What will you measure?** Define metric(s) and baseline (Per.6).

84.8 How cost models interact with measurement (the correct workflow)

A correct performance workflow is:

1. Build a cost model (hypothesize the bottleneck).

2. Measure and profile to confirm the real hotspot (Per.3, Per.6).
3. Make the smallest change that improves the dominant cost.
4. Re-measure; keep changes only if the metric improves (Per.6).

Cost models prevent random optimization; measurement prevents self-deception.

84.9 Summary

- A cost model is your disciplined prediction of performance costs.
- Use both **algorithmic costs** (Big-O, amortized complexity) and **machine costs** (locality, allocation, branching, contention).
- Standard library complexity guarantees are part of the cost model (e.g., `sort`, `vector`, `unordered_map`).
- The Core Guidelines require measurement to validate performance claims (Per.6) and recommend minimizing allocations (Per.14).

Chapter 85

Cache-Friendly Code

85.1 Purpose

On modern machines, **performance is often dominated by memory access**, not arithmetic. Code that repeatedly accesses memory in unpredictable patterns can be many times slower than code that accesses the same data sequentially. The ISO C++ Core Guidelines explicitly highlight cache sensitivity and recommend predictable, linear access to adjacent data.
This chapter explains what "cache-friendly" means, how to design cache-friendly data layouts and loops in modern C++, and how to validate improvements by measurement.

85.2 What caches are and why they dominate performance

85.2.1 The key idea: the CPU is much faster than memory

Modern CPUs execute operations in a few cycles, but fetching data from main memory can take far longer. Caches mitigate this by keeping recently used data close to the CPU. When your code has good locality, it hits in cache; when it does not, it stalls.

85.2.2 Two kinds of locality

A practical cost model for cache friendliness uses:

- **Spatial locality:** access nearby addresses (e.g., sequential array traversal).
- **Temporal locality:** reuse the same data soon (e.g., repeated work on a small working set).

Cache-friendly code increases both.

85.2.3 The Core Guidelines performance message

The Performance section emphasizes memory behavior:

- **Per.18:** Space is time (memory layout and footprint matter).
- **Per.19:** Access memory predictably (linear access to adjacent data is favored).

85.3 The main rules of cache-friendly design

85.3.1 Rule 1: Prefer contiguous storage for hot data

Contiguous storage improves cache line utilization and hardware prefetching. In standard C++, `std::vector<T>` stores elements contiguously (except `std::vector<bool>` specialization).

85.3.1.1 Example: contiguous storage with `std::vector`

```
#include <vector>
#include <cstddef>

double sum(const std::vector<double>& v)
```

```
{
    double s = 0.0;
    for (std::size_t i = 0; i < v.size(); ++i) {
        s += v[i]; // predictable linear access
    }
    return s;
}
```

85.3.2 Rule 2: Minimize pointer chasing on critical paths

Node-based structures (linked lists, trees with heap-allocated nodes) often require following pointers to unrelated locations in memory. This destroys locality and prefetching. Prefer data-oriented structures where the critical traversal is mostly linear.

85.3.2.1 Example: replace pointer-based “next” with index-based adjacency

```
#include <cstdint>
#include <vector>

struct Node {
    std::uint32_t value;
    std::uint32_t next; // index into a contiguous array of nodes
};

std::uint64_t sum_indexed_list(const std::vector<Node>& nodes, std::uint32_t head)
{
    std::uint64_t s = 0;
    for (std::uint32_t i = head; i != UINT32_MAX; i = nodes[i].next) {
        s += nodes[i].value;
    }
```

```
    return s;
}
```

This is still a “linked” structure logically, but the nodes are stored contiguously, improving locality.

85.3.3 Rule 3: Iterate in the order the data is laid out

C and C++ store multi-dimensional arrays in row-major order. Accessing a matrix row-by-row is usually cache-friendly; column-by-column often is not.

85.3.3.1 Example: row-major traversal

```
#include <cstddef>
#include <vector>

double sum_row_major(const std::vector<double>& a, std::size_t rows, std::size_t
↪  cols)
{
    double s = 0.0;
    for (std::size_t r = 0; r < rows; ++r) {
        for (std::size_t c = 0; c < cols; ++c) {
            s += a[r * cols + c]; // contiguous within each row
        }
    }
    return s;
}
```

85.3.3.2 Anti-pattern: column-major access on row-major data

```
#include <cstddef>
```

```
#include <vector>

double sum_bad_column_major(const std::vector<double>& a, std::size_t rows,
↪  std::size_t cols)
{
    double s = 0.0;
    for (std::size_t c = 0; c < cols; ++c) {
        for (std::size_t r = 0; r < rows; ++r) {
            s += a[r * cols + c]; // large stride, poor locality
        }
    }
    return s;
}
```

This violates the predictable, adjacent-access principle emphasized by Per.19.

85.3.4 Rule 4: Use blocking (tiling) for large working sets

When data does not fit in cache, restructure computation to reuse a small block repeatedly before moving on. This increases temporal locality.

85.3.4.1 Example: blocked matrix multiply sketch

This is a simplified sketch to illustrate the pattern. The core idea is to work on sub-blocks.

```
#include <cstddef>
#include <vector>
#include <algorithm>

void matmul_blocked(const std::vector<double>& A,
                    const std::vector<double>& B,
```

```
                    std::vector<double>& C,
                    std::size_t N,
                    std::size_t BS)
{
    auto idx = [N](std::size_t r, std::size_t c) { return r * N + c; };

    for (std::size_t ii = 0; ii < N; ii += BS) {
        for (std::size_t kk = 0; kk < N; kk += BS) {
            for (std::size_t jj = 0; jj < N; jj += BS) {

                const std::size_t i_end = std::min(ii + BS, N);
                const std::size_t k_end = std::min(kk + BS, N);
                const std::size_t j_end = std::min(jj + BS, N);

                for (std::size_t i = ii; i < i_end; ++i) {
                    for (std::size_t k = kk; k < k_end; ++k) {
                        const double a = A[idx(i, k)];
                        for (std::size_t j = jj; j < j_end; ++j) {
                            C[idx(i, j)] += a * B[idx(k, j)];
                        }
                    }
                }
            }
        }
    }
}
```

Blocking is a practical way to turn many cache-miss-dominated accesses into cache-hit-dominated accesses, consistent with the “memory access times dominate” guidance in the performance rules.

85.3.5 Rule 5: Reduce allocations and shrink the working set

Allocations increase overhead and often scatter objects in memory, harming locality. The Core Guidelines recommend minimizing allocations on performance-critical paths (e.g., Per.14). A smaller working set (less memory touched per unit of work) increases cache hit rate and reduces TLB and cache pressure.

85.3.6 Rule 6: Arrange hot data together and cold data apart

If a struct contains a few frequently used fields and many rarely used fields, mixing them forces the CPU to fetch cold fields into cache lines, increasing memory traffic. A common design approach:

- keep hot fields in a compact struct,
- keep cold fields in a separate struct accessed only when needed.

The Core Guidelines also mention layout decisions such as putting the most used members first in a time-critical struct (e.g., Per.17).

85.3.6.1 Example: split hot/cold fields

```
#include <cstdint>
#include <string>

struct ColdUserData {
    std::string biography;
    std::string address;
};

struct UserHot {
```

```
    std::uint32_t id;
    std::uint32_t flags;
    double score;
    ColdUserData* cold; // accessed rarely
};
```

85.4 Data layout choices: AoS vs SoA

85.4.1 Array of Structures (AoS)

Good when you typically process all fields of each object together.

85.4.2 Structure of Arrays (SoA)

Good when you process one or two fields across many objects (common in numeric and simulation code). SoA often improves vectorization and cache line utilization.

85.4.2.1 Example: SoA for field-wise processing

```
#include <vector>
#include <cstddef>

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

void integrate_x(ParticlesSoA& p, float dt)
{
    for (std::size_t i = 0; i < p.x.size(); ++i) {
```

```
        p.x[i] += p.vx[i] * dt; // linear, field-wise
    }
}
```

This design aligns with Per.19 (predictable access) and benefits from contiguous storage guarantees of `std::vector`.

85.5 Cache friendliness in concurrent code: avoid false sharing

85.5.1 What false sharing is

False sharing happens when multiple threads frequently write to different variables that happen to share the same cache line. Even though the variables are logically independent, the cache coherence protocol forces cache lines to bounce between cores, hurting performance.

85.5.2 A common pattern to avoid

Per-thread counters stored adjacent in memory can cause false sharing.

85.5.2.1 Example: pad per-thread counters (conceptual)

```
#include <atomic>
#include <cstddef>
#include <vector>

struct alignas(64) Counter {
    std::atomic<std::uint64_t> value{0};
    // Padding is implicit due to alignment; size may still vary by ABI.
```

```
};

void increment_many(std::vector<Counter>& c, std::size_t tid, std::size_t n)
{
    for (std::size_t i = 0; i < n; ++i) {
        c[tid].value.fetch_add(1, std::memory_order_relaxed);
    }
}
```

Note. `alignas(64)` is a common choice on many systems, but the actual cache line size is platform-dependent. You should measure and tune for your target environment.

85.6 Practical do's and don'ts

85.6.1 Do

- Use contiguous containers for hot data (`std::vector`, `std::string`, `std::span` views).
- Traverse memory linearly and predictably (Per.19).
- Use blocking/tiling for large arrays and matrix-like computations.
- Reduce allocations and shrink the working set (Per.14, Per.18).
- Measure with realistic workloads and repeat runs (Per.6).

85.6.2 Don't

- Do not assume "clever" pointer tricks improve cache behavior; they often make it worse.
- Do not use linked lists for performance-critical sequential traversals unless measurement proves it helps.

- Do not add manual prefetching unless profiling and counters show memory stalls dominate and the access pattern is suitable.
- Do not optimize without measurements (Per.6).

85.7 How to validate cache-friendly changes

85.7.1 Measure the right thing

Cache-friendly changes often improve:

- throughput (operations per second),
- tail latency (P95/P99),
- CPU cycles per operation.

85.7.2 Use evidence-based workflow (Per)

1. Build a hypothesis: “this code is cache-miss dominated”.
2. Change layout or traversal to improve locality (Per.19, Per.18).
3. Measure before/after under the same conditions (Per.6).

85.8 Summary

- Cache-friendly code is mostly about **predictable access to adjacent data** and **small working sets**.
- Prefer contiguous storage for hot data; `std::vector<T>` provides contiguous storage (except `vector<bool>`).

- Iterate in memory order, and use blocking/tiling for large datasets.
- Reduce allocations and separate hot from cold data (Per.18, Per.14).
- In concurrent code, watch for false sharing and cache-line ping-pong.
- Always confirm improvements by measurement (Per.6).

Part XIV

The Standard Library (SL)

Chapter 86

SL.1 – SL.16

86.1 Scope and mapping note

The ISO C++ Core Guidelines section **SL: The Standard Library** currently defines the top-level rules **SL.1–SL.4** and then continues with **subsections** (e.g., **SL.con.***, **SL.str.***, **SL.io.***).
To provide a single, sequential chapter **SL.1–SL.16** (as requested), this chapter uses the following mapping:

- **SL.1–SL.4** correspond directly to **SL.1–SL.4**.
- **SL.5–SL.8** correspond to **SL.con.1–SL.con.4** (containers).
- **SL.9–SL.13** correspond to **SL.str.1–SL.str.5** (strings/bytes).
- **SL.14–SL.16** correspond to **SL.io.1–SL.io.3** (iostreams).

Each rule title includes the original guideline identifier for precise traceability.

86.2 SL.1 (SL.1): Use libraries wherever possible

86.2.1 Rule

Use existing libraries rather than rewriting common facilities.

86.2.2 Reason

Reusing well-established library facilities saves time, reduces bugs, and benefits from continuous improvements by others.

86.2.3 Guidance

- Prefer standard algorithms (`std::sort`, `std::find`, `std::accumulate`, …) over handwritten loops when they clearly express the intent.
- Prefer standard containers (`vector`, `array`, `string`, …) over raw arrays and manual memory management.

86.2.4 Example

```
#include <algorithm>
#include <vector>

void sort_ids(std::vector<int>& ids)
{
    std::sort(ids.begin(), ids.end()); // clear intent, well-tested
}
```

86.2.5 Common mistakes

- Writing custom sort/search utilities without a demonstrated need.
- Re-implementing containers, strings, or smart pointers for general-purpose code.

86.3 SL.2 (SL.2): Prefer the standard library to other libraries

86.3.1 Rule

Prefer standard library components when they meet the requirements.

86.3.2 Reason

The standard library is widely known, stable, and broadly available across platforms and toolchains.

86.3.3 Guidance

- Prefer `std::vector` over a project-specific dynamic array unless you need a proven missing feature.
- Prefer `std::unique_ptr`/`std::shared_ptr` over ad-hoc ownership types.
- Prefer `std::filesystem` for portable path and file operations when available on your target toolchain.

86.3.4 Exception

Use a non-standard library when it provides essential guarantees the standard library does not, or when project constraints require it (e.g., certified safety libraries).

86.4 SL.3 (SL.3): Do not add non-standard entities to namespace `std`

86.4.1 Rule

Never add your own types, functions, or overloads to namespace `std`, except where the C++ Standard explicitly permits specialization for user-defined types.

86.4.2 Reason

Adding entities to `std` can change the meaning of otherwise conforming code and can conflict with future standard library additions.

86.4.3 Guidance

- Put your extensions in your own namespace.
- Only specialize standard templates where the Standard explicitly allows it (for example, some customizations for user-defined types).

86.4.4 Example

```
namespace std {           // BAD: violates the standard
    class My_vector { };
}

namespace mylib {         // GOOD
    class My_vector { };
}
```

86.5 SL.4 (SL.4): Use the standard library in a type-safe manner

86.5.1 Rule

Use the standard library without breaking its type assumptions.

86.5.2 Reason

Violating type requirements can lead to undefined behavior, memory corruption, and security vulnerabilities.

86.5.3 Guidance

- Avoid type-punning and raw byte operations on non-trivial objects.
- Use the correct types for ownership and views (`string`, `string_view`, `span`, `byte`).
- Prefer safe access patterns (range-for, iterators, bounds-checked access when needed).

86.5.4 Example: avoid raw-memory tricks on non-trivial objects

```
#include <cstring>
#include <string>

void bad(std::string& s)
{
    std::memset(&s, 0, sizeof(s)); // BAD: destroys object invariants
}
```

86.6 SL.5 (SL.con.1): Prefer `std::array` or `std::vector` over C arrays

86.6.1 Rule

Use `std::array` for fixed size and `std::vector` for variable size, instead of C arrays.

86.6.2 Reason

C arrays decay to pointers, do not carry size information, and are easy to misuse. `array` and `vector` are safer and generally have no disadvantages for typical code.

86.6.3 Guidance

- Use `std::array<T,N>` when the size is compile-time constant.
- Use `std::vector<T>` when the size varies or is decided at runtime.

86.6.4 Example

```
#include <array>
#include <vector>

constexpr std::size_t N = 16;

std::array<int, N> fixed{};
std::vector<int> dynamic(std::size_t n) { return std::vector<int>(n); }
```

86.7 SL.6 (SL.con.2): Prefer `std::vector` by default

86.7.1 Rule

Use `std::vector` as the default container unless you have a specific reason to choose a different one.

86.7.2 Reason

`vector` has contiguous storage, low overhead, excellent cache behavior, and fast general-purpose access.

86.7.3 Guidance

- If the size never changes after construction, consider `std::array`.
- Prefer `map`/`unordered_map` only when their lookup/insert characteristics are actually needed and measured.

86.7.4 Example: common initialization pitfall

```
#include <vector>

std::vector<int> v1(20);  // 20 elements, value-initialized (zeros)
std::vector<int> v2{20};  // 1 element with value 20
```

86.8 SL.7 (SL.con.3): Avoid bounds errors

86.8.1 Rule

Do not read or write outside the bounds of a container or array.

86.8.2 Reason

Out-of-bounds access is a major source of crashes, data corruption, and security vulnerabilities.

86.8.3 Guidance

- Prefer range-for and iterator-based loops over index arithmetic.
- When indexing is required, ensure indices are validated against `size()`.
- Use bounds-checked access (such as `.at()`) when appropriate, especially at trust boundaries.

86.8.4 Example: safe loop

```
#include <vector>

int sum(const std::vector<int>& v)
{
    int s = 0;
    for (int x : v) s += x;   // cannot exceed bounds
    return s;
}
```

86.9 SL.8 (SL.con.4): Do not use `memset`/`memcpy` on non-trivially-copyable objects

86.9.1 Rule

Do not use raw byte operations on objects that are not trivially copyable.

86.9.2 Reason

Such operations can break object invariants (e.g., overwrite a vptr), resulting in undefined behavior.

86.9.3 Guidance

- Use constructors, assignment, and standard algorithms (`std::fill`, `std::copy`) for objects.
- Raw byte functions are only appropriate for trivially copyable types and carefully controlled low-level code.

86.9.4 Example

```
#include <algorithm>
#include <array>

void good()
{
    std::array<int, 10> a{};
    a.fill(0); // correct and clear
}
```

86.10 SL.9 (SL.str.1): Use `std::string` to own character sequences

86.10.1 Rule

Use `std::string` as the owning type for text.

86.10.2 Reason

`string` correctly manages allocation, ownership, copying, and growth, and provides useful operations.

86.10.3 Example

```
#include <string>
#include <vector>
#include <iostream>

std::vector<std::string> read_until(const std::string& terminator)
{
    std::vector<std::string> res;
    for (std::string s; std::cin >> s && s != terminator; )
        res.push_back(s);
    return res;
}
```

86.11 SL.10 (SL.str.2): Use `std::string_view` or `span<char>` to refer to character sequences

86.11.1 Rule

Use non-owning views to refer to existing character data.

86.11.2 Reason

A view provides simple, efficient access independent of how the characters are stored. `std::string_view` is read-only.

86.11.3 Guidance

- Use `std::string_view` for read-only parameters.
- Use a mutable view (often `span<char>`) when mutation is required.
- Always ensure the referenced data outlives the view.

86.11.4 Example

```
#include <string>
#include <string_view>

bool has_prefix(std::string_view s, std::string_view prefix)
{
    return s.substr(0, prefix.size()) == prefix;
}
```

```
int main()
{
    std::string owned = "core-guidelines";
    (void)has_prefix(owned, "core"); // cheap view of owned data
}
```

86.12 SL.11 (SL.str.3): Use `zstring`/`czstring` for C-style zero-terminated strings

86.12.1 Rule

When you must refer to a C-style, zero-terminated string, use an explicit type/alias that documents that contract.

86.12.2 Reason

C-style strings have specific expectations (zero-termination) that are easy to violate when using raw `char*`. Making the intent explicit reduces errors.

86.12.3 Guidance

- Prefer `std::string` for ownership and `std::string_view` for non-owning read-only views.
- Use C-style strings only when required by C APIs or OS interfaces.

86.13 SL.12 (SL.str.4): Use `char*` to refer to a single character

86.13.1 Rule

Do not use `char*` to mean "string" unless it is explicitly a zero-terminated string; otherwise, `char*` should mean pointer to a single character.

86.13.2 Reason

The many informal meanings of `char*` in legacy code are a major source of bugs.

86.13.3 Example: dangerous misuse

```
#include <iostream>

void print(const char* p) { std::cout << p << '\n'; }

void bad()
{
    char arr[] = {'a', 'b', 'c'}; // NOT zero-terminated
    print(arr); // undefined behavior: reads past the array
}
```

86.14 SL.13 (SL.str.5): Use `std::byte` for non-character byte values

86.14.1 Rule

Use `std::byte` to represent raw bytes that are not characters.

86.14.2 Reason

Using `char` for non-text data is confusing and can inhibit optimizations and correct reasoning about intent. `std::byte` is the standard byte type (C++17).

86.14.3 Example

```
#include <cstddef>   // std::byte
#include <vector>

void xor_mask(std::vector<std::byte>& data, std::byte mask)
{
    for (auto& b : data) {
        b = b ^ mask;
    }
}
```

86.15 SL.14 (SL.io.1): Use character-level input only when you have to

86.15.1 Rule

Do not build token parsing manually from `cin.get()` unless you truly need character-by-character processing.

86.15.2 Reason

Manual character-level token composition is often more error-prone and can be less efficient than using formatted input into higher-level types.

86.15.3 Example

```
#include <iostream>
#include <string>

std::string read_word()
{
    std::string s;
    std::cin >> s; // simple token extraction
    return s;
}
```

86.16 SL.15 (SL.io.2): When reading, always consider ill-formed input

86.16.1 Rule

Treat input as potentially invalid and validate as early as possible.

86.16.2 Reason

If input is not validated at boundaries, every subsequent function must defensively handle corrupted data, which is not practical and leads to vulnerabilities.

86.16.3 Guidance

- Check stream state after reads (`if (!cin)`).
- Validate ranges and formats immediately after parsing.

- Prefer fail-fast or early error handling at input boundaries.

86.16.4 Example

```
#include <iostream>

bool read_int_in_range(int& out, int lo, int hi)
{
    int x{};
    if (!(std::cin >> x)) return false;
    if (x < lo || x > hi) return false;
    out = x;
    return true;
}
```

86.17 SL.16 (SL.io.3): Prefer iostreams for I/O

86.17.1 Rule

Prefer C++ iostreams for I/O in C++ code.

86.17.2 Reason

`iostream` is type-safe, flexible, and extensible, supports locales and buffering strategies, and integrates well with C++ types.

86.17.3 Guidance

- Use formatted extraction/insertion (>>, <<) when appropriate.

- Use unformatted functions (`read`, `write`, `getline`) for binary or line-based protocols.
- If you do not use `printf`-family I/O, consider disabling synchronization for performance-sensitive streaming code (`ios_base::sync_with_stdio(false)`).

86.17.4 Example

```
#include <iostream>
#include <string>

int main()
{
    std::ios_base::sync_with_stdio(false);

    std::string name;
    if (std::cin >> name) {
        std::cout << "Hello " << name << '\n';
    }
}
```

86.18 Consolidated checklist

- Prefer standard facilities and standard vocabulary types (`vector`, `string`, `string_view`, `byte`).
- Do not modify namespace `std`.
- Avoid out-of-bounds access and raw byte operations on non-trivial objects.
- Use views for non-owning references and ensure lifetime correctness.

- Treat input as hostile: validate early.

Chapter 87

Proper Use of the STL

87.1 Purpose

The Standard Template Library (STL) is the core of modern C++ programming style: **containers** hold data, **algorithms** process it, and **iterators/ranges** connect them. Proper STL use leads to code that is:

- safer (fewer bounds and lifetime mistakes),
- clearer (intent is explicit),
- faster (well-optimized library implementations),
- more portable (standardized behavior and complexity guarantees).

This chapter provides practical rules for correct and effective STL usage, aligned with the ISO C++ Core Guidelines **SL** section.

87.2 The STL mental model

87.2.1 Three parts that must match

- **Containers:** store elements (`vector`, `array`, `string`, `map`, `unordered_map`, …).
- **Algorithms:** operate on iterator ranges (`sort`, `find`, `transform`, `accumulate`, …).
- **Iterators/Ranges:** define where the algorithm operates.

87.2.2 Rule of thumb

Prefer writing code as:

`algorithm(range, ...)` instead of `manual loops and indexing`

unless a manual loop is clearer for that specific case.

87.3 Core Guidelines alignment (SL)

87.3.1 High-level rules you should enforce

- Use libraries wherever possible (do not re-invent standard tools).
- Prefer the standard library when it meets your needs.
- Never add your own entities to namespace `std`.
- Use the standard library in a type-safe manner (avoid raw-memory tricks).

87.4 Prefer algorithms over raw loops

87.4.1 Why

- Expresses intent: `sort` means sorting, `find` means search.
- Reduces bugs: avoids off-by-one and invalid indexing.
- Enables optimization: implementations can be specialized and tuned.

87.4.2 Example: `find` + `erase` with correct semantics

```
#include <algorithm>
#include <vector>

void remove_value(std::vector<int>& v, int x)
{
    v.erase(std::remove(v.begin(), v.end(), x), v.end()); // erase-remove idiom
}
```

87.4.3 Example: transform instead of manual mapping

```
#include <algorithm>
#include <vector>

std::vector<int> squares(const std::vector<int>& in)
{
    std::vector<int> out;
    out.reserve(in.size());
    std::transform(in.begin(), in.end(), std::back_inserter(out),
```

```
                   [](int x) { return x * x; });
    return out;
}
```

87.5 Use the right container (the default is `std::vector`)

87.5.1 `std::vector` as the default

Use `std::vector` unless you have a concrete reason to choose another container:

- contiguous storage (cache-friendly),
- fast random access,
- excellent general-purpose performance.

87.5.2 When `std::array` is better

Use `std::array<T,N>` for fixed-size sequences known at compile time.

```
#include <array>

std::array<int, 4> digits{1,2,3,4};
```

87.5.3 When `std::deque` is useful

Use `std::deque` when you need efficient push/pop at both ends and stable references across growth patterns (but not the same locality as `vector`).

87.5.4 When node-based containers are appropriate

Use `map`/`set` for ordered associative structures, and `unordered_map`/`unordered_set` for hash-based lookup, but understand the trade-offs:

- node-based containers allocate per node and typically have weaker cache locality,
- hash tables can rehash (cost spikes) and depend on hash quality,
- ordered maps have `O(log n)` access with ordering guarantees.

87.6 Know iterator invalidation rules

Iterator/reference invalidation is one of the most common STL correctness bugs.

87.6.1 Key facts (high-impact)

- `vector` reallocation invalidates **all** iterators/references/pointers into the vector.
- `vector::erase` invalidates iterators from the erased position to `end()`.
- `unordered_map` may invalidate iterators during rehashing.

87.6.2 Example: avoid keeping references across vector growth

```
#include <vector>

void bad()
{
    std::vector<int> v;
    v.push_back(1);
```

```
    int& r = v[0];       // reference into v
    v.push_back(2);      // may reallocate, invalidating r
    // using r here would be undefined if reallocation happened
}
```

87.6.3 Practical rule

If you must keep stable references:

- reserve enough capacity first, or
- store indices instead of pointers/references, or
- choose a container whose invalidation rules match your needs.

87.7 Avoid bounds errors by design

87.7.1 Prefer range-based loops

```
#include <vector>

int sum(const std::vector<int>& v)
{
    int s = 0;
    for (int x : v) s += x;
    return s;
}
```

87.7.2 Use `.at()` at trust boundaries

When reading untrusted sizes/indices (file, network, user input), consider bounds-checked access:

```
#include <vector>

int safe_get(const std::vector<int>& v, std::size_t i)
{
    return v.at(i); // throws std::out_of_range if i is invalid
}
```

87.8 Use views to avoid unnecessary allocations and copies

87.8.1 `std::string_view` for read-only string parameters

Prefer `std::string_view` for functions that do not need ownership:

```
#include <string_view>

bool is_keyword(std::string_view s)
{
    return s == "if" || s == "while" || s == "return";
}
```

Rule: the referenced data must outlive the view.

87.8.2 `std::span` for non-owning array views

Use `std::span` to pass contiguous data with size:

```
#include <span>
#include <cstddef>

int sum(std::span<const int> a)
{
    int s = 0;
    for (int x : a) s += x;
    return s;
}
```

87.9 Prefer value semantics and RAII

87.9.1 Use standard owning types

Use:

- `std::unique_ptr` for exclusive ownership,
- `std::shared_ptr` only when shared ownership is truly required,
- standard containers to own sequences.

87.9.2 Avoid manual `new`/`delete`

Prefer factory helpers and containers:

```
#include <memory>

struct X { int a; };

auto make_x()
```

```
{
    return std::make_unique<X>(X{42});
}
```

87.10 Use the correct algorithm for the job

87.10.1 Do not misuse `std::sort` vs `std::stable_sort`

- Use `sort` when stability is not required.
- Use `stable_sort` when equal elements must keep original relative order.

87.10.2 Prefer partitioning and selection algorithms when you do not need full sorting

If you only need:

- "all elements that match a predicate first": use `partition`.
- "the k-th element": use `nth_element`.

This can reduce work compared to full sorting.

87.11 Comparators and strict weak ordering

Many ordered algorithms and containers require a comparator that forms a **strict weak ordering**. Violating this contract leads to undefined behavior or incorrect results.

87.11.1 Safe comparator example

```
#include <string>

struct ByLengthThenLex {
    bool operator()(const std::string& a, const std::string& b) const
    {
        if (a.size() != b.size()) return a.size() < b.size();
        return a < b;
    }
};
```

87.12 Avoid raw memory functions on non-trivial types

87.12.1 Rule

Do not use `memset`/`memcpy`/`memcmp` to manipulate objects that are not trivially copyable.

87.12.2 Correct alternatives

Use:

- constructors/assignment,
- `std::fill`, `std::copy`,
- container operations.

```
#include <algorithm>
#include <array>
```

```
void good()
{
    std::array<int, 8> a{};
    std::fill(a.begin(), a.end(), 0);
}
```

87.13 Understand complexity and avoid accidental quadratic behavior

87.13.1 Common trap: repeated growth without reserving

```
#include <string>
#include <vector>

std::string join_words(const std::vector<std::string>& words)
{
    std::string out;
    // Better: compute total size and reserve
    for (const auto& w : words) {
        out += w; // may cause repeated reallocations
    }
    return out;
}
```

87.13.2 Better: reserve capacity

```
#include <string>
#include <vector>
```

```
std::string join_words_better(const std::vector<std::string>& words)
{
    std::size_t total = 0;
    for (const auto& w : words) total += w.size();

    std::string out;
    out.reserve(total);

    for (const auto& w : words) out += w;
    return out;
}
```

87.14 Use ranges when they improve clarity (C++20 and later)

Ranges reduce iterator boilerplate and help express pipelines.

87.14.1 Example: filter then transform

```
#include <vector>
#include <ranges>

std::vector<int> evens_squared(const std::vector<int>& v)
{
    auto view = v
        | std::views::filter([](int x){ return (x % 2) == 0; })
        | std::views::transform([](int x){ return x * x; });

    return {view.begin(), view.end()};
}
```

87.14.2 Important rule

Views are typically non-owning; ensure lifetime correctness (similar to `string_view`).

87.15 Use polymorphic allocators (`std::pmr`) when allocation strategy matters

For performance-sensitive subsystems with many allocations, `std::pmr` allows you to control allocation centrally.

```
#include <memory_resource>
#include <vector>
#include <string>

std::pmr::vector<std::pmr::string> build(std::pmr::memory_resource* mr)
{
    std::pmr::vector<std::pmr::string> v{mr};
    v.emplace_back("alpha", mr);
    v.emplace_back("beta", mr);
    return v;
}
```

87.16 Common STL anti-patterns (and fixes)

87.16.1 Anti-pattern 1: using `std::vector<bool>` as a normal container

`vector<bool>` is a space-optimized specialization that does not store `bool` as normal elements. If you need a true contiguous array of bytes, prefer `vector<unsigned char>` or `vector<std::byte>`.

87.16.2 Anti-pattern 2: storing raw pointers as ownership

Prefer `unique_ptr` or values. Raw pointers should usually mean **non-owning observation**.

87.16.3 Anti-pattern 3: ignoring invalidation and lifetime

Never keep iterators/references/views beyond the operations that may invalidate them.

87.16.4 Anti-pattern 4: mixing unrelated responsibilities

Do not embed parsing, allocation management, and business logic into one giant loop. STL works best with decomposed steps.

87.17 Practical checklist: proper STL usage

- Choose the simplest container that matches your access pattern (default: `vector`).
- Prefer algorithms (and ranges where appropriate) over handwritten loops.
- Use `string_view` and `span` for non-owning parameters; validate lifetimes.
- Know invalidation rules for the containers you use.
- Avoid raw memory operations on non-trivial types.
- Use `reserve()` when growth is predictable.
- Validate complexity; do not accidentally write quadratic loops.
- Keep your code type-safe; do not abuse `void*` or byte tricks.
- Never add anything to namespace `std`.

87.18 Summary

Proper STL use means writing **intent-first**, **type-safe**, **range-based** code that respects container and algorithm contracts:

- containers manage ownership,
- algorithms express operations over ranges,
- iterators/ranges define valid traversal,
- views (`string_view`, `span`, ranges views) require careful lifetime reasoning.

This style matches the Core Guidelines goal: simpler code that is safer and usually faster.

Chapter 88

Containers

88.1 Purpose

Standard library containers are the primary way to store and organize data in modern C++. Proper container choice and correct use improves:

- safety (fewer lifetime and bounds errors),
- clarity (intent is visible in types and operations),
- performance (cache-friendly layout, fewer allocations, predictable complexity),
- portability (standardized semantics and guarantees).

This chapter summarizes the Core Guidelines container rules (SL.con) and provides practical, modern C++ usage patterns.

88.2 Container taxonomy

The standard containers are commonly grouped into:

88.2.1 Sequence containers

- `std::array<T,N>` fixed size, contiguous
- `std::vector<T>` dynamic size, contiguous (except `vector<bool>`)
- `std::deque<T>` dynamic size, segmented storage
- `std::list<T>` doubly-linked list (node-based)
- `std::forward_list<T>` singly-linked list (node-based)

88.2.2 Associative containers (ordered)

- `std::map<Key,T>`, `std::multimap<Key,T>`
- `std::set<Key>`, `std::multiset<Key>`

These maintain elements ordered by a comparator (typically `std::less<>`).

88.2.3 Unordered associative containers (hash-based)

- `std::unordered_map<Key,T>`, `std::unordered_multimap<Key,T>`
- `std::unordered_set<Key>`, `std::unordered_multiset<Key>`

These provide average constant-time lookup under typical assumptions.

88.2.4 Container adaptors

- `std::stack<T>`, `std::queue<T>`, `std::priority_queue<T>`

Adaptors expose restricted interfaces built on an underlying container.

88.3 The Core Guidelines rules for containers (SL.con)

88.3.1 SL.con.1: Prefer using STL `array` or `vector` instead of a C array

Rule. Prefer `std::array` or `std::vector` over C arrays.

Rationale. C arrays decay to pointers, lose size information, and are easy to misuse. `array`/`vector` carry size and integrate naturally with algorithms.

Example.

```
#include <array>
#include <vector>

std::array<int, 4> fixed_ids{1,2,3,4};

std::vector<int> dynamic_ids(std::size_t n)
{
    return std::vector<int>(n);
}
```

88.3.2 SL.con.2: Prefer using `vector` by default unless you have a reason to use a different container

Rule. Use `std::vector` as the default container.

Rationale. `vector` has low overhead, contiguous storage (good locality), and a simple cost model. For most workloads it is the best starting point.

Example: the most common initialization pitfall

```
#include <vector>

std::vector<int> a(10);  // 10 elements, value-initialized
```

```
std::vector<int> b{10};  // 1 element: value 10
```

88.3.3 SL.con.3: Avoid bounds errors

Rule. Avoid out-of-bounds access.

Rationale. Bounds errors are a major source of crashes and vulnerabilities.

Preferred patterns.

- Prefer range-for loops and algorithms.
- Prefer iterator ranges and `std::span` for non-owning views over raw pointer pairs.
- Use bounds-checked access (such as `.at()`) at trust boundaries when appropriate.

Example: range-for avoids index mistakes

```
#include <vector>

int sum(const std::vector<int>& v)
{
    int s = 0;
    for (int x : v) s += x;
    return s;
}
```

88.3.4 SL.con.4: Do not use `memset`/`memcpy` for arguments that are not trivially-copyable

Rule. Do not apply raw byte operations to non-trivially-copyable types.

Rationale. Such operations can violate invariants (e.g., corrupt a vptr, destroy a `std::string`).

Use instead. `std::fill`, `std::copy`, constructors, assignment, and container operations.

Example.

```cpp
#include <algorithm>
#include <array>

void zero_out(std::array<int, 8>& a)
{
    std::fill(a.begin(), a.end(), 0);
}
```

88.4 How to choose the right container

88.4.1 Start with the access pattern

Use this decision order:

1. Do you need contiguous storage (cache friendliness, SIMD, C API interop)?
2. Do you need stable references/iterators across insert/erase?
3. Do you need fast random access by index?
4. Do you need ordered traversal by key?
5. Do you need hash-based lookup?

88.4.2 Default recommendations

- **Fixed size:** `std::array`
- **Resizable sequence:** `std::vector`
- **Many insert/erase in middle with stable iterators:** consider `std::list` only if profiling justifies it

- **Ordered map/set:** `std::map`/`std::set`
- **Fast average key lookup:** `std::unordered_map`/`std::unordered_set`
- **Stack/queue interface only:** `std::stack`/`std::queue` adaptors

88.5 Contiguous storage and locality

88.5.1 `std::vector` and contiguous elements

With the special case of `std::vector<bool>`, `std::vector<T>` stores elements contiguously, enabling efficient linear traversal and interoperability with APIs expecting `T*`. This property is one reason `vector` is the recommended default.

88.5.2 Avoid `std::vector<bool>` when you need normal element semantics

`vector<bool>` is a space-optimized specialization where elements may be packed as bits, so it does not behave like a normal `vector<T>`.

Preferred alternatives.

- `std::vector<unsigned char>` or `std::vector<std::byte>` for packed storage with normal element references
- `std::vector<char>` for raw character buffers (when appropriate)

88.6 Iterator and reference invalidation

Iterator/reference invalidation is one of the most common STL correctness bugs. Always understand the invalidation rules of the container you choose.

88.6.1 A safe rule for `std::vector`

Any operation that causes reallocation invalidates all pointers, references, and iterators into the vector.

88.6.1.1 Example: avoid keeping references across growth

```
#include <vector>

void bad()
{
    std::vector<int> v;
    v.push_back(1);
    int& r = v[0];

    v.push_back(2); // may reallocate, invalidating r

    // Using r here is unsafe if reallocation occurred.
}
```

88.6.1.2 Safer approaches

- call `reserve()` when the final size is known or bounded,
- store indices instead of references,
- redesign to avoid keeping iterators across mutating operations.

88.6.2 Erasing while iterating

Prefer the erase-remove idiom for sequences, and use the returned iterator from `erase` for associative containers.

88.6.2.1 Erase-remove idiom for `vector`

```
#include <algorithm>
#include <vector>

void remove_value(std::vector<int>& v, int x)
{
    v.erase(std::remove(v.begin(), v.end(), x), v.end());
}
```

88.6.2.2 Erasing from `unordered_map` while iterating

```
#include <unordered_map>

void erase_if_small(std::unordered_map<int, int>& m, int limit)
{
    for (auto it = m.begin(); it != m.end(); ) {
        if (it->second < limit) it = m.erase(it);
        else ++it;
    }
}
```

88.7 Performance guidance for containers

88.7.1 Minimize allocations

`vector` growth and node-based containers can allocate frequently. In performance-sensitive code:

- use `reserve()` for `vector` when size is predictable,

- prefer contiguous layouts for hot data,
- consider `std::pmr` containers for allocation strategy control when justified.

88.7.2 Prefer locality over pointer-chasing on hot paths

Node-based containers (`list`, `map`) are often slower for traversal-heavy workloads due to poor locality. Do not assume they are faster; measure.

88.8 Ownership and lifetime with containers

88.8.1 Store values by default

Prefer storing values rather than owning raw pointers. This improves safety, clarity, and exception correctness.

88.8.2 If you need dynamic lifetime, prefer owning smart pointers

Use:

- `std::unique_ptr<T>` in containers for exclusive ownership,
- `std::shared_ptr<T>` only when shared ownership is required.

88.8.2.1 Example: `vector<unique_ptr<T>>`

```
#include <memory>
#include <vector>

struct Job { int id; };
```

```
std::vector<std::unique_ptr<Job>> make_jobs(int n)
{
    std::vector<std::unique_ptr<Job>> v;
    v.reserve(static_cast<std::size_t>(n));
    for (int i = 0; i < n; ++i) {
        v.push_back(std::make_unique<Job>(Job{i}));
    }
    return v;
}
```

88.9 Non-owning views: `std::span`

88.9.1 Why `span` helps

`std::span<T>` represents a non-owning view of contiguous data with a size. It is a safer alternative to `T*` plus length.

88.9.2 Example

```
#include <span>
#include <vector>

int sum(std::span<const int> a)
{
    int s = 0;
    for (int x : a) s += x;
    return s;
}
```

```
int main()
{
    std::vector<int> v{1,2,3};
    (void)sum(v); // implicit span view of v's elements
}
```

88.10 Common container mistakes and fixes

88.10.1 Mistake: choosing `list` for "fast insert/erase" without measuring

Insert/erase can be fast, but traversal and memory overhead are often worse. Choose based on workload and profiling.

88.10.2 Mistake: using `unordered_map` without reserving in growth-heavy code

Rehashing can be expensive and can invalidate iterators. If you can estimate the number of elements, pre-reserve.

88.10.3 Mistake: mixing representation with interface

Hide container choice behind a stable interface when it may change. This keeps performance tuning possible without breaking users.

88.11 Summary

- Prefer `std::array`/`std::vector` over C arrays (SL.con.1).
- Use `std::vector` as the default container (SL.con.2).

- Avoid bounds errors by design: algorithms, ranges, `span`, and safe access (SL.con.3).
- Never use raw byte operations on non-trivial objects (SL.con.4).
- Understand iterator invalidation and erase patterns.
- Prefer locality and fewer allocations on hot paths; validate with measurements.

Chapter 89

Algorithms

89.1 Purpose

The C++ standard library algorithms (commonly called the STL algorithms) provide a large set of well-specified, well-tested operations that work on *ranges of elements*. Proper use of these algorithms makes code:

- clearer (the algorithm name expresses intent),
- safer (fewer indexing and loop-bound mistakes),
- often faster (library implementers can optimize aggressively),
- more maintainable (less handwritten control flow).

This chapter follows the ISO C++ Core Guidelines philosophy for the standard library (use the library, prefer standard facilities, keep code type-safe) and uses the standard algorithm contracts as the reference model for correctness and complexity.

89.2 The algorithm model: “operate on a range”

89.2.1 Classic iterator-pair form

Most algorithms take a half-open range `[first,last)`.

```
#include <algorithm>
#include <vector>

void normalize(std::vector<int>& v)
{
    std::sort(v.begin(), v.end()); // sorts the range [begin,end)
}
```

89.2.2 C++20 ranges algorithms (recommended when available)

C++20 introduces constrained algorithms in `std::ranges`. They:

- accept a single range object (container or view) or iterator/sentinel,
- support **projections** (extract a key without writing a comparator),
- often return richer results (useful iterators packaged in result types).

```
#include <algorithm>
#include <ranges>
#include <vector>

struct Item { int id; int score; };

void sort_by_score(std::vector<Item>& v)
{
```

```
    std::ranges::sort(v, {}, &Item::score); // projection: compare by score
}
```

89.3 Core principles for proper algorithm use

89.3.1 Principle 1: Prefer algorithms to handwritten loops

Use named algorithms when they express the intent:

- `std::find`, `std::count`, `std::any_of` for search and predicates,
- `std::transform` for mapping,
- `std::sort`/`std::stable_sort` for sorting,
- `std::accumulate` / `std::reduce` for reductions.

This aligns with the Core Guidelines preference for using standard facilities and keeping code simple and type-safe.

89.3.2 Principle 2: Make preconditions explicit

Algorithms have contracts. Common preconditions include:

- iterators must be valid and refer to the same sequence,
- ranges must not overlap when the algorithm forbids overlap,
- the comparator must satisfy strict-weak-ordering requirements (for sorting/ordered operations),
- the predicate must not mutate the range unless explicitly allowed.

89.3.3 Principle 3: Understand iterator invalidation

Algorithms do not usually invalidate iterators by themselves, but *container operations* often do (e.g., erase, insertion, reallocation). Avoid keeping iterators, pointers, or references across container modifications.

89.4 Algorithm categories (what to reach for)

89.4.1 Non-modifying sequence algorithms

Use these to inspect data without changing it:

- `std::find`, `std::find_if`
- `std::count`, `std::count_if`
- `std::any_of`, `std::all_of`, `std::none_of`

```
#include <algorithm>
#include <vector>

bool contains_even(const std::vector<int>& v)
{
    return std::any_of(v.begin(), v.end(), [](int x){ return (x % 2) == 0; });
}
```

89.4.2 Modifying sequence algorithms

Use these to transform elements:

- `std::transform` (map)

- `std::replace`, `std::replace_if`
- `std::remove`, `std::remove_if` (logical removal, see erase-remove idiom)
- `std::reverse`, `std::rotate`

```
#include <algorithm>
#include <vector>

void square_in_place(std::vector<int>& v)
{
    std::transform(v.begin(), v.end(), v.begin(), [](int x){ return x * x; });
}
```

89.4.3 Sorting and ordering

- `std::sort`: fastest general-purpose sort when stability is not required
- `std::stable_sort`: preserves relative order of equivalent elements
- `std::partial_sort`, `std::nth_element`: avoid full sort when you only need top-k or the k-th element

```
#include <algorithm>
#include <vector>

int median(std::vector<int> v)
{
    auto mid = v.begin() + (v.size() / 2);
    std::nth_element(v.begin(), mid, v.end());
    return *mid;
}
```

89.4.4 Binary search on sorted data

If a range is sorted by a comparator, use:

- `std::binary_search`
- `std::lower_bound`, `std::upper_bound`
- `std::equal_range`

89.5 Correctness contracts you must respect

89.5.1 Comparators: strict weak ordering

Many algorithms require that the comparator define a strict weak ordering (informally: consistent "less-than" with transitivity and no contradictions). Violating this can lead to undefined behavior or incorrect results.

89.5.1.1 Correct comparator example

```
#include <string>

struct ByLenThenLex {
    bool operator()(const std::string& a, const std::string& b) const
    {
        if (a.size() != b.size()) return a.size() < b.size();
        return a < b;
    }
};
```

89.5.2 Projections (C++20 ranges): prefer them to custom comparators

When sorting by a field, projections often reduce bugs:

```
#include <ranges>
#include <vector>

struct P { int id; int score; };

void sort_by_id(std::vector<P>& v)
{
    std::ranges::sort(v, {}, &P::id); // projection instead of comparator
    ↪ boilerplate
}
```

89.5.3 Overlapping ranges: do not assume it is safe

Some algorithms permit overlap (e.g., `std::copy_backward`) while others do not. When overlap is possible, choose the algorithm that explicitly supports it. When in doubt, consult the algorithm contract.

89.6 Erase correctly: the erase-remove idiom

89.6.1 Why `remove` does not erase

`std::remove` and `std::remove_if` do not change the container size; they move "kept" elements to the front and return a new logical end.

89.6.2 The correct pattern for sequence containers

```
#include <algorithm>
#include <vector>

void remove_value(std::vector<int>& v, int x)
{
    v.erase(std::remove(v.begin(), v.end(), x), v.end());
}
```

This is a foundational STL pattern and should be preferred over manual shifting loops.

89.7 Numeric algorithms and reductions

89.7.1 `std::accumulate` vs `std::reduce`

- `std::accumulate` processes in order (deterministic for non-associative operations).
- `std::reduce` (since C++17) may reorder operations (especially with parallel execution), so the binary operation should be associative and commutative for reliable results (floating-point sums can differ).

```
#include <numeric>
#include <vector>

double sum_ordered(const std::vector<double>& v)
{
    return std::accumulate(v.begin(), v.end(), 0.0); // fixed order
}
```

89.8 Parallel algorithms (C++17 execution policies): power with sharp edges

89.8.1 What execution policies do

Many algorithms have overloads that accept an execution policy (since C++17), enabling:

- sequenced execution (`std::execution::seq`),
- parallel execution (`std::execution::par`),
- parallel and vectorized/unsequenced execution (`std::execution::par_unseq`),
- vectorized/unsequenced execution (`std::execution::unseq`).

89.8.2 Critical safety rule: avoid data races and disallowed synchronization

With parallel/unsequenced policies:

- the callable must be safe under concurrent/unsequenced invocation,
- you must not introduce data races (no shared non-atomic writes without synchronization),
- for unsequenced forms, operations that rely on sequencing or synchronization are restricted; the standard library describes additional constraints for such policies.

89.8.3 Exception behavior with parallel algorithms

For standard parallel algorithm overloads, if a user-provided callable exits via an uncaught exception, the standard specifies that `std::terminate` is called (implementations may offer additional policies with different handling).

89.8.4 Example: safe parallel transformation into an independent output

```
#include <algorithm>
#include <execution>
#include <vector>

void square_parallel(const std::vector<int>& in, std::vector<int>& out)
{
    out.resize(in.size());
    std::transform(std::execution::par,
                   in.begin(), in.end(),
                   out.begin(),
                   [](int x){ return x * x; });
}
```

89.9 Ranges pipelines (C++20): expressive composition

89.9.1 Views are lazy; algorithms are eager

Ranges introduces:

- **views** (lazy adaptors, composed with |),
- **range algorithms** (eager computations over ranges).

89.9.2 Example: filter + transform pipeline, then materialize

```
#include <ranges>
#include <vector>
```

```
std::vector<int> evens_squared(const std::vector<int>& v)
{
    auto view = v
        | std::views::filter([](int x){ return (x % 2) == 0; })
        | std::views::transform([](int x){ return x * x; });

    return {view.begin(), view.end()};
}
```

89.9.3 Lifetime rule

Many views are non-owning; ensure the underlying range outlives the view usage.

89.10 Practical selection guide

89.10.1 Common tasks and the right algorithms

- **Search:** `find`, `find_if`, `any_of`
- **Transform:** `transform`
- **Filter (classic):** `remove_if` + `erase` (erase-remove idiom)
- **Sort:** `sort`, `stable_sort`
- **Top-k:** `partial_sort` or `nth_element`
- **Binary search:** `lower_bound`/`upper_bound` on sorted ranges
- **Fold/reduce:** `accumulate` (ordered), `reduce` (potentially reordered)

89.11 Common mistakes and how to avoid them

89.11.1 Mistake 1: writing loops that re-implement algorithms

Fix: use the named algorithm; it documents intent and reduces bugs.

89.11.2 Mistake 2: using `remove` without `erase`

Fix: always complete the erase-remove idiom for containers that support erase.

89.11.3 Mistake 3: invalidating iterators and then using them

Fix: do not keep iterators across container modifications; capture returned iterators where required.

89.11.4 Mistake 4: using parallel policies with shared mutable state

Fix: ensure callables are free of data races; write into independent output ranges; avoid locks and shared mutation with `par_unseq`/`unseq`.

89.11.5 Mistake 5: assuming floating-point reductions are identical

Fix: use `accumulate` for stable ordered accumulation; treat `reduce` (especially parallel) as potentially different due to reordering.

89.12 Checklist: proper algorithm usage

- Prefer named algorithms (or ranges algorithms) over manual loops.

- Use `std::ranges` algorithms when available for clearer range-based calls and projections.
- Respect contracts: valid iterators, correct comparator, no forbidden overlap, no data races.
- Use the erase-remove idiom for sequence containers.
- Use execution policies only when callables are race-free and meet the policy constraints.
- Measure performance; do not guess.

Chapter 90

Ranges

90.1 Purpose

The C++20 Ranges library generalizes the classic "iterators + algorithms" STL model into a safer and more composable design:

- **Ranges** describe sequences (with explicit begin/end semantics).
- **Views** are lightweight, usually non-owning, *lazy* ranges that adapt or transform other ranges.
- **Constrained algorithms** in `std::ranges` accept a range directly, support **projections**, and return richer results.

Ranges reduce common STL mistakes (iterator mismatches, accidental dangling iterators, noisy boilerplate) while preserving the STL philosophy of expressing intent using standard facilities.

90.2 Core ideas and terminology

90.2.1 Range

A **range** is something you can iterate: it provides a `begin` and an `end` (possibly different types: iterator + sentinel).

90.2.2 View

A **view** is a range that is cheap to copy and typically does not own its elements. Views usually adapt another range (filter, transform, take, drop, split, join, …) and are often evaluated lazily when iterated.

90.2.3 Iterator and sentinel

Ranges allow **iterator/sentinel** pairs instead of requiring two iterators of the same type. This makes it possible to represent end conditions efficiently (e.g., null-terminated sequences, streaming sources).

90.2.4 Constrained algorithms

C++20 provides most algorithms as constrained overloads in `std::ranges`:

- pass a range directly: `std::ranges::sort(v)` instead of `std::sort(v.begin(), v.end())`
- projections: specify "how to view each element" for comparison/search without writing a comparator wrapper
- richer return types: many algorithms return structs (e.g., subranges) with useful iterators

90.3 Why ranges improve STL usage

90.3.1 Less error-prone

Ranges algorithms reduce mistakes such as:

- passing `begin()` of one container and `end()` of another,
- returning iterators that dangle when a temporary range is passed,
- writing custom loops where an algorithm expresses intent more clearly.

90.3.2 Better composability

Views compose using `|` into readable pipelines. Each stage remains a small, testable operation.

90.3.3 Performance-friendly by default

Views are typically lazy: they do not allocate new containers unless you materialize the result. When designed well, the pipeline can avoid intermediate allocations and copies.

90.4 Basic usage patterns

90.4.1 Ranges algorithms (direct range argument)

```
#include <algorithm>
#include <ranges>
#include <vector>

void sort_values(std::vector<int>& v)
{
```

```
    std::ranges::sort(v); // range-based
}
```

This is the most direct “upgrade” from iterator-pair algorithms.

90.4.2 Projections (C++20) to reduce comparator boilerplate

```
#include <algorithm>
#include <ranges>
#include <vector>

struct Item { int id; int score; };

void sort_by_score(std::vector<Item>& v)
{
    std::ranges::sort(v, {}, &Item::score); // projection: compare scores
}
```

Projections reduce bugs and clarify intent.

90.4.3 Finding elements with projections

```
#include <algorithm>
#include <ranges>
#include <vector>

struct User { int id; int flags; };

auto find_user(std::vector<User>& users, int id)
{
```

```
    return std::ranges::find(users, id, &User::id); // projection: compare by id
}
```

90.5 Views and pipelines

90.5.1 Filter and transform

```
#include <ranges>
#include <vector>

std::vector<int> evens_squared(const std::vector<int>& v)
{
    auto view = v
        | std::views::filter([](int x){ return (x % 2) == 0; })
        | std::views::transform([](int x){ return x * x; });

    // Materialize into a vector
    return {view.begin(), view.end()};
}
```

90.5.2 Key rule: views are usually non-owning

Most views refer to the underlying range. If the underlying container (or other range) is destroyed, the view becomes invalid. Treat views like iterators or `string_view`: their lifetime is tied to the referenced data.

90.5.3 When to materialize

Materialize (copy results into an owning container) when you need:

- a stable, owning result,
- multiple passes over the result with independent lifetime,
- random access and indexing (not always provided by a view),
- interoperability with APIs that require owning storage.

90.6 Returning iterators safely: borrowed ranges and `dangling`

90.6.1 The problem

Many algorithms return iterators/subranges pointing into the input range. If you pass a temporary (rvalue) range, those iterators would dangle.

90.6.2 The solution in C++20

Ranges introduces:

- `std::ranges::borrowed_range`: ranges for which it is safe to return iterators even when passed as rvalues
- `std::ranges::dangling`: a placeholder returned when an iterator/subrange would otherwise dangle

90.6.3 What this means in practice

If you call a constrained algorithm with an rvalue range that is not a borrowed range, the return type may be `dangling` rather than an iterator/subrange. This is a safety feature: it prevents you from accidentally using a dangling result.

90.7 Subranges: a first-class “slice” type

90.7.1 `std::ranges::subrange`

A `subrange` bundles an iterator and sentinel (and optionally size information) into a view-like object. It is commonly used as a return type from constrained algorithms that want to return “the remaining range” or “a found part.”

90.7.2 Example: remove returns a subrange

Many constrained algorithms return subranges that carry useful endpoints. For example, the ranges version of remove returns a subrange representing the new logical end.

```
#include <algorithm>
#include <ranges>
#include <vector>

void remove_value(std::vector<int>& v, int x)
{
    auto r = std::ranges::remove(v, x); // returns subrange [new_end, old_end)
    v.erase(r.begin(), v.end());        // erase the moved-to-the-end elements
}
```

90.8 Contracts and correctness rules

90.8.1 Predicates and projections should be pure

In pipelines and constrained algorithms, the library may evaluate predicates in patterns you should not rely on (e.g., order and number of calls may vary). For correctness and predictability:

- keep predicates and projections free of side effects,
- do not mutate shared state from a predicate unless it is carefully synchronized and allowed by your design.

90.8.2 Comparators must satisfy strict weak ordering

Sorting and ordered operations require a comparator that forms a strict weak ordering. If violated, behavior is not reliable.

90.8.3 Iterator category matters

Some algorithms require stronger iterator categories:

- `sort` requires random-access iterators (so it works for `vector` and `deque`, not for `list`).
- many views preserve only certain capabilities (e.g., not every view is random-access).

The range concepts enforce these constraints at compile time.

90.9 Interoperability with existing STL types

90.9.1 Containers as ranges

All standard containers can be used as ranges:

- pass the container directly to `std::ranges` algorithms,
- build pipelines from containers with `std::views`.

90.9.2 Non-owning views you should combine with ranges

Two important standard library “view” types fit naturally:

- `std::string_view` for non-owning text views,
- `std::span<T>` for non-owning views over contiguous sequences.

Use them to avoid raw pointer + length pairs.

90.10 Performance guidance for ranges

90.10.1 Costs to remember

- Views are typically cheap objects, but iteration can introduce extra layers of indirection.
- Some pipelines generate iterator types that are more complex than plain pointers.
- Materializing can cost allocations and copies but may be required for random access, stability, or repeated passes.

90.10.2 Rule: measure before micro-optimizing

When performance is critical, follow the Core Guidelines principle: do not claim improvements without measurement. Prefer readability and correctness, then measure, then optimize hotspots.

90.11 Common mistakes and how to avoid them

90.11.1 Mistake 1: dangling views

Bad: returning a view that refers to a local container.

```cpp
#include <ranges>
#include <vector>

auto bad_view()
{
    std::vector<int> v{1,2,3,4};
    return v | std::views::filter([](int x){ return x % 2 == 0; }); // dangling
}
```

Fix: return an owning container, or ensure the referenced range outlives the view.

```cpp
#include <ranges>
#include <vector>

std::vector<int> good_result()
{
    std::vector<int> v{1,2,3,4};
    auto view = v | std::views::filter([](int x){ return x % 2 == 0; });
    return {view.begin(), view.end()};
}
```

90.11.2 Mistake 2: assuming a view is multi-pass

Some views are single-pass (input ranges). If you need multi-pass behavior, materialize or ensure the view models a stronger range concept (e.g., `forward_range`).

90.11.3 Mistake 3: side effects in predicates

Predicates/projections may be called in ways you should not depend on. Keep them pure.

90.11.4 Mistake 4: forgetting that algorithms may return `dangling`

If you pass temporaries to constrained algorithms that return iterators, the library may intentionally return `dangling`. Store the range in a variable with sufficient lifetime when you need returned iterators/subranges.

90.12 Practical checklist

- Prefer `std::ranges` algorithms to reduce iterator boilerplate and mismatch bugs.
- Use projections to express "compare/search by field" directly.
- Use views for pipelines; materialize when you need ownership or stable lifetime.
- Treat views like non-owning references: control lifetimes carefully.
- Expect constrained algorithms to return `dangling` on unsafe temporaries.
- Keep predicates/projections side-effect free.
- Measure performance in hotspots; do not guess.

Chapter 91

Avoiding Reinvention

91.1 Purpose

In modern C++, the fastest path to correct, maintainable, and efficient software is to **reuse proven library facilities** instead of rewriting them. The ISO C++ Core Guidelines explicitly recommend using libraries wherever possible and preferring the standard library when it meets your needs.

This chapter explains what "reinvention" looks like in practice, why it is risky, and how to systematically replace handwritten code with standard library facilities (STL containers, algorithms, utilities, and ranges).

91.2 Core Guidelines position

91.2.1 SL.1: Use libraries wherever possible

Rule. Prefer library solutions over writing your own.

Why. Libraries encapsulate expert design, extensive testing, and broad usage. Reimplementing

common components often creates subtle bugs, weaker portability, and higher long-term maintenance cost.

91.2.2 SL.2: Prefer the standard library to other libraries

Rule. If the standard library provides a facility that meets your requirements, prefer it.
Why. Standard facilities are widely available, familiar to C++ developers, and have stable specifications and complexity contracts.

91.2.3 What "avoid reinvention" means

Avoiding reinvention does *not* mean "never write custom code." It means:

- do not rewrite generic building blocks (containers, smart pointers, algorithms, string handling, parsing utilities) without a demonstrated need,
- build your domain logic by composing standard components,
- write custom code only when there is a clear missing requirement (API, performance, constraints, certification, platform).

91.3 Why reinvention is harmful

91.3.1 Correctness risk

Handwritten replacements often miss corner cases:

- integer overflow/underflow cases,
- iterator invalidation and lifetime issues,
- undefined behavior from out-of-bounds access,

- locale/encoding complexities for text,
- concurrency hazards.

91.3.2 Security risk

Many security vulnerabilities come from reinvented low-level utilities:

- buffer overflows from manual arrays,
- use-after-free from raw ownership,
- double-free from ad-hoc resource handling,
- string handling errors (missing termination, wrong bounds).

91.3.3 Performance risk

Reinvented code can be slower:

- library implementations are often optimized across platforms and compilers,
- standard algorithms enable inlining and optimizer recognition patterns,
- containers like `std::vector` provide locality and strong cost models.

91.3.4 Maintenance and portability cost

- standard facilities have stable, portable behavior,
- custom utilities require internal documentation and continuous fixes,
- team onboarding is easier with standard vocabulary types.

91.4 A practical “anti-reinvention” checklist

When you are about to write a utility, ask:

1. Is there an equivalent in the standard library?
2. If not, is there a widely adopted, well-maintained library that fits the constraints?
3. If we still need custom code, can we build it *on top of* standard components?
4. Can we express the intent using standard algorithms/ranges?
5. Can we make ownership/lifetime explicit using standard types?

91.5 Common reinventions and the correct STL replacements

91.5.1 Reinventing dynamic arrays

91.5.1.1 Anti-pattern: manual dynamic array

```
#include <cstddef>

struct IntBuffer {
    int* p = nullptr;
    std::size_t n = 0;

    explicit IntBuffer(std::size_t count) : p(new int[count]), n(count) {}
    ~IntBuffer() { delete[] p; } // copy/move missing, exception safety fragile
};
```

91.5.1.2 Replacement: `std::vector`

```
#include <vector>
#include <cstddef>

std::vector<int> make_buffer(std::size_t n)
{
    return std::vector<int>(n); // owns memory, correct copy/move, safe growth
}
```

Use `std::vector` as the default container unless there is a strong reason to choose otherwise.

91.5.2 Reinventing fixed-size arrays

Use `std::array<T,N>` instead of C arrays when size is fixed.

```
#include <array>

std::array<int, 4> digits{1,2,3,4};
```

91.5.3 Reinventing ownership and lifetime management

91.5.3.1 Anti-pattern: raw owning pointers

```
struct X { int v; };

X* make_x()
{
    return new X{42}; // who deletes? easy to leak
}
```

91.5.3.2 Replacement: `std::unique_ptr` and factories

```
#include <memory>

struct X { int v; };

std::unique_ptr<X> make_x()
{
    return std::make_unique<X>(X{42});
}
```

91.5.4 Reinventing search, counting, and loops

91.5.4.1 Anti-pattern: handwritten search loop

```
#include <vector>

bool contains(const std::vector<int>& v, int x)
{
    for (std::size_t i = 0; i < v.size(); ++i) {
        if (v[i] == x) return true;
    }
    return false;
}
```

91.5.4.2 Replacement: algorithms (and ranges when available)

```
#include <algorithm>
#include <vector>

bool contains(const std::vector<int>& v, int x)
```

```
{
    return std::find(v.begin(), v.end(), x) != v.end();
}
```

```
#include <algorithm>
#include <ranges>
#include <vector>

bool contains_ranges(const std::vector<int>& v, int x)
{
    return std::ranges::find(v, x) != v.end();
}
```

Using algorithms expresses intent and reduces common mistakes.

91.5.5 Reinventing “remove elements”

91.5.5.1 Anti-pattern: manual erase loop with index bugs

```
#include <vector>

void remove_value(std::vector<int>& v, int x)
{
    for (std::size_t i = 0; i < v.size(); ++i) {
        if (v[i] == x) v.erase(v.begin() + i); // BUG: skips elements, quadratic
    }
}
```

91.5.5.2 Replacement: erase-remove idiom

```
#include <algorithm>
```

```cpp
#include <vector>

void remove_value(std::vector<int>& v, int x)
{
    v.erase(std::remove(v.begin(), v.end(), x), v.end());
}
```

91.5.6 Reinventing sorting and ordering

Use `std::sort`, `std::stable_sort`, and related selection algorithms:

- `std::nth_element` for selecting the k-th element without full sorting,
- `std::partial_sort` for top-k,
- binary-search algorithms (`lower_bound`, `upper_bound`) for sorted ranges.

91.5.7 Reinventing string handling and string parameters

91.5.7.1 Use `std::string` for ownership

Avoid manual `char*` management for owned text.

91.5.7.2 Use `std::string_view` for read-only parameters

```cpp
#include <string_view>

bool starts_with(std::string_view s, std::string_view prefix)
{
    return s.size() >= prefix.size() && s.substr(0, prefix.size()) == prefix;
}
```

Rule: `string_view` does not own; the referenced characters must outlive the view.

91.5.8 Reinventing “pointer + length” array parameters

Use `std::span<T>` (C++20) for non-owning views of contiguous sequences.

```
#include <span>

int sum(std::span<const int> a)
{
    int s = 0;
    for (int x : a) s += x;
    return s;
}
```

91.5.9 Reinventing byte buffers

Use `std::byte` for non-character bytes.

```
#include <cstddef>
#include <vector>

void fill_mask(std::vector<std::byte>& buf, std::byte v)
{
    for (auto& b : buf) b = v;
}
```

91.5.10 Reinventing timing and durations

Use `std::chrono` clocks and duration types rather than raw integers.

91.5.11 Reinventing concurrency primitives

Use standard threading primitives where appropriate:

- `std::mutex`, `std::lock_guard`, `std::unique_lock`
- `std::jthread` and `std::stop_token` (C++20)
- `std::atomic` for lock-free synchronization when required

91.6 When reinvention *is* justified

Avoiding reinvention is a guideline, not a prohibition. Writing custom code can be correct when:

- the standard library lacks a required feature (API shape, specialized data structure),
- certified/safety-critical constraints require a specific library subset,
- performance requirements are proven and cannot be met with standard components,
- platform constraints demand a custom allocator, container, or I/O layer,
- you are implementing a domain-specific abstraction on top of standard components.

Even in these cases, prefer to:

- build on standard types and idioms,
- keep custom code small and well-tested,
- document the reason and measurable benefit.

91.7 A disciplined replacement workflow

To migrate legacy code away from reinvention:

1. Identify custom utilities that duplicate standard features.
2. Replace internal interfaces with **standard vocabulary types** (`string`, `vector`, `span`, `string_view`).
3. Replace loops with standard algorithms/ranges where it improves clarity.
4. Add tests around behavior before changing implementations.
5. Measure performance before and after when performance is part of the justification.

91.8 Summary

- The Core Guidelines strongly encourage avoiding reinvention by using library facilities (SL.1) and preferring the standard library (SL.2).
- Use standard vocabulary types (`vector`, `array`, `string`, `span`, `string_view`, smart pointers) to express intent and enforce safety.
- Replace handwritten loops with standard algorithms and ranges where they clarify intent and reduce bugs.
- Reinvent only when there is a documented requirement the standard library cannot satisfy, and verify with tests and measurements.

Part XV

Source Files and Build Structure (SF)

Chapter 92

SF.1 – SF.11: Source Files and Build Structure

92.1 Scope and intent

This chapter covers rules SF.1 through SF.11 from the ISO C++ Core Guidelines section *SF: Source files*. The goal is to keep projects buildable, readable, and maintainable by establishing predictable structure for interfaces (headers) and implementations (source files), and by minimizing hidden build dependencies.

92.2 Background: what these rules protect

In a typical C++ build, each `.cpp` file is compiled as a separate translation unit, and then the linker combines the results. Headers are normally `#included` into multiple translation units. If a header accidentally contains definitions that violate the One Definition Rule (ODR), you can get multiple-definition link errors or, worse, subtle ODR violations.

These rules also reduce *fragile builds*: code that compiles only because it accidentally relies on transitive includes, include order, or implementation details of library headers.

92.3 Recommended project layout (practical baseline)

A conventional, scalable layout aligned with SF.1–SF.11 is:

- **Public interface headers** in a stable include tree (e.g., `include/mylib/...`).
- **Implementation sources** in `src/...`.
- **Private/internal headers** (implementation details) in `src/detail/...` or `include/mylib/detail/...`, depending on whether they are installed/exported.
- **Each** `.cpp` includes its corresponding header first (directly or after minimal configuration headers), so the compiler can validate that the header is sufficient and consistent.

92.4 SF.1: Use a `.cpp` suffix for code files and `.h` for interface files (unless your project already follows another convention)

Rule

Use `.cpp` for implementation files and `.h` for header/interface files, unless your codebase already uses a different consistent convention (e.g., `.cxx`, `.hpp`).

Reason

This is a long-standing convention. The key point is **consistency within a project**.

Example

```
// foo.h:
extern int a;      // declaration
extern void foo(); // declaration

// foo.cpp:
int a;             // definition
void foo() { ++a; }
```

This keeps the interface in the header and the definitions in the source file.

Enforcement

Flag non-conventional file names when the project has an established naming scheme, and check that headers and sources follow the rules below.

92.5 SF.2: A `.h` file must not contain object definitions or non-inline function definitions

Rule

A header must not contain object definitions or non-inline function definitions.

Reason

If multiple translation units include such a header, ODR violations typically become link errors.

Example (bad)

```
// file.h:
```

```
namespace Foo {
    int x = 7;                  // BAD: object definition in a header
    int xx() { return x + x; } // BAD: non-inline function definition in a header
}

// file1.cpp:
#include "file.h"

// file2.cpp:
#include "file.h"
```

Linking `file1.cpp` and `file2.cpp` typically yields multiple-definition errors.

What *is* allowed in headers (practical checklist)

The guidelines give a positive list. A header is typically limited to:

- `#include`s of other headers (with include guards),
- templates,
- class definitions,
- function declarations,
- `extern` declarations,
- `inline` function definitions,
- `constexpr` definitions,
- `const` definitions,
- `using` alias definitions.

Enforcement

Check the positive list above and flag violations.

92.6 SF.3: Use `.h` files for all declarations used in multiple source files

Rule

If a declaration is used in more than one source file, put it in a header.

Reason

Maintainability and readability: the interface has a single authoritative declaration, and changes are made in one place.

Example (bad)

```
// bar.cpp:
void bar() { cout << "bar\n"; }

// foo.cpp:
extern void bar();   // BAD: repeated manual declaration
void foo() { bar(); }
```

Here, `bar`'s declaration can silently diverge from its definition, and errors may appear late (often at link time).

Enforcement

Flag declarations of entities defined in other source files that are not placed in a header.

92.7 SF.4: Include `.h` files before other declarations in a file

Rule

Place `#includes` at the top of the file (both `.h` and `.cpp`).

Reason

Minimizes context dependencies and improves readability.

Example (good)

```
#include <vector>
#include <algorithm>
#include <string>

// ... code ...
```

Example (bad)

```
#include <vector>

// ... code ...

#include <algorithm>
#include <string>
```

Note (important nuance)

There is a known technique of including headers *after* code to reduce exposure to macros and declarations, but the guidelines explain why that is fragile across multiple headers and

translation units; robust solutions ultimately benefit from language-supported modularity.

92.8 SF.5: A .cpp file must include the .h file(s) that defines its interface

Rule

Every implementation file should include the header(s) that declare its interface.

Reason

This enables early consistency checking by the compiler.

Example (bad)

```
// foo.h:
void foo(int);
int bar(long);
int foobar(int);

// foo.cpp:
void foo(int) { /*...*/ }
int bar(double) { /*...*/ }  // mismatch: long vs double
double foobar(int);          // mismatch: int vs double
```

These errors may escape compilation and surface only when code is linked and used.

Example (better)

```
// foo.cpp:
#include "foo.h"
```

```
void foo(int) { /*...*/ }
int bar(double) { /*...*/ }   // still problematic, but easier to detect in practice
double foobar(int);           // caught immediately as a return-type mismatch
```

Including "foo.h" makes several interface mismatches visible earlier.

Enforcement

The guidelines mark enforcement as non-trivial in general. In practice, many teams enforce: "the first include in foo.cpp is foo.h" plus CI checks that compile translation units in isolation.

92.9 SF.6: Use `using namespace` directives for transition, for foundation libraries (such as `std`), or within a local scope only

Rule

Use `using namespace X;` sparingly: mainly for transition, for truly foundational namespaces, or inside local scopes (including a .cpp file).

Reason

It can cause name clashes, but sometimes ubiquitous qualification harms readability in codebases that overwhelmingly use one namespace (commonly the standard library).

Example (acceptable in context)

```
#include <string>
```

```
#include <vector>
#include <iostream>
#include <memory>
#include <algorithm>

using namespace std;

// ...
```

When the standard library dominates and no other competing namespaces exist, this may be a reasonable local choice.

Example (risk)

```
#include <cmath>
using namespace std;

int g(int x)
{
    int sqrt = 7;
    return sqrt(x); // error: name clash (sqrt is now a variable)
}
```

Enforcement

Flag multiple `using namespace` directives for different namespaces in a single source file.

92.10 SF.7: Don't write `using namespace` at global scope in a header file

Rule

Never write `using namespace ...;` at global scope in a header.

Reason

It forces every includer to accept your namespace pollution, removes their ability to disambiguate, and can make include order affect meaning.

Example (bad)

```
// bad.h
#include <iostream>
using namespace std; // BAD in a header

// user.cpp
#include "bad.h"

bool copy(/*...*/); // user's function named copy

int main()
{
    copy(/*...*/);  // may conflict with std::copy; could become ambiguous
}
```

Note (narrow exception)

The guidelines mention a narrow exception: `using namespace std::literals;` can be needed in headers that use standard literals, and user-defined literal naming rules reduce the chance of collision.

Enforcement

Flag any `using namespace` directive at global scope in a header.

92.11 SF.8: Use `#include` guards for all `.h` files

Rule

Every header should have an include guard.

Reason

To prevent a header from being included multiple times in the same translation unit.

Example

```
// foobar.h
#ifndef LIBRARY_FOOBAR_H
#define LIBRARY_FOOBAR_H

// ... declarations ...

#endif // LIBRARY_FOOBAR_H
```

The guidelines recommend guards that include a library/component prefix to reduce collisions (do not rely only on the filename).

Portability note

Some compilers offer `#pragma once`, but the guidelines caution it is not standard and can inject filesystem semantics and vendor lock-in; prefer ISO C++ portability by default.

Enforcement

Flag headers that lack include guards.

92.12 SF.9: Avoid cyclic dependencies among source files

Rule

Avoid include cycles between headers (direct or indirect).

Reason

Cycles slow compilation, make code harder to understand, and complicate future migration to modules.

Note

Do not “solve” cycles by depending on include guards to break them; eliminate the design cycle.

Example (bad)

```
// file1.h
#include "file2.h"

// file2.h
#include "file3.h"

// file3.h
#include "file1.h"
```

Enforcement

Detect and flag all include cycles.

92.13 SF.10: Avoid dependencies on implicitly `#include`d names

Rule

Do not rely on transitive includes (names that appear only because another header happened to include what you needed). Always include what you use.

Reason

This prevents surprises, avoids fragile dependencies on implementation details, and reduces churn when headers change their internal includes.

Example (bad)

```
#include <iostream>

using namespace std;

void use()
{
    string s;
    cin >> s;              // may compile
    getline(cin, s);       // error if <string> wasn't included
    if (s == "surprise") { // error if required operators weren't visible
        // ...
```

```
    }
}
```

The guideline points out that some headers may expose types without guaranteeing the full required interface (e.g., `getline`) unless the correct header is included.

Example (good)

```
#include <iostream>
#include <string>

using namespace std;

void use()
{
    string s;
    cin >> s;
    getline(cin, s);
    if (s == "surprise") {
        // ...
    }
}
```

Note (allowed aggregation headers)

The rule does *not* forbid deliberate aggregation headers (project headers that intentionally include a curated set of standard/library headers). It forbids *accidental* dependence on whatever a header currently happens to include.

Enforcement (realistic approach)

Perfect enforcement requires knowing which names are intended to be part of a header's interface versus incidental; the guidelines note this becomes much easier with modules. A practical enforcement strategy is:

- Build with "include what you use" discipline (explicit includes in each file).
- Add CI jobs that compile translation units after shuffling include order (within reason) and that compile headers in isolation (see SF.11).

92.14 SF.11: Header files should be self-contained

Rule

A header should compile when included on its own; it must include the headers it depends on.

Reason

This improves usability and prevents hard-to-diagnose errors in client code. Clients should not have to guess which prerequisite headers to include first.

Example

```
#include "helpers.h"
// helpers.h uses std::string and therefore includes <string>
```

If `helpers.h` uses `std::string`, it should include `<string>` itself.

Note (paths)

The guidelines warn to be careful with relative include paths because implementations may differ in how they interpret them; aim for a consistent include policy in the build system.

Enforcement (high value, easy to automate)

Add a test that verifies:

- each public header compiles by itself (or),
- a `.cpp` that includes only that header compiles.

This is one of the most effective ways to prevent SF.10 and SF.11 regressions.

92.15 Build-structure checklist (practical summary)

To make SF.1–SF.11 actionable in real projects:

- **Interfaces in headers, implementations in sources:** keep headers free of non-inline definitions (SF.2).
- **One declaration of an interface:** shared declarations live in headers (SF.3).
- **Predictable includes:** includes at the top; in `.cpp`, include the corresponding header early (SF.4, SF.5).
- **No global namespace pollution from headers:** avoid `using namespace` in headers (SF.7).
- **Defensive header hygiene:** include guards everywhere, no cycles, no transitive-include reliance, and each header compiles standalone (SF.8–SF.11).

92.16 Tool support (adoption at scale)

Many teams enforce these rules with static analysis and build checks:

- **Clang-tidy** provides checks aligned with the C++ Core Guidelines (rules named `cppcoreguidelines-*`).
- **MSVC code analysis** includes "C++ Core Guidelines checkers" support as part of Visual Studio tooling.
- **CI header compilation tests** (SF.11) and include-cycle detection (SF.9) catch problems earlier than link time and reduce platform-specific surprises.

Chapter 93

File organization

93.1 Purpose

File organization is not cosmetic. In C++, it directly affects:

- correctness (ODR safety, avoiding accidental multiple definitions),
- build reliability (avoiding fragile transitive includes),
- maintainability (clear ownership of interfaces vs implementations),
- compile times and scalability (controlled dependencies).

This chapter provides a practical, modern file-organization blueprint aligned with the *Source Files and Build Structure (SF)* guidance in the C++ Core Guidelines, with engineering-grade conventions that scale from small libraries to large systems.

93.2 Core principle: separate interface from implementation

93.2.1 Interface lives in headers

Headers represent **interfaces**: declarations and types that other translation units consume.

93.2.2 Implementation lives in source files

Source files contain **definitions**: non-inline functions, non-constexpr objects, and other implementation details.

93.2.3 Why this matters (ODR and link-time failures)

A definition that appears in multiple translation units can violate the One Definition Rule (ODR), producing:

- immediate multiple-definition linker errors, or
- worse, silently inconsistent programs when the violation is not diagnosed.

93.3 A scalable directory layout

A widely used structure that works well with both library and application code:

```
project-root/
  include/
    mylib/
      api.h
      widget.h
      detail/          (optional: installed, but "not for users")
        widget_impl.h
  src/
    widget.cpp
    widget_internal.h (internal-only, not installed)
```

```
tests/
  test_widget.cpp
examples/
  demo.cpp
tools/              (optional: generators, scripts)
cmake/              (optional: toolchain/cmake modules)
CMakeLists.txt      (or alternative build system files)
```

93.3.1 Public vs private headers

- **Public headers** (installed/exported): placed under `include/mylib/`.
- **Private headers** (internal-only): placed under `src/` or a clearly marked internal folder.
- **Detail headers** (advanced): may be installed but must be clearly documented as unstable (`detail/`).

93.4 Header content rules (what belongs in a header)

A header should primarily contain:

- declarations (`class`, `struct`, function declarations),
- templates,
- `inline` functions,
- `constexpr` entities,
- `extern` declarations (not definitions),
- type aliases (`using`).

93.4.1 Avoid definitions that cause ODR problems

Do not place non-inline function definitions or non-const non-`inline` objects in a header.

93.4.2 Safe header constants: modern patterns

Prefer one of these patterns:

```
// In a header:

// 1) inline variable (C++17 and later)
inline constexpr int max_items = 1024;

// 2) constexpr (implicitly internal linkage for some cases, but keep it simple)
constexpr double pi = 3.141592653589793;

// 3) enum for integral constants (works everywhere)
enum : int { default_timeout_ms = 2000 };
```

93.5 Make headers self-contained

93.5.1 Rule

A header should compile when included alone. It must include everything it needs.

93.5.2 Practical checklist

If `mylib/widget.h` uses:

- `std::string` ⇒ it includes `<string>`,
- `std::vector` ⇒ it includes `<vector>`,

- `std::unique_ptr` ⇒ it includes `<memory>`.

93.5.3 A robust pattern: include what you use

```
// widget.h
#ifndef MYLIB_WIDGET_H
#define MYLIB_WIDGET_H

#include <string>
#include <vector>

namespace mylib {

class widget {
public:
    explicit widget(std::string name);
    void add_tag(std::string tag);
    const std::string& name() const noexcept;

private:
    std::string name_;
    std::vector<std::string> tags_;
};

} // namespace mylib

#endif // MYLIB_WIDGET_H
```

93.6 Include guards and portability

93.6.1 Rule

Every header should use include guards.

93.6.2 Guard naming

Use a project-specific prefix to reduce collision risk:

```
#ifndef MYLIB_WIDGET_H
#define MYLIB_WIDGET_H
// ...
#endif
```

93.6.3 About `#pragma once`

Many compilers support `#pragma once`, but it is not part of the ISO C++ standard. For portable, standard-oriented codebases, include guards remain the default.

93.7 Include order and consistency

93.7.1 Place includes first

Put `#includes` at the top of `.h` and `.cpp`.

93.7.2 In `.cpp`, include the corresponding header first

This catches missing includes and mismatch errors early.

```
// widget.cpp
#include "mylib/widget.h"

#include <utility>
```

```
namespace mylib {

widget::widget(std::string name) : name_(std::move(name)) {}

void widget::add_tag(std::string tag) {
    tags_.push_back(std::move(tag));
}

const std::string& widget::name() const noexcept {
    return name_;
}

} // namespace mylib
```

93.8 Avoid transitive-include dependencies

93.8.1 The problem

Code that compiles only because another header *happened* to include what you needed becomes fragile:

- it breaks when headers change,
- it breaks across platforms and standard library implementations,
- it breaks when include order changes.

93.8.2 Rule of thumb

If your file uses a name, it includes the header that provides it.

93.9 Avoid cyclic header dependencies

93.9.1 Why cycles are harmful

Include cycles:

- slow builds,
- complicate dependency reasoning,
- make refactoring harder,
- obstruct migration to modular builds.

93.9.2 How to break cycles safely

- Use forward declarations where valid (for pointers/references).
- Move shared abstractions to a smaller header.
- Introduce a stable interface boundary.
- Use PImpl to hide heavy/private dependencies.

93.10 Forward declarations vs includes

93.10.1 When forward declarations are appropriate

Forward-declare a type **only** when you need it as:

- a pointer/reference member,
- a pointer/reference parameter,

- an incomplete type in the interface (e.g., PImpl).

93.10.2 Example: forward declaration with PImpl

```
// widget.h
#ifndef MYLIB_WIDGET_H
#define MYLIB_WIDGET_H

#include <memory>
#include <string>

namespace mylib {

class widget_impl; // forward declaration

class widget {
public:
    explicit widget(std::string name);
    widget(widget&&) noexcept;
    widget& operator=(widget&&) noexcept;
    ~widget(); // defined in .cpp where widget_impl is complete

    const std::string& name() const noexcept;

private:
    std::unique_ptr<widget_impl> p_; // keeps implementation out of the header
};

} // namespace mylib

#endif
```

```
// widget.cpp
```

```
#include "mylib/widget.h"

#include <utility>

namespace mylib {

class widget_impl {
public:
    explicit widget_impl(std::string n) : name(std::move(n)) {}
    std::string name;
};

widget::widget(std::string name)
    : p_(std::make_unique<widget_impl>(std::move(name))) {}

widget::widget(widget&&) noexcept = default;
widget& widget::operator=(widget&&) noexcept = default;
widget::~widget() = default;

const std::string& widget::name() const noexcept { return p_->name; }

} // namespace mylib
```

93.10.3 When forward declarations are *not* enough

You must include the full header when you need:

- object members by value (complete type required),
- inheritance (base class must be complete at class definition),
- inline functions that use the full type,
- `sizeof(T)`, `alignof(T)`, or deleting through a complete type requirement.

93.11 Namespaces and headers

93.11.1 No `using namespace` in headers

Do not place `using namespace ...;` at global scope in a header. It forces all users of your header to accept your namespace choices and can change the meaning of their code.

93.11.2 Preferred style in headers

Use qualified names:

```
std::string make_name();
```

93.11.3 Local allowances in `.cpp`

In `.cpp` files, a limited `using namespace` can be acceptable when it improves readability and does not escape into headers.

93.12 Configuration headers and build-generated headers

93.12.1 Configuration header policy

If your project uses a configuration header (feature flags, platform macros), keep it:

- small,
- stable,
- included in a controlled way (preferably from `.cpp` or a single top-level public header if required).

93.12.2 Do not leak build-system details into public interfaces

Avoid exposing macros that change ABI or class layout in public headers unless you have a strict compatibility strategy.

93.13 Precompiled headers and unity builds (build-time techniques)

93.13.1 Precompiled headers (PCH)

PCH can improve build times, but it does not replace correct include hygiene:

- headers must remain self-contained,
- files must not rely on transitive includes.

93.13.2 Unity builds

Unity builds (compiling multiple `.cpp` files together) can reduce build time but also:

- hide missing includes,
- make include order and macro leakage more dangerous.

If you use unity builds, keep an independent build mode that compiles each translation unit separately in CI.

93.14 Modules note (C++20 and later)

C++20 modules can reduce many problems caused by textual inclusion, but most real projects still contain headers for:

- third-party dependencies,
- platform SDKs,
- compatibility layers,
- gradual migration strategies.

Therefore, disciplined file organization remains essential even when adopting modules incrementally.

93.15 Automated enforcement

93.15.1 High-value CI checks

- Compile a small translation unit for each public header:

```
// header_selftest.cpp
#include "mylib/widget.h"
int main() {}
```

- Detect include cycles.
- Build with at least one configuration that compiles each `.cpp` separately (no unity build).

93.15.2 Static analysis

- **Clang-Tidy** provides checks named `cppcoreguidelines-*`.
- **MSVC Code Analysis** includes C++ Core Guidelines checker support for a subset of rules.

Chapter 94

Header files

94.1 Purpose of header files

In C++, header files are the primary mechanism for sharing declarations across translation units. A good header design makes code:

- correct (avoids ODR and linkage problems),
- robust (does not depend on include order or transitive includes),
- scalable (keeps dependencies and build times under control),
- usable (clients can include a header without guessing prerequisites).

These goals align directly with the C++ Core Guidelines section *SF: Source files*, especially SF.2, SF.7–SF.11.

94.2 What belongs in a header

94.2.1 Header files express interfaces

A header should primarily contain declarations that define an interface:

- `class`/`struct` declarations and their member declarations,
- function declarations,
- type aliases (`using`),
- templates (declarations and definitions),
- `inline` function definitions,
- `constexpr` entities intended to be shared,
- `extern` declarations (not definitions).

This supports the guideline principle: keep interfaces stable, and keep implementations out of headers unless they must be visible (templates, `inline`).

94.2.2 SF.2: Do not put non-inline definitions in headers

Rule (SF.2): A `.h` file must not contain object definitions or non-inline function definitions.

94.2.2.1 Why this rule exists

Headers are typically included in many translation units. If a header contains a definition with external linkage (for example, a non-inline function definition or a non-inline global object definition), then each translation unit may emit the same definition, violating the One Definition Rule (ODR).

94.2.2.2 Bad example: definitions in a header

```
// bad.h
#ifndef MYLIB_BAD_H
#define MYLIB_BAD_H

namespace mylib {
    int counter = 0;                 // BAD: object definition in a header
    int next_id() { return ++counter; } // BAD: non-inline function definition
}

#endif
```

94.2.2.3 Good example: declarations in the header, definitions in a source file

```
// good.h
#ifndef MYLIB_GOOD_H
#define MYLIB_GOOD_H

namespace mylib {
    extern int counter;  // declaration only
    int next_id();       // declaration only
}

#endif
```

```
// good.cpp
#include "good.h"

namespace mylib {
    int counter = 0;     // single definition (one translation unit)
    int next_id() { return ++counter; }
}
```

94.2.3 Safe shared constants in headers (modern patterns)

When you want a constant available in many translation units, prefer forms that do not create multiple external definitions.

94.2.3.1 Preferred: `inline constexpr` variables (C++17 and later)

```
// constants.h
#ifndef MYLIB_CONSTANTS_H
#define MYLIB_CONSTANTS_H

namespace mylib {
    inline constexpr int max_items = 1024;
    inline constexpr double pi = 3.14159265358979323846;
}

#endif
```

94.2.3.2 Alternative: `enum` constants (works in all C++ versions)

```
enum : int { default_timeout_ms = 2000 };
```

94.3 Headers must be self-contained

94.3.1 SF.11: A header should compile on its own

Rule (SF.11): Header files should be self-contained. A user should be able to include the header without needing to include other headers first.

94.3.2 SF.10: Do not depend on transitive includes

Rule (SF.10): Avoid dependencies on implicitly included names. If you use a facility, include the header that provides it.

94.3.3 Example: self-contained public header

```
// widget.h
#ifndef MYLIB_WIDGET_H
#define MYLIB_WIDGET_H

#include <string>
#include <vector>

namespace mylib {

class widget {
public:
    explicit widget(std::string name);
    void add_tag(std::string tag);

    const std::string& name() const noexcept;
    const std::vector<std::string>& tags() const noexcept;

private:
    std::string name_;
    std::vector<std::string> tags_;
};

} // namespace mylib

#endif // MYLIB_WIDGET_H
```

This header includes <string> and <vector> because it uses std::string and std::vector in the interface. This directly prevents fragile "works by accident" builds.

94.3.4 Recommended CI check: compile each header alone

A high-value build check is to compile a tiny translation unit per public header:

```
// header_selftest_widget.cpp
#include "mylib/widget.h"
int main() {}
```

This detects violations of SF.10 and SF.11 early and consistently.

94.4 Include guards and portability

94.4.1 SF.8: Use include guards for all headers

Rule (SF.8): Use #include guards for all .h files.

94.4.2 Why include guards are the baseline

Include guards are standard, portable, and prevent repeated inclusion of the same header within a translation unit.

94.4.3 Guard naming

Use names that minimize collision risk (project prefix + path-derived name is common):

```
#ifndef MYLIB_DETAIL_WIDGET_IMPL_H
#define MYLIB_DETAIL_WIDGET_IMPL_H
// ...
#endif
```

94.4.4 About `#pragma once`

Some compilers support `#pragma once`, but the Core Guidelines explicitly prefer ISO C++ portability and describe `#pragma once` as non-standard and non-portable in principle.

94.5 Include placement and include order

94.5.1 SF.4: Put includes first

Place `#include`s at the top of the file to make dependencies explicit and to avoid context-sensitive code.

94.5.2 SF.5: In a `.cpp`, include the matching header

A practical rule that strengthens header correctness is: the `.cpp` includes its corresponding header (and often includes it first). This catches missing includes and interface mismatches earlier in the build.

```
// widget.cpp
#include "mylib/widget.h"   // include the interface first

#include <utility>

namespace mylib {

widget::widget(std::string name) : name_(std::move(name)) {}

void widget::add_tag(std::string tag) {
    tags_.push_back(std::move(tag));
}
```

```
const std::string& widget::name() const noexcept { return name_; }
const std::vector<std::string>& widget::tags() const noexcept { return tags_; }

} // namespace mylib
```

94.6 Namespace rules for headers

94.6.1 SF.7: Never use `using namespace` at global scope in a header

Rule (SF.7): Do not write `using namespace` at global scope in a header file.

94.6.2 Why this is dangerous

A header is included by many clients. A global `using namespace` directive forces namespace pollution into every includer and can change meaning, create ambiguity, or break unrelated code.

94.6.3 Bad example

```
// bad_header.h
#ifndef MYLIB_BAD_HEADER_H
#define MYLIB_BAD_HEADER_H

#include <algorithm>
using namespace std; // BAD in a header

#endif
```

94.6.4 Acceptable alternatives

- Fully qualify names in headers: `std::string`, `std::vector`, `std::size_t`.

- If needed, use narrow `using` declarations inside a class or a function (local scope), not at global scope in a header.

94.6.5 Narrow exception mentioned by the guidelines

The guidelines note that `using namespace std::literals;` can be necessary in headers that use standard literal suffixes, and that user-defined literal naming rules reduce collision risk. Use this only when you truly expose literals in the header interface.

94.7 Forward declarations, incomplete types, and dependency control

94.7.1 Why forward declarations matter

Forward declarations reduce header dependencies and can significantly improve build times by avoiding unnecessary includes. They also help prevent cyclic include dependencies (SF.9).

94.7.2 When forward declarations are safe

Forward-declare a type in a header when the interface uses it only as:

- a pointer or reference,
- an opaque handle (PImpl),
- a parameter type by reference/pointer.

94.7.3 PImpl pattern to keep heavy includes out of headers

```
// widget2.h
```

```
#ifndef MYLIB_WIDGET2_H
#define MYLIB_WIDGET2_H

#include <memory>
#include <string>

namespace mylib {

class widget2_impl; // forward declaration

class widget2 {
public:
    explicit widget2(std::string name);
    widget2(widget2&&) noexcept;
    widget2& operator=(widget2&&) noexcept;
    ~widget2(); // defined in .cpp after widget2_impl is complete

    const std::string& name() const noexcept;

private:
    std::unique_ptr<widget2_impl> p_;
};

} // namespace mylib

#endif
```

```
// widget2.cpp
#include "mylib/widget2.h"

#include <utility>

namespace mylib {
```

```
class widget2_impl {
public:
    explicit widget2_impl(std::string n) : name(std::move(n)) {}
    std::string name;
};

widget2::widget2(std::string name)
    : p_(std::make_unique<widget2_impl>(std::move(name))) {}

widget2::widget2(widget2&&) noexcept = default;
widget2& widget2::operator=(widget2&&) noexcept = default;
widget2::~widget2() = default;

const std::string& widget2::name() const noexcept { return p_->name; }

} // namespace mylib
```

94.7.4 When you must include the full definition in the header

You need the complete type in a header when:

- a member is stored by value,
- you derive from a base class,
- you define inline functions that access the full type,
- you use `sizeof`, `alignof`, or delete an object requiring a complete type.

94.8 Macro hygiene in headers

94.8.1 Why macros are risky in interfaces

Macros are not scoped, can change meanings unexpectedly, and can create include-order dependencies. Prefer:

- `constexpr` variables,
- `enum` constants,
- inline functions,
- templates,
- named configuration APIs instead of preprocessor switches.

94.8.2 If you must use macros

- Use a strict project prefix (`MYLIB_...`).
- Keep macro lifetimes as short as possible: define, use, then `#undef` in the same header when feasible.
- Avoid macros that change class layout or calling conventions in public headers unless you have a documented ABI strategy.

94.9 Cyclic include dependencies (SF.9)

94.9.1 What to avoid

Cyclic dependencies among headers make code hard to reason about and can hide design problems. The guidelines explicitly recommend avoiding such cycles.

94.9.2 How to break cycles

- Introduce a smaller abstraction header used by both sides.
- Replace by-value members with pointers/references plus forward declarations (when ownership/lifetime rules are clear).
- Use PImpl to isolate private dependencies.

94.10 Templates and inline functions in headers

94.10.1 Why templates are normally defined in headers

Templates are instantiated where used. Therefore, their definitions must typically be visible to the translation unit that instantiates them (unless you use explicit instantiation strategies).

94.10.2 Inline functions: what “inline” really means here

For header design, `inline` primarily allows a function definition to appear in multiple translation units without violating the ODR, as long as the definitions are the same. It does not guarantee performance inlining.

94.10.3 Guideline for header-only utilities

Keep header-only functions small, stable, and free of heavy dependencies. If an inline function requires many heavy includes, prefer moving it to a source file (or hiding it behind PImpl).

94.11 Interaction with C++20 modules and header units

94.11.1 Modules reduce many header problems, but do not remove the need for discipline

C++20 introduces modules as a language feature that can reduce the cost and fragility of textual inclusion. Many codebases will still use headers for a long time because of existing libraries, platform SDKs, and incremental migration constraints.

94.11.2 Important practical note: macros and standard library modules

C++23 standard library modules export the standard library contents except for parts defined as macros; macros remain a special case in modular builds. This makes macro hygiene even more important when designing public interfaces.

94.12 Tooling and enforcement

94.12.1 Clang-Tidy

Clang-Tidy provides checks related to header hygiene and guideline enforcement (including checks aligned with the Core Guidelines).

94.12.2 MSVC code analysis

Visual Studio supports a subset of C++ Core Guidelines rules via its C++ code analysis tooling ("Core Guidelines checkers").

94.12.3 Recommended enforcement policy for teams

- Every public header has include guards (SF.8).
- No global `using namespace` directives in headers (SF.7).

- Headers compile standalone (SF.11).
- Each `.cpp` includes its own header (SF.5).
- No reliance on transitive includes (SF.10).
- Include cycle detection is enabled in CI (SF.9).

94.13 Checklist: what a professional header should guarantee

A professional public header should satisfy all of the following:

- It has include guards (SF.8).
- It is self-contained: compiles when included alone (SF.11).
- It includes what it uses; no transitive-include dependence (SF.10).
- It contains no non-inline function definitions and no non-inline object definitions (SF.2).
- It does not use `using namespace` at global scope (SF.7).
- It avoids cyclic dependencies and limits includes (SF.9).
- Its interface minimizes macros and avoids macro-based ABI changes unless explicitly designed.

Chapter 95

Include guards

95.1 What an include guard is and why it exists

An *include guard* is a small preprocessor pattern placed in a header file to ensure that the header's contents are processed at most once per translation unit, even if the header is included multiple times through different include paths.
This supports three core goals emphasized by the C++ Core Guidelines:

- **Correctness:** prevent redefinition errors caused by repeated inclusion.
- **Robust builds:** reduce sensitivity to include order and complex include graphs.
- **Scalability:** keep large projects buildable as dependencies grow.

95.2 SF.8: Use `#include` guards for all `.h` files

95.2.1 Rule

Every header should use include guards.

95.2.2 Reason

Headers are typically included by many translation units and often indirectly through other headers. Without a guard, repeated inclusion can cause:

- class/struct redefinition errors,
- duplicate declarations with conflicting meaning,
- macro redefinition conflicts,
- subtle differences when macros alter header behavior across multiple includes.

The guard makes repeated inclusion safe and predictable.

95.3 How include guards work (mechanics)

Include guards rely on the preprocessor macro table inside one translation unit.

95.3.1 Canonical pattern

```
// file: mylib/widget.h
#ifndef MYLIB_WIDGET_H
#define MYLIB_WIDGET_H

// declarations, types, templates, inline functions...

#endif // MYLIB_WIDGET_H
```

95.3.2 Execution model

- On first inclusion, `MYLIB_WIDGET_H` is not defined, so the header body is processed and the macro becomes defined.

- On later inclusions in the same translation unit, the macro is already defined, so the header body is skipped.

95.4 Guard macro naming: make collisions practically impossible

95.4.1 Why uniqueness matters

If two different headers accidentally use the same guard macro name, then including one header can silently prevent inclusion of the other. That can lead to:

- missing declarations,
- confusing compilation errors far from the cause,
- platform-dependent behavior (because include graphs differ).

95.4.2 Recommended naming strategy

Use a **project-specific prefix** plus a path-derived name:

- Project prefix: `MYLIB_`
- Include path: `mylib/detail/widget_impl.h`
- Guard: `MYLIB_DETAIL_WIDGET_IMPL_H`

95.4.3 Example: public header

```
// file: include/mylib/api.h
#ifndef MYLIB_API_H
```

```
#define MYLIB_API_H

namespace mylib {
    int version_major() noexcept;
    int version_minor() noexcept;
}

#endif // MYLIB_API_H
```

95.4.4 Example: internal/detail header

```
// file: include/mylib/detail/abi_config.h
#ifndef MYLIB_DETAIL_ABI_CONFIG_H
#define MYLIB_DETAIL_ABI_CONFIG_H

// Internal configuration details

#endif // MYLIB_DETAIL_ABI_CONFIG_H
```

95.5 Guard placement and structure rules

95.5.1 Place the guard at the very top

Put the `#ifndef/#define` immediately after a short file banner (if any), before most other directives and declarations, so the header is skipped as early as possible.

95.5.2 Do not wrap includes unnecessarily

It is common to place the guard around the entire header including `#includes`:

```
#ifndef MYLIB_WIDGET_H
#define MYLIB_WIDGET_H
```

```
#include <string>
#include <vector>

// declarations...

#endif
```

This is correct and standard.

95.5.3 Avoid multiple guards in a single header

One header should have exactly one guard macro for clarity and predictable behavior.

95.6 Include guards and self-contained headers (SF.10 and SF.11)

Include guards solve *repeat-inclusion* problems, but they do not solve:

- missing includes (depending on transitive includes),
- cyclic dependencies,
- incomplete or non-self-contained headers.

Therefore, include guards should be used together with these disciplines:

- **Include what you use (SF.10):** do not rely on transitive includes.
- **Headers are self-contained (SF.11):** each header compiles when included alone.

95.7 Common include-guard pitfalls (and how to avoid them)

95.7.1 Pitfall 1: guard name collision

Symptom: a header appears to be ignored, leading to missing type/function declarations.
Prevention:

- Use a project prefix.
- Derive the rest from the full include path, not just the filename.

95.7.2 Pitfall 2: inconsistent include paths

Symptom: the same physical file is included through different paths, producing confusion in diagnostics and build tooling.
Prevention:

- Establish a single include style for your project, such as `#include "mylib/widget.h"`.
- Avoid mixing deep relative paths and installed-style include paths for the same file.

95.7.3 Pitfall 3: macro-dependent headers

Symptom: a header's contents change depending on macros defined before inclusion.
Why guards matter here: Once the header is included the first time, further includes are skipped, so later macro changes will not affect it. That can be good (stability) or dangerous (surprising configuration behavior).
Prevention:

- Keep macro configuration out of public headers when possible.
- Prefer `constexpr` options, typed configuration objects, or separate configuration headers with clear rules.

95.8 `#pragma once` and include guards

95.8.1 What `#pragma once` is

`#pragma once` is a widely supported but **non-standard** directive intended to ensure a header is included once per translation unit.

95.8.2 Core Guidelines position

The C++ Core Guidelines recommend using ISO C++ mechanisms and explicitly describe `#pragma once` as not standard and not portable, noting that it can inject filesystem semantics into the program's meaning.

95.8.3 MSVC practical note

Microsoft documents that they recommend the include-guard idiom when portability is required and also describes real-world scenarios (such as filesystem aliasing or aliased include paths) where `#pragma once` optimizations can be problematic.

95.8.4 Engineering guidance

- For portable, standards-first code (especially libraries): use include guards everywhere.
- If a project intentionally standardizes on `#pragma once`, adopt it consistently and validate builds under your supported compilers and filesystem layouts, but understand it is not ISO C++ and may fail in specialized environments.

95.9 Performance considerations (what matters and what does not)

95.9.1 Guard form

Both of the following are correct:

```
#ifndef MYLIB_X_H
#define MYLIB_X_H
// ...
#endif
```

```
#if !defined(MYLIB_X_H)
#define MYLIB_X_H
// ...
#endif
```

In practice, modern compilers are good at optimizing repeated include processing. The main performance win still comes from reducing unnecessary includes and breaking include cycles, not from micro-optimizing guard syntax.

95.9.2 Build-time strategies do not replace guards

Techniques such as precompiled headers or unity builds can reduce build time, but they do not remove the need for correct include-guard and header-hygiene practices. Guards are about correctness and predictable semantics.

95.10 Testing and enforcement in real projects

95.10.1 Header self-containment test (high value)

For each public header, compile a minimal translation unit:

```
// header_selftest_api.cpp
#include "mylib/api.h"
int main() {}
```

This catches missing prerequisites and many ordering dependencies early (supports SF.10 and SF.11).

95.10.2 Static analysis support

- Clang-Tidy includes a portability check that detects `#pragma once` usage and suggests standard include guards in portability-focused codebases.
- The C++ Core Guidelines also describe tool support (Clang-Tidy and Microsoft's checkers) for broader guideline enforcement.

95.11 Practical checklist

A professional header should satisfy all of the following:

- Exactly one include guard macro, near the top of the file.
- Guard macro name is globally unique within the project (prefix + path-derived name).
- No `using namespace` directives at global scope in headers.
- Header includes what it uses (no transitive-include dependence).
- Header compiles when included alone (self-contained).

Chapter 96

Modules (where applicable)

96.1 Why modules matter to file and build structure

C++ modules (standardized since C++20) provide a language-supported alternative to many traditional uses of textual headers. Instead of repeatedly `#include`ing headers into every translation unit, a module interface can be compiled once and then `import`ed by consumers. In the context of *Source Files and Build Structure (SF)*, modules are relevant because they can:

- reduce fragile build dependencies caused by transitive includes and include order,
- reduce repeated parsing work and improve build scalability,
- reduce accidental macro and preprocessor leakage across interfaces.

At the same time, real projects often need to mix modules and headers for compatibility, third-party libraries, and incremental migration.

96.2 What the C++ standard calls “modules”

In standard terms, modules are composed of two related mechanisms:

- **Named modules:** module interface units, implementation units, and optional partitions.
- **Header units:** a way to `import` a header as a unit (still ultimately based on header content, but integrated into the module model).

This distinction is used explicitly in the Clang documentation for Standard C++ Modules.

96.3 Named modules: the core building block

96.3.1 Module interface unit

A *module interface unit* declares the module name and specifies what it exports.

```
// file: math.ixx  (extension is project-specific; the content is what matters)
export module math;

export int add(int a, int b);
export int sub(int a, int b);
```

96.3.2 Module implementation unit

A *module implementation unit* provides definitions for exported declarations (and may contain non-exported implementation details).

```
// file: math.cpp
module math;

int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }
```

This parallels the header/source split, but uses `export module` and `module` instead of `.h/.cpp`.

96.3.3 Importing a named module

```
// file: main.cpp
import math;

#include <iostream> // still fine in mixed codebases

int main() {
    std::cout << add(2, 3) << '\n';
}
```

96.3.4 Module partitions (optional)

Partitions let you split a module into multiple interface/implementation pieces while keeping a single logical module name.

```
// file: math_ops_part.ixx
export module math:ops;

export int mul(int a, int b);
```

```
// file: math_ops_part.cpp
module math:ops;

int mul(int a, int b) { return a * b; }
```

```
// file: math.ixx
export module math;

export import :ops;   // re-export the partition's exports
export int add(int a, int b);
```

96.4 Header units: importing headers

96.4.1 What a header unit is

A *header unit* allows importing a header using `import <header>;` or `import "header";` (subject to toolchain and build support). Conceptually, it treats a header as an importable unit in the modules system.

96.4.2 Example

```
import <vector>;
import <string>;

int main() {
    std::vector<std::string> v;
}
```

96.4.3 Important practical note

Header units do not magically fix poor header hygiene. If a header relies on macros, include order, or transitive includes, those issues can still surface (sometimes differently) when the header is turned into a header unit. For this reason, SF rules about self-contained headers and include hygiene remain valuable even in modular codebases.

96.5 The global module fragment and mixing with legacy headers

96.5.1 Why it exists

Modules aim to reduce exposure to macros and textual inclusion, but real code still needs headers (platform SDKs, third-party libraries, configuration headers).
C++ provides a *global module fragment* syntax to include headers *before* the module declaration in a module interface unit, isolating some preprocessor effects from the module itself.

96.5.2 Example

```
// file: net.ixx
module;                 // global module fragment (not exported)
#include <cstdint>       // legacy header needed for types
#include "legacy_api.h"  // third-party header (non-modular)

export module net;

export std::uint32_t checksum(const unsigned char* data, std::size_t n);
```

This supports incremental adoption while preserving a cleaner exported surface. The details of module interface/implementation organization are described in Microsoft's modules overview, and the general module model is captured by standard references.

96.6 Standard library modules (where supported)

96.6.1 What is standardized vs what is provided

The C++ standard defines the language feature of modules. Whether your standard library ships named modules for `std` is an implementation and toolchain matter, and it depends on compiler and library versions.

96.6.2 MSVC named STL modules: `std` and `std.compat`

Microsoft documents two named modules:

- `std`
- `std.compat`

and provides guidance for importing the standard library using modules.

96.6.3 Example (MSVC documentation model)

```
import std;

int main() {
    std::vector<int> v{1,2,3};
}
```

96.6.4 Engineering guidance

- Prefer `import std;` only when your toolchain clearly supports it and your build system is configured to handle module dependency scanning and BMI (binary module interface) generation.
- Otherwise, continue using standard headers with `#include`, and adopt named modules primarily for your own project interfaces first.

96.7 Build system reality: modules require dependency scanning

96.7.1 Why modules change the build graph

With headers, inclusion is textual and each translation unit can be compiled independently (though often slowly). With named modules, `import` establishes compilation ordering constraints:

- a module interface must be compiled before any translation unit that imports it,
- partitions and re-exports introduce additional ordering rules.

Build systems must discover and respect these constraints reliably.

96.7.2 CMake support

CMake documents first-class C++ module build support in its `cmake-cxxmodules(7)` manual (added in CMake 3.28). It explains that CMake asks the compiler to scan sources for module dependencies, infers ordering constraints, and updates the build graph accordingly.

96.7.3 Practical implication

To use modules in a portable way:

- you need a compiler with usable module support,
- you need a build system that understands module dependency scanning (or you must manage ordering manually),
- you should expect some toolchain- and build-specific configuration.

96.8 Toolchain status (high-level, standards-based guidance)

96.8.1 Do not assume uniform support

Even in late 2025, module support quality varies by compiler, library, and build integration. For example, GCC describes its C++20 modules support as experimental and indicates it requires enabling via compiler flags.

96.8.2 Authoritative reference points

- **GCC:** C++ standards support and notes about modules enablement and maturity are maintained on GCC's C++ status page.
- **Clang/LLVM:** Standard C++ Modules documentation describes named modules and header units and provides implementation guidance.
- **MSVC:** Microsoft documents module concepts, file roles, and standard library module importing (when supported by the installed STL).

96.9 File organization patterns for modular projects

96.9.1 Recommended directory layout

A practical layout that coexists with headers:

```
project-root/
  include/                (public headers that still exist)
    mylib/
      legacy_api.h
  modules/                (named module interfaces)
    mylib.core.ixx
    mylib.io.ixx
    mylib.io_part.ixx
```

```
src/                    (module implementation units and classic .cpp)
  mylib.core.cpp
  mylib.io.cpp
  legacy_bridge.cpp
tests/
CMakeLists.txt
```

96.9.2 Naming guidance

- Use a consistent file extension policy (e.g., `.ixx`, `.cppm`, or `.mpp`). Consistency matters more than the specific suffix.
- Mirror module names in paths and filenames to make dependency graphs discoverable by humans.
- Keep module interfaces small and stable; isolate heavy dependencies behind partitions or implementation units.

96.10 Rules of thumb: when to use modules vs headers

96.10.1 Prefer named modules when

- you are defining stable library boundaries within your own project,
- you want to reduce compile-time cost from repeatedly parsing large interfaces,
- you want to minimize macro leakage and include-order fragility across interfaces.

96.10.2 Prefer headers when

- you must support older compilers or build systems without module scanning,

- you depend on third-party libraries that are header-based,
- you need simple, universally-supported distribution formats.

96.10.3 A pragmatic migration strategy

- First, enforce strong header hygiene (self-contained headers, no transitive include dependencies).
- Then, convert the most stable, frequently included interfaces into named modules.
- Keep compatibility headers that forward users toward modules where supported.
- Expect a mixed codebase for a long time in real projects.

This strategy aligns with the SF emphasis on robust interfaces and predictable build structure.

96.11 Common pitfalls and how to avoid them

96.11.1 Pitfall: macros in public interfaces

Macros remain special: they are not namespaced, can change meaning across translation units, and can undermine modular boundaries. Prefer typed alternatives (`constexpr`, enums, functions). Keep configuration macros isolated and minimal.

96.11.2 Pitfall: assuming `import` behaves like `#include`

`import` is not textual inclusion. It imports a compiled interface. Many "header tricks" (include order dependencies, macro-based customization, accidental transitive includes) either stop working or behave differently.

96.11.3 Pitfall: build graph not module-aware

If the build system does not scan module dependencies and enforce compilation order, builds can fail unpredictably. Prefer build systems with explicit module support (for example, CMake's module scanning support as documented) and keep module boundaries clear.

96.12 How modules relate back to SF rules

Modules reduce pressure on some SF rules, but do not eliminate the need for disciplined organization:

- **SF.10 and SF.11 remain valuable** in mixed codebases and for headers that become header units.
- **Include guards still matter** for headers that remain headers (and for compatibility layers).
- **Clear interface boundaries remain essential** whether expressed via headers or module interfaces.

Part XVI

Const Correctness (Con)

Chapter 97

Con.1 – Con.4: Const Correctness (Constants and Immutability)

97.1 Overview: what "const correctness" buys you

Const correctness is the disciplined use of `const` (and related techniques) to express and enforce immutability:

- **Better reasoning:** if an object cannot change, you can understand code with fewer hidden possibilities.
- **Clearer interfaces:** function signatures communicate whether a call may modify an argument or the object.
- **Earlier bug detection:** the compiler rejects accidental modification.
- **Concurrency benefits:** an immutable object cannot participate in data races.

These motivations are explicitly emphasized in the Core Guidelines section *Con: Constants*

and immutability. The Standard C++ FAQ also highlights the value of `const` as semantic documentation enforced by the compiler.

97.2 Const rule summary (Con.1–Con.4)

- **Con.1:** By default, make objects immutable.
- **Con.2:** By default, make member functions `const`.
- **Con.3:** By default, pass pointers and references to `const`s.
- **Con.4:** Use `const` to define objects with values that do not change after construction.

97.3 Con.1: By default, make objects immutable

97.3.1 Rule

Make objects `const` by default. Make an object non-`const` only when it must change.

97.3.2 Reason

Immutable objects are easier to reason about and prevent accidental or hard-to-notice changes of value.

97.3.3 Example

```
for (const int i : c) std::cout << i << '\n'; // reading: const

for (int i : c) std::cout << i << '\n';       // bad: suggests i might change
```

The `const` loop variable documents intent: “read-only”.

97.3.4 Important exceptions

97.3.4.1 Returned local by value: avoid blocking moves

The guidelines explicitly warn: a local variable returned by value and cheaper to move than copy should *not* be declared `const`, because it can force an unnecessary copy.

```
std::vector<int> make_vec(int i)
{
    std::vector<int> v{ i, i, i };  // const not needed
    return v;                       // allow move/copy elision as appropriate
}
```

97.3.4.2 Pass-by-value parameters: do not enforce const

Function parameters passed by value are rarely mutated but also rarely declared `const`. To avoid confusion and false positives, the guidelines recommend not enforcing Con.1 for pass-by-value parameters.

```
void g(const int i) { /* ... */ } // pedantic (parameter is a local copy)
```

97.3.5 Enforcement guidance

Flag non-`const` variables that are not modified, except:

- pass-by-value parameters, and
- returned locals where `const` could inhibit move.

97.4 Con.2: By default, make member functions `const`

97.4.1 Rule

A member function should be marked `const` unless it changes the object's *observable state*.

97.4.2 Reason

Marking member functions `const` provides:

- a more precise statement of design intent,
- better readability,
- more errors caught by the compiler,
- and sometimes more optimization opportunities.

97.4.3 Example: a missing `const` breaks legitimate use

```
class Point {
    int x, y;
public:
    int getx() { return x; }   // bad: should be const
};

void f(const Point& pt)
{
    int x = pt.getx();         // error: getx() not callable on const Point
}
```

This is exactly the kind of bug const correctness prevents.

97.4.4 Logical constness, caches, and `mutable`

97.4.4.1 Const is about observable state

The guidelines note that `const` is not transitive: a `const` member function may still modify:

- `mutable` members, and

- objects reachable through pointer members.

This is acceptable only when it preserves the class's promised semantics (invariants), such as maintaining a cache.

97.4.4.2 Example: memoized string representation

```
class Date {
public:
    const std::string& string_ref() const
    {
        if (string_val == "") compute_string_rep();
        return string_val;
    }
private:
    void compute_string_rep() const;   // stores into string_val
    mutable std::string string_val;    // cache
};
```

The object remains logically the same to users, while enabling efficient repeated queries.

97.4.5 Dealing with legacy APIs that are not const-correct

The guidelines acknowledge that some older APIs take `T*` even when they do not modify `*T`. Preferred solutions, in order:

- update the library/API to be const-correct (best long-term),
- avoid casting away const,
- when you cannot change the API (e.g., third-party library), use a wrapper.

```
void legacy_f(int* p); // does not actually modify *p, but signature is wrong
```

```
void f(const int* p)   // wrapper: const-correct interface
{
    legacy_f(const_cast<int*>(p)); // patch only when you cannot fix legacy_f
}
```

The wrapper is a compromise and should not be the default design choice.

97.4.6 Enforcement guidance

Flag a member function that is not marked `const` but does not perform a non-`const` operation on any data member.

97.4.7 Tool support note

Microsoft's C++ Core Guidelines checker introduced rules related to the *Constants and Immutability* section and discusses practical enforcement tradeoffs (such as false positives for certain patterns).

97.5 Con.3: By default, pass pointers and references to `const`s

97.5.1 Rule

When a function should not modify an argument, pass it as `const T&` or `const T*` by default. Use non-`const` only when mutation is intended.

97.5.2 Reason

Passing pointers/references to `const` avoids unexpected modification and makes programs easier to reason about.

97.5.3 Example: the signature communicates intent

```
void f(char* p);        // does f modify *p? assume it does
void g(const char* p);  // g does not modify *p
```

If you see `T*` or `T&`, you must assume the function may modify the object. If it will not, the signature should say so.

97.5.4 Critical note: do not cast away const

The guideline explicitly warns: do not cast away const. If a function receives `const T*` or `const T&`, modifying that object (especially via a cast) breaks the caller's expectations and can be undefined behavior in real programs.

97.5.5 Enforcement guidance

- Flag a function that does not modify an object passed by pointer or reference to non-`const`.
- Flag a function that (using a cast) modifies an object passed by pointer or reference to `const`.

97.6 Con.4: Use `const` to define objects with values that do not change after construction

97.6.1 Rule

If an object's value is not intended to change after initialization, declare it `const`.

97.6.2 Reason

This prevents surprises: if a variable is non-`const`, readers must assume it may change, even if it never does.

97.6.3 Example: remove unnecessary uncertainty

```
void f()
{
    int x = 7;
    const int y = 9;

    for (;;) {
        // ...
    }
}
```

Because `x` is not `const`, a reader must assume it might be modified somewhere in the loop, even if it is not. Declaring truly constant values `const` makes reasoning tighter.

97.6.4 Where Con.4 is most valuable

- **Loop invariants:** values used in loop bodies that should never change.
- **Fixed configuration values:** constants with runtime initialization that should remain stable.
- **Object invariants:** data members that should not change after construction (when the type design supports it).

97.6.5 Enforcement guidance

Flag unmodified non-`const` variables.

97.7 Practical patterns and pitfalls (for Con.1–Con.4)

97.7.1 Prefer const-correct interfaces over const-correct call sites

If a function does not mutate, express that in the function type:

```
void print(const std::string& s);  // communicates non-mutation
```

Call-site constness alone does not protect other callers.

97.7.2 Do not sprinkle const on pass-by-value parameters

```
void bad(const int i);  // does not help callers; i is a local copy
```

This is specifically called out as pedantic and not worth enforcing.

97.7.3 Const correctness is about semantics, not just syntax

A `const` member function can still mutate caches or pointed-to objects. This is not "wrong" by itself, but the class must preserve the promised meaning of `const` for its users (observable state).

97.7.4 Do not use `const_cast` as a design tool

Using `const_cast` to call legacy APIs should be treated as a temporary compatibility technique, not a normal coding style. Prefer fixing the API or adding a small wrapper with a const-correct signature.

97.8 Checklist for code reviews

- Are variables that never change declared `const` (Con.1, Con.4)?

- Are member functions `const` unless they change observable state (Con.2)?
- Are pointer/reference parameters `const` unless mutation is required (Con.3)?
- Are there any `const_cast` uses that can be removed by fixing signatures or adding wrappers?

Chapter 98

`const` objects

98.1 Scope of this chapter

This chapter explains what a `const` object means in ISO C++, how it behaves at different scopes (local, member, namespace scope), how it interacts with initialization, linkage, and the build model, and how to apply it correctly in modern codebases.
The guidance here aligns with the C++ Core Guidelines section *Con: Constants and immutability*, especially Con.1 and Con.4, which encourage immutability by default and using `const` for values that do not change after construction.

98.2 Definition: what is a `const` object?

98.2.1 The language rule (observable effect)

A `const` object is an object whose type is `const`-qualified (or a non-`mutable` subobject of a `const` object). Such an object cannot be modified:

- direct modification attempts are rejected by the compiler,
- indirect modification through a non-`const` pointer/reference is **undefined behavior**.

This is a core property of C++ cv-qualification.

98.2.2 Shallow immutability

In C++, `const` is not automatically transitive through pointers:

- `const T` means the `T` object itself may not be modified.
- If the object contains pointers, `const` does not automatically make the pointees `const`.

```
struct S {
    int  value;
    int* p;        // points elsewhere
};

void demo(const S& s)
{
    // s.value = 1;              // error: cannot modify const object
    *s.p = 42;                   // OK: does not modify s itself
}
```

98.3 Con.1 and Con.4: when to declare objects `const`

98.3.1 Make objects immutable by default

The Core Guidelines recommend making objects immutable by default (Con.1) and using `const` for values that do not change after construction (Con.4). The goal is to make intent explicit and prevent accidental modification.

98.3.2 What `const` promises

Declaring an object `const` communicates and enforces:

- the object will not be reassigned or mutated (except `mutable` subobjects),
- readers can treat the value as stable after initialization,
- APIs should accept it through `const` references/pointers unless mutation is intended.

98.4 Initialization: when can a `const` object be assigned?

98.4.1 Initialization is allowed, later modification is not

A `const` object must be initialized, and after that its value may not be modified.

```
const int a = 10;   // OK: initialized
// a = 20;          // error: cannot assign to const object
```

98.4.2 `const` data members must be initialized in constructors

A `const` data member cannot be assigned in the constructor body; it must be initialized in the member initializer list.

```
struct Config {
    const int timeout_ms;

    explicit Config(int t) : timeout_ms(t) {} // OK
};
```

98.4.3 Use `constexpr` when you need compile-time constants

`const` expresses immutability, but it does not by itself require compile-time evaluation. If you need a compile-time constant, use `constexpr`.

```
constexpr int n = 16;        // compile-time constant expression
const int m = n * 2;         // const, may still be a constant expression in many cases
```

98.4.4 Use `constinit` to prevent dynamic initialization

For variables with static or thread storage duration, `constinit` asserts that the initialization is *static initialization* (zero-initialization and constant initialization). If the variable would require dynamic initialization, the program is ill-formed.

```
struct X { int v; };

constinit X gx{42};        // OK if this is constant initialization
// constinit X gy = makeX(); // ill-formed if it requires dynamic initialization
```

98.5 Const and member functions: what a `const` object can call

98.5.1 A `const` object can call only `const` member functions

A `const` object can only call member functions that are `const`-qualified, which is why Con.2 recommends making member functions `const` by default.

```
class Point {
    int x_{0}, y_{0};
public:
    int x() const noexcept { return x_; }   // const member function
```

```
    void move(int dx, int dy) { x_ += dx; y_ += dy; } // mutator
};

void use(const Point& p)
{
    int v = p.x();   // OK
    // p.move(1,1);  // error: cannot call non-const on const object
}
```

98.5.2 Logical constness and `mutable`

A `const` member function is permitted to modify `mutable` members. This is commonly used for caches and memoization while keeping the logical (observable) state unchanged.

```
#include <string>

class Date {
public:
    const std::string& text() const
    {
        if (cache_.empty()) build_cache();
        return cache_;
    }
private:
    void build_cache() const { cache_ = "2025-12-25"; } // writes cache_
    mutable std::string cache_; // allowed to change in const function
};
```

98.6 Const and pointers: reading the type correctly

98.6.1 Top-level vs low-level const

- `const T*` means: pointer to `const T` (you cannot modify `*p` through `p`).
- `T* const` means: `const` pointer to `T` (you cannot reassign the pointer itself).
- `const T* const` means: both.

```
int x = 1, y = 2;

const int* p1 = &x;   // pointer to const int
// *p1 = 3;           // error
p1 = &y;              // OK

int* const p2 = &x;   // const pointer
*p2 = 3;              // OK
// p2 = &y;           // error
```

98.7 Const at namespace scope: linkage and headers

98.7.1 A common surprise: `const` affects linkage

In C++, a namespace-scope (or global-scope) `const` variable that is not declared `extern` has **internal linkage** by default, meaning each translation unit may have its own copy. This is a key build-structure detail when placing constants in headers.

```
// constants.h
#ifndef MYLIB_CONSTANTS_H
#define MYLIB_CONSTANTS_H

const int k = 7;  // internal linkage by default at namespace scope

#endif
```

Including this header in multiple translation units typically does not create a multiple-definition linker error because each translation unit gets its own internal-linkage entity, but it can still be undesirable if you expected a single shared object with one address.

98.7.2 Prefer C++17 inline variables for shared header-defined constants

If you want a single program-wide object defined in a header, use an `inline` variable (C++17 and later). Inline variables are designed to allow the same definition to appear in multiple translation units without violating the One Definition Rule, provided the definitions are identical.

```
// constants.h
#ifndef MYLIB_CONSTANTS_H
#define MYLIB_CONSTANTS_H

inline constexpr int max_items = 1024; // safe header definition (C++17+)

#endif
```

98.7.3 ODR reminder

Across the whole program:

- non-inline variables/functions that are odr-used must have exactly one definition,
- inline variables/functions may have definitions in multiple translation units, but they must be identical.

98.8 Const does not guarantee “compile-time”

98.8.1 Const vs constant expression

A `const` object is immutable after initialization, but whether it can be used in contexts requiring a constant expression depends on the rules for constant expressions.

```
#include <array>

int f();

const int a = 3;          // may be a constant expression when initialized by a constant
↪  expression
// const int b = f();     // not a constant expression

std::array<int, a> ok{};  // OK if a is a constant expression
// std::array<int, b> bad{}; // error: b is not a constant expression
```

98.9 Undefined behavior: “casting away const” and writing anyway

98.9.1 Why it is dangerous

Attempting to modify a truly `const`-qualified object through a pointer/reference to non-`const` is undefined behavior. The compiler is allowed to place `const` objects in read-only storage or assume they never change.

```
void bad(const int& r)
{
    int& x = const_cast<int&>(r);
    x = 99; // undefined behavior if r refers to a const object
}
```

98.9.2 Acceptable use: fixing a non-const legacy API via a wrapper

Sometimes third-party APIs are not const-correct. The Core Guidelines recommend fixing the API when possible; otherwise, isolate the workaround in a wrapper rather than spreading casts throughout the codebase.

98.10 Const and concurrency

Immutability helps in concurrent code because an object that never changes cannot be involved in a data race via modification. The Core Guidelines explicitly emphasize immutability as a tool for safer reasoning.

98.11 Practical guidance: when and how to use `const` objects

98.11.1 Prefer `const` for true invariants

Good uses:

- configuration values that must not change after initialization,
- loop invariants and helper values that should remain stable,
- references/pointers used only for reading.

98.11.2 Avoid meaningless `const` on pass-by-value parameters

The Core Guidelines note that function arguments passed by value are local copies; declaring them `const` is often pedantic and usually not enforced.

98.11.3 Be careful with returned locals and move

For returned local objects, unnecessary `const` can inhibit moves in some patterns. The guidelines call this out as an exception to “const by default.”

98.12 Checklist for reviews and refactoring

- Is every value that never changes declared `const` (or `constexpr` if it should be compile-time)?
- Are namespace-scope constants in headers expressed as `inline constexpr` (C++17+) rather than as plain `const` objects when a single shared object is intended?
- Are member functions correctly marked `const` when they do not change observable state?
- Are there any `const_cast` writes that could become undefined behavior?
- For static/thread storage variables that must not have dynamic initialization, is `constinit` used?

Chapter 99

`const` member functions

99.1 Purpose and definition

A `const` member function is a non-static member function declared with a `const` qualifier after its parameter list:

```
class X {
public:
    int size() const;   // const member function
    void clear();       // non-const member function
};
```

The meaning is simple and very important:

- Inside a `const` member function, the implicit object parameter (`*this`) is treated as **const**.
- Therefore, you cannot modify non-`mutable` data members, and you cannot call non-`const` member functions on `*this`.

- A `const` object (or a reference/pointer to const) can call only `const` member functions.

This is the foundation of *const correctness* for class interfaces.

99.2 Core Guideline Con.2: default to `const`

99.2.1 Rule

Mark member functions `const` unless they change the object's *observable state*.

99.2.2 Reason

Declaring a member function `const`:

- documents intent in the type system,
- prevents accidental mutation,
- enables use of the type through `const` references/pointers,
- often improves usability because read-only operations become widely callable.

99.2.3 Example: missing `const` breaks normal usage

```
class Point {
    int x_{0}, y_{0};
public:
    int x() { return x_; }              // BAD: should be const
    int y() const { return y_; }        // OK
};

void f(const Point& p)
```

```
{
    int a = p.x(); // error: x() is not const
    int b = p.y(); // OK
}
```

99.3 How the language models `const` member functions

99.3.1 The implicit object parameter (`this`)

For an ordinary non-static member function, the call:

```
obj.size();
```

is conceptually a function call with an implicit object argument. The `const` qualifier changes the cv-qualification of that implicit object parameter.
A practical mental model:

- `int size();` behaves like a function taking `X*` as the implicit object parameter.
- `int size() const;` behaves like a function taking `const X*` as the implicit object parameter.

99.3.2 Overloading on constness

Member functions may overload on cv-qualification:

```
class Buffer {
public:
    int& at(std::size_t i) { return data_[i]; }             // mutable access
    const int& at(std::size_t i) const { return data_[i]; } // read-only access
private:
    int data_[8]{};
};
```

Overload resolution selects the non-`const` overload for non-`const` objects, and the `const` overload for `const` objects:

```
void demo()
{
    Buffer b;
    const Buffer cb{};

    b.at(0) = 7;     // calls non-const at()
    // cb.at(0) = 7; // error: returns const int&, calls const at()
}
```

99.4 Ref-qualifiers with `const` member functions

Beyond `const`, modern C++ also allows ref-qualifiers on member functions: `&` and `&&`. These specify whether the function may be called on lvalues, rvalues, or both.

99.4.1 Why this matters

Ref-qualifiers help prevent dangerous APIs such as returning references to subobjects of a temporary.

99.4.2 Example: safe accessors

```
#include <string>
#include <utility>

class Name {
    std::string s_;
public:
    explicit Name(std::string s) : s_(std::move(s)) {}
```

```
    // Lvalue object: return reference
    const std::string& get() const & { return s_; }

    // Rvalue object (temporary): return by value
    std::string get() && { return std::move(s_); }
};
```

```
Name make();

void use()
{
    Name n("abc");
    const std::string& r = n.get(); // OK: const& overload

    std::string v = make().get();   // OK: && overload, avoids dangling reference
}
```

99.5 What `const` allows and what it does not

99.5.1 Const is not deep immutability

A `const` member function cannot modify non-`mutable` data members, but it *can* still modify:

- objects reachable through pointers/references stored in the object,
- `mutable` members (see next section).

```
struct S {
    int  value{0};
    int* p{nullptr};
    void read_only() const
    {
```

```
        // value = 1;   // error
        if (p) *p = 9;  // OK: modifies *p, not this->value
    }
};
```

99.5.2 Const should match semantics (observable state)

The important design rule is: `const` should mean the operation does not change what users can observe through the public interface (except for allowed performance-related effects like caches).

99.6 Logical constness and `mutable`

99.6.1 Motivation

Sometimes you want a read-only operation to compute and store a cache, or to record statistics, without changing the object's logical value. C++ supports this via `mutable` data members.

99.6.2 Example: caching in a const operation

```
#include <string>
#include <utility>

class Document {
public:
    explicit Document(std::string text) : text_(std::move(text)) {}

    std::size_t word_count() const
    {
        if (!cache_valid_) {
            cache_ = compute_word_count();
```

```
            cache_valid_ = true;
        }
        return cache_;
    }

private:
    std::size_t compute_word_count() const
    {
        // naive count for illustration
        std::size_t n = 0;
        bool in_word = false;
        for (char c : text_) {
            const bool space = (c == ' ' || c == '\n' || c == '\t');
            if (!space && !in_word) { ++n; in_word = true; }
            if (space) in_word = false;
        }
        return n;
    }

private:
    std::string text_;
    mutable std::size_t cache_{0};
    mutable bool cache_valid_{false};
};
```

99.6.3 Guidance

- Use `mutable` for *performance caches* or *non-semantic bookkeeping* only.
- Do not use `mutable` to bypass a poor class design. If a function truly mutates the object's meaning, it should not be `const`.

99.7 Const member functions and thread safety

99.7.1 Const does not imply thread-safe

A `const` member function may still write to `mutable` members, and multiple threads calling such a function concurrently can create data races unless synchronization is used.

99.7.2 Example: cache with synchronization

```
#include <mutex>
#include <string>
#include <utility>

class SafeDocument {
public:
    explicit SafeDocument(std::string text) : text_(std::move(text)) {}

    std::size_t word_count() const
    {
        std::lock_guard<std::mutex> lock(m_);
        if (!cache_valid_) {
            cache_ = compute_word_count_unlocked();
            cache_valid_ = true;
        }
        return cache_;
    }

private:
    std::size_t compute_word_count_unlocked() const
    {
        std::size_t n = 0;
        bool in_word = false;
```

```
        for (char c : text_) {
            const bool space = (c == ' ' || c == '\n' || c == '\t');
            if (!space && !in_word) { ++n; in_word = true; }
            if (space) in_word = false;
        }
        return n;
    }

private:
    std::string text_;
    mutable std::mutex m_;
    mutable std::size_t cache_{0};
    mutable bool cache_valid_{false};
};
```

This preserves a const interface while preventing data races in the cache.

99.8 Designing APIs with const and non-const overloads

99.8.1 Pattern 1: read-only vs mutable accessors

Provide both:

- a `const` overload returning a read-only view,
- a non-`const` overload returning a mutable reference or proxy.

```
#include <string>

class Person {
    std::string name_;
public:
    const std::string& name() const noexcept { return name_; }
```

```
    std::string& name() noexcept { return name_; }
};
```

99.8.2 Pattern 2: avoid duplication with one implementation

A common technique is to implement the const overload and forward from the non-const overload, or vice versa, carefully:

```
class Vec {
    int data_[4]{};
public:
    const int& at(std::size_t i) const { return data_[i]; }
    int& at(std::size_t i) { return const_cast<int&>(static_cast<const Vec&>(*this).at(i));
    ↪  }
};
```

This avoids duplicated bounds checks, but use it only when you are sure the const overload is correct and the non-const one simply relaxes mutability. The cast must never be used to modify a truly const object.

99.8.3 Better alternative (when possible)

If you can centralize logic without casts, prefer that, for example by putting shared checks into a private helper.

99.9 Interaction with `constexpr`, `noexcept`, and `[[nodiscard]]`

99.9.1 `const` composes with other function qualifiers

Const member functions frequently pair well with:

- `noexcept` for simple observers,
- `constexpr` when the function can be evaluated at compile time,
- `[[nodiscard]]` for values that should not be ignored.

```
class Counter {
    int n_{0};
public:
    [[nodiscard]] constexpr int value() const noexcept { return n_; }
    void inc() noexcept { ++n_; }
};
```

99.10 Modern note: C++23 “deducing `this`” (explicit object parameter)

C++23 allows writing a member-like function that takes an explicit object parameter using the `this` introducer. This can reduce the need for manually writing multiple overloads (e.g., `const&` vs `&&`) by using templates and normal deduction.

99.10.1 Example: one definition handles multiple cv/ref cases

```
#include <string>
#include <utility>

struct Box {
    std::string s;

    template<class Self>
    decltype(auto) get(this Self&& self)
    {
```

```
        return std::forward<Self>(self).s;
    }
};
```

```
void demo()
{
    Box b{"abc"};
    const Box cb{"xyz"};

    b.get() = "changed";     // mutable lvalue: returns std::string&
    // cb.get() = "no";      // const lvalue: returns const std::string&
    std::string moved = Box{"tmp"}.get(); // rvalue: returns std::string&& (moves)
}
```

This is an advanced technique; it is most useful when you would otherwise need many cv/ref-qualified overloads.

99.11 Common mistakes and how to avoid them

99.11.1 Mistake 1: marking a logically mutating function as const

If an operation changes what users observe (not just a cache), it should not be `const`. Otherwise, const becomes misleading.

99.11.2 Mistake 2: forgetting const on observers

If a function is an observer (query), it should almost always be `const` and often `noexcept`.

99.11.3 Mistake 3: returning non-const references from const member functions

A `const` member function returning `T&` usually breaks const correctness, because it allows modification through a const object.

```
class Bad {
    int x_{0};
public:
    int& x() const { return x_; } // BAD: allows mutation through const object
};
```

99.11.4 Mistake 4: using `const_cast` as a design tool

Casting away const to mutate data violates the caller's contract and can be undefined behavior if the original object is truly const. Use wrappers only for unavoidable legacy APIs, and keep the cast isolated.

99.12 Review checklist

- Are observer/query member functions marked `const` (and typically `noexcept`)?
- Do const functions preserve the object's observable state (logical constness)?
- If a const function writes to caches/statistics, is it clearly `mutable`, and is it thread-safe when needed?
- Are const and non-const overloads provided for accessors that return references?
- Are ref-qualifiers (`&`, `&&`) used when returning references that could dangle from temporaries?

- Is C++23 explicit-object-parameter used only when it truly simplifies multi-overload designs?

Chapter 100

Logical vs. bitwise const

100.1 Why this distinction exists

In real C++ code, `const` has two different (and sometimes conflicting) interpretations:

- **Bitwise (physical) constness:** the object's memory representation does not change.
- **Logical constness:** the object's *observable meaning* does not change, even if some internal bookkeeping bits change.

The ISO C++ community guidance is that `const` is primarily a *logical* promise: a `const` operation should not change what a caller can observe through the type's public interface. This is the model used in the ISO C++ FAQ and widely taught in modern C++ guidance.

100.2 Definitions

100.2.1 Bitwise (physical) constness

An operation is **bitwise const** if it performs no writes to the object's subobjects (no changes to the object's stored bits), except possibly to memory that is *not part of the object*.
Bitwise constness is easy to understand and easy to test, but it can be too strict:

- it forbids internal caches,
- it forbids internal counters and instrumentation,
- it forbids internal locking state needed for thread safety.

100.2.2 Logical constness

An operation is **logically const** if it does not change the object's *externally visible state* (its meaning to clients), even if it updates internal representation details (cache, statistics, locks) that are not considered part of that meaning.
The ISO C++ FAQ describes this as the intention behind `mutable`: it marks parts of the object's *physical state* that are not part of the object's *logical state*. Herb Sutter's GotW #6 similarly emphasizes that `const` is about the object's observable state, not necessarily every internal bit.

100.3 What the language enforces vs. what you promise

100.3.1 What the compiler enforces inside a `const` member function

Inside a `const` member function:

- `*this` is treated as `const`,
- you cannot modify non-`mutable` data members,
- you cannot call non-`const` member functions on `*this`.

However, C++ deliberately allows two important escape hatches:

- `mutable` members may be modified in `const` member functions,
- objects reachable through pointers/references stored in the object may be modified (because `const` is not deep/transitive).

The cv-qualification model and the role of `mutable` are summarized clearly in cppreference.

100.3.2 What `const` should mean in design

The Core Guidelines' const-correctness rules (Con.2 and related guidance) are best satisfied when `const` member functions preserve *logical* state. That is: a `const` member function should be callable on a `const` object without surprising the caller by changing the object's meaning.

100.4 Examples: bitwise const vs. logical const

100.4.1 Example 1: bitwise-const observer

```
class Point {
    int x_{0}, y_{0};
public:
    int x() const noexcept { return x_; } // bitwise const
    int y() const noexcept { return y_; } // bitwise const
};
```

These observers perform no writes to `*this`. They are both bitwise and logically const.

100.4.2 Example 2: logically const cache

A cache changes internal bits, but (if correctly designed) does not change the observable meaning.

```
#include <string>
#include <utility>

class Date {
public:
    explicit Date(int y, int m, int d) : y_(y), m_(m), d_(d) {}

    const std::string& text() const
    {
        if (!cache_valid_) {
            cache_ = build_text();
            cache_valid_ = true;
        }
        return cache_;
    }

private:
    std::string build_text() const
    {
        return std::to_string(y_) + "-" + std::to_string(m_) + "-" + std::to_string(d_);
    }

private:
    int y_, m_, d_;
    mutable std::string cache_;
    mutable bool cache_valid_{false};
};
```

This is **not bitwise const** (it writes `cache_` and `cache_valid_`), but it **can be logically const** if the cache does not affect the value that callers observe. The ISO C++ FAQ explicitly uses caching as a classic reason for `mutable`.

100.5 The role of `mutable`

100.5.1 What `mutable` means

Declaring a non-static data member `mutable` permits modifying that member even when the containing object is `const` (including from a `const` member function). This is documented in both Microsoft's C++ reference and cppreference.

100.5.2 What `mutable` is for (logical state separation)

`mutable` is intended for data that is part of the object's **physical** state but not part of its **logical** state:

- memoization caches,
- instrumentation counters,
- lazily-computed representations,
- synchronization primitives used to keep the logical state consistent.

This exact interpretation is stated in the ISO C++ FAQ and echoed in cppreference's description (caches, mutexes, instrumentation).

100.6 Logical constness and concurrency

100.6.1 Critical rule: logically-const writes must be safe under concurrency

A major pitfall is believing that "`const` means thread-safe." It does not.
If a `const` member function performs internal writes (for example, updating a cache), then concurrent calls can create a data race unless the internal state is properly synchronized.

cppreference explicitly notes the common pairing of `mutable` with `std::mutex` for this reason (often summarized as the “M&M rule”: *mutable and mutex go together*).

100.6.2 Example: logically const with synchronization

```
#include <mutex>
#include <string>
#include <utility>

class SafeDate {
public:
    explicit SafeDate(int y, int m, int d) : y_(y), m_(m), d_(d) {}

    const std::string& text() const
    {
        std::lock_guard<std::mutex> lock(m_);
        if (!cache_valid_) {
            cache_ = build_text_unlocked();
            cache_valid_ = true;
        }
        return cache_;
    }

private:
    std::string build_text_unlocked() const
    {
        return std::to_string(y_) + "-" + std::to_string(m_) + "-" + std::to_string(d_);
    }

private:
    int y_, m_, d_;
    mutable std::mutex m_;
    mutable std::string cache_;
```

```
    mutable bool cache_valid_{false};
};
```

This design keeps a `const` interface while making the internal writes safe.
Herb Sutter's GotW discussion emphasizes that if a `const` function performs internal writes, they should be properly synchronized so concurrent callers cannot observe inconsistent behavior.

100.7 When logical constness becomes a bug

Logical constness is a *design contract*. You can break it even while staying within the compiler's rules.

100.7.1 Bug pattern 1: changing semantics via a `mutable` member

```
class Account {
public:
    int balance() const noexcept { return balance_; }

    // BAD: this changes the observable meaning of the object
    void apply_fee() const { balance_ -= 10; }

private:
    mutable int balance_{100}; // misuse: balance is clearly logical state
};
```

This compiles, but violates the meaning of `const`. `mutable` should not be used to bypass design intent.

100.7.2 Bug pattern 2: transitive writes through pointers

Because `const` is not deep, a `const` member function can modify pointed-to data:

```
struct S {
    int* p;
    void f() const { *p = 1; } // allowed by the type system
};
```

Whether this is logically const depends on your class contract. If `*p` is part of the object's logical state (from the user's perspective), then this violates logical constness even though it compiles.

100.8 Practical guidance for robust const-correct design

100.8.1 Prefer bitwise constness for simple observers

If an observer can naturally be bitwise const, make it bitwise const. This gives:

- simplest reasoning,
- easiest testability,
- best default thread-safety properties (no internal writes).

100.8.2 Use logical constness deliberately (and document it)

Use `mutable` only when:

- the modified data is not part of the object's logical state,
- the modification is invisible in the object's public semantics,
- concurrent access is safe (mutex/atomics) if the type is used across threads.

This matches the ISO C++ FAQ's intended use of `mutable` and cppreference's common patterns.

100.8.3 Checklist for code reviews

For every `const` member function that performs writes:

- Is the modified member truly non-semantic (cache, stats, lock)?
- Can a caller observe any difference besides performance?
- If used concurrently, is the mutation synchronized (mutex/atomic)?
- Could the same result be achieved by computing on the fly without caching (simpler)?

Part XVII

Type Safety (Type)

Chapter 101

Type.1 – Type.7: Type Safety

101.1 Overview: what "type safety" means in the Core Guidelines

In the C++ Core Guidelines, **type safety** means that a program uses objects according to the rules of their types: you do not accidentally treat memory that holds an object of one type as if it were an unrelated type, and you avoid conversions that silently change meaning (such as narrowing). The Type Safety rules focus on the main sources of type violations in real C++: **unsafe casts**, **use of uninitialized values**, and **raw unions**.

101.1.1 Rule summary (Type.1–Type.7)

- **Type.1:** Avoid casts (explicit type conversion).
- **Type.2:** Don't use `static_cast` to downcast; use `dynamic_cast` instead.
- **Type.3:** Don't use `const_cast` to cast away `const`.

- **Type.4:** Don't use C-style or functional casts; prefer construction or named casts.
- **Type.5:** Don't use a variable before it has been initialized.
- **Type.6:** Always initialize a data member.
- **Type.7:** Avoid naked `union`; use `std::variant` instead.

101.2 Type.1: Avoid casts

101.2.1 Rule

Avoid explicit casts. If you must convert, prefer safer, intention-revealing alternatives (construction with braces, well-named helpers, or the appropriate named cast).

101.2.2 Rationale

Casts are a major escape hatch from the type system. They can:

- conceal narrowing conversions (loss of information),
- conceal reinterpretation of representation (type punning),
- make code fragile under maintenance (a later type change can silently introduce bugs),
- bypass compiler diagnostics that would otherwise prevent misuse.

The Core Guidelines emphasize avoiding casts as a primary way to preserve static type safety.

101.2.3 Prefer construction (especially `T{e}`) for value conversion

Brace construction is explicit and rejects many narrowing conversions.

```
void f(double d)
{
    int a{d};   // error if narrowing would occur (preferred: catches bugs early)
    int b = d;  // bad: narrowing allowed
}
```

101.2.4 When narrowing is intended: use GSL narrowing helpers

The Guidelines Support Library (GSL) provides two explicit narrowing tools:

- `gsl::narrow_cast<T>(x)`: perform the narrowing because you requested it.
- `gsl::narrow<T>(x)`: checks and throws if the narrowing would lose information.

```
#include <gsl/gsl>
#include <cmath>

int to_int_checked(double d)
{
    return gsl::narrow<int>(d);       // throws gsl::narrowing_error if lossy
}

int to_int_explicit(double d)
{
    return gsl::narrow_cast<int>(d);  // explicit: you accept potential loss
}
```

101.2.5 Type.1 strict sub-rules (what the profile bans)

The Core Guidelines type-safety profile tightens Type.1 by flagging:

- use of `reinterpret_cast`,
- use of `static_cast` for arithmetic conversions (prefer braces or GSL narrow),

- unnecessary pointer-to-pointer casts (same source/target type),
- pointer casts that could be done implicitly (suggests the cast is redundant or suspicious).

101.2.6 Safer alternatives to common cast motives

- **Polymorphic downcast:** `dynamic_cast` (Type.2).
- **Narrowing numeric conversion:** braces or `gsl::narrow`.
- **Bit-level reinterpretation:** prefer well-defined library techniques (e.g., copying bytes with `std::memcpy`); avoid `reinterpret_cast` in most code.

101.3 Type.2: Don't use `static_cast` to downcast; use `dynamic_cast`

101.3.1 Rule

Avoid `static_cast` for downcasts in class hierarchies. When hierarchy navigation is unavoidable, use `dynamic_cast` instead.

101.3.2 Rationale

A downcast using `static_cast` does not perform a runtime check. If the dynamic type of the object is not the target derived type, the program can:

- interpret an object as the wrong type,
- violate invariants,
- produce undefined behavior.

Static analysis tools treat this as a high-value diagnostic for type safety.

101.3.3 Example: unsafe `static_cast` vs checked `dynamic_cast`

```
#include <iostream>

struct Base { virtual ~Base() = default; };
struct D1 : Base { void f() const { std::cout << "D1\n"; } };
struct D2 : Base { void g() const { std::cout << "D2\n"; } };

void use(Base& b)
{
    // BAD: if b is actually a D2, this is type confusion
    auto& d1 = static_cast<D1&>(b);
    d1.f(); // may be undefined behavior

    // GOOD: checked cast (returns nullptr for pointers, throws for references)
    if (auto* p = dynamic_cast<D1*>(&b)) {
        p->f();
    }
}
```

101.3.4 Design note: prefer avoiding downcasts entirely

Frequent downcasting usually indicates a design issue (missing virtual function, visitor, or variant-based modeling). Type.2 is a safety rule; it does not endorse downcasting as a normal design technique.

101.4 Type.3: Don't use `const_cast` to cast away `const`

101.4.1 Rule

Do not use `const_cast` to remove `const` (effectively: don't cast away `const` at all).

101.4.2 Rationale

Casting away `const` breaks the contract expressed by types and often leads to undefined behavior when you attempt to modify an object that was originally `const`-qualified. It also destroys the reasoning benefits of const-correct interfaces.

101.4.3 Example: undefined behavior risk

```
void bad(const int& r)
{
    int& x = const_cast<int&>(r);
    x = 42; // undefined behavior if r refers to an actually-const object
}
```

101.4.4 Practical guidance

If you feel you need `const_cast`:

- fix the API to be const-correct (best),
- or add a small wrapper that isolates legacy behavior,
- or redesign so that mutation is explicit and honest in the type system.

101.5 Type.4: Don't use C-style or functional casts

101.5.1 Rule

Avoid:

- C-style casts: `(T)expr`

- functional casts: `T(expr)`

Prefer construction (`T{expr}` or `T(expr)` where it is unambiguously construction), or a named cast (`static_cast`, `dynamic_cast`, `reinterpret_cast`, `const_cast`) when you truly need that specific meaning.

101.5.2 Rationale

C-style casts can perform several different conversions, including dangerous ones, without clearly signaling which one you intended. Named casts force you to be explicit about intent and make code review and tooling more effective.

101.5.3 Example

```
struct B { virtual ~B() = default; };
struct D : B {};

void f(B* pb)
{
    D* p1 = (D*)pb;                // BAD: ambiguous, unchecked
    D* p2 = dynamic_cast<D*>(pb); // GOOD: checked downcast
}
```

101.6 Type.5: Don’t use a variable before it has been initialized

101.6.1 Rule

Never read from an uninitialized object. Always initialize objects.

101.6.2 Rationale

Using an uninitialized value is a direct path to undefined behavior, security bugs, and irreproducible failures. The rule is especially important for fundamental types with automatic storage duration, where default initialization does not produce a meaningful value.

101.6.3 Example

```
#include <iostream>

void g()
{
    int x;                // uninitialized
    // std::cout << x;     // BAD: reading uninitialized value
    int y{0};             // GOOD: initialized
    std::cout << y;
}
```

101.6.4 Recommended initialization style

Prefer brace initialization because it is explicit and consistent:

```
int a{};          // value-initialized (0)
double d{};       // 0.0
std::string s{};  // empty string
```

101.6.5 Tooling

Microsoft's Core Guidelines checkers explicitly diagnose missing initialization as a Type.5 violation.

101.7 Type.6: Always initialize a data member

101.7.1 Rule

Every data member should be initialized (either with a default member initializer or in every constructor).

101.7.2 Rationale

Uninitialized members create objects that are invalid immediately after construction and lead to fragile invariants. This is a frequent source of production defects, especially when new members are added later and not wired into every constructor.

101.7.3 Preferred pattern: default member initializers

Default member initializers scale well as the class evolves:

```
#include <string>

class Config {
public:
    Config() = default;
    explicit Config(int port) : port_(port) {}

    int port() const noexcept { return port_; }
    const std::string& host() const noexcept { return host_; }

private:
    int port_{8080};              // always initialized
    std::string host_{"localhost"}; // always initialized
};
```

101.7.4 When constructor initialization is better

For members that must depend on constructor parameters, initialize them in the member initializer list.

```
class Range {
public:
    Range(int lo, int hi) : lo_(lo), hi_(hi) {}
private:
    int lo_;
    int hi_;
};
```

101.8 Type.7: Avoid naked `union`; use `std::variant` instead

101.8.1 Rule

Avoid raw (“naked”) unions as general-purpose sum types. Prefer a type-safe alternative, especially `std::variant`.

101.8.2 Rationale

A raw union does not track which member is active. Reading from an inactive member is undefined behavior in common cases, and maintaining the discriminator manually is error-prone. `std::variant` provides:

- a tracked active alternative,
- safe access (`std::get`, `std::get_if`),
- structured visitation (`std::visit`).

101.8.3 Example: from union to variant

```
#include <variant>
#include <string>
#include <iostream>

using Value = std::variant<int, double, std::string>;

void print(const Value& v)
{
    std::visit([](const auto& x) { std::cout << x << '\n'; }, v);
}

void demo()
{
    Value v = 42;
    print(v);

    v = std::string{"hello"};
    print(v);
}
```

101.8.4 When a union can still be appropriate

Unions are sometimes used for low-level tasks with carefully managed active members and strict rules (e.g., certain ABI or hardware register overlays). Type.7 is a safety rule: it warns that **the default choice for sum types in normal application code should be** `std::variant`, not raw unions.

101.9 Putting it together: a practical Type Safety checklist

- **Casts:** Can you remove the cast by changing the interface, using construction, or using a safer library facility?
- **Downcasts:** If you truly must navigate a hierarchy, use `dynamic_cast` (or redesign).
- **Const:** Never remove `const` to mutate. Make mutation explicit.
- **Initialization:** Initialize every object and every member. Prefer default member initializers.
- **Sum types:** Prefer `std::variant` over `union`.

Chapter 102

Type Safety

102.1 What “type safety” means in modern C++

In the C++ Core Guidelines, **type safety** means using values according to their declared types so that:

- objects are not accessed through the wrong type,
- conversions do not silently change meaning (especially by narrowing),
- operations are applied only to valid, initialized objects,
- “sum types” are represented with a tracked active alternative (not a raw union without a discriminator).

The Guidelines group these concerns into the *Type Safety* rules (Type.#) and the *Type-safety profile* intended for stronger checking and tool enforcement.

102.2 Why type safety matters (engineering, not style)

Type-safety violations are a major cause of:

- undefined behavior (mis-typed access, uninitialized reads),
- security vulnerabilities (type confusion, memory corruption),
- fragile code that breaks when a type changes (casts hiding intent).

Type safety improves maintainability because types become reliable contracts enforced by the compiler and by static analysis tools.

102.3 The Core Guidelines view: main sources of type-unsafe code

The Core Guidelines concentrate on practical, high-impact issues:

- **Casts** that bypass the type system (Type.1–Type.4),
- **Initialization failures** (Type.5–Type.6),
- **Unions** used as unchecked variants (Type.7).

This chapter explains these themes and provides concrete, modern alternatives.

102.4 Casts: the primary escape hatch (Type.1–Type.4)

102.4.1 Principle: avoid casts when a clearer alternative exists

A cast can hide which conversion is happening and why. Prefer alternatives that express intent in the type system:

- construction (especially brace initialization),
- overload sets with correct parameter types,
- small well-named helper functions,
- standard library facilities that preserve defined behavior.

This matches the Core Guidelines direction in Type.1 and Type.4.

102.4.2 Numeric conversions and narrowing

102.4.2.1 Prefer brace initialization to detect narrowing

Brace initialization rejects many narrowing conversions.

```
void demo(double d)
{
    int a{d};   // error if narrowing would occur (preferred)
    int b = d;  // allowed: may silently narrow
}
```

102.4.2.2 When narrowing is intentional, make it explicit

When you intentionally accept narrowing, do it explicitly and consistently. The Guidelines Support Library (GSL) provides `gsl::narrow` (checked) and `gsl::narrow_cast` (explicit) and Microsoft analysis warnings recommend these patterns for Type.1-related narrowing diagnostics.

```
#include <gsl/gsl>

int to_int_checked(double d)
{
    return gsl::narrow<int>(d);       // throws gsl::narrowing_error if lossy
```

```
}

int to_int_explicit(double d)
{
    return gsl::narrow_cast<int>(d);  // explicit: you accept potential loss
}
```

102.4.3 Downcasting in class hierarchies (Type.2)

102.4.3.1 Why `static_cast` downcasts are dangerous

A `static_cast` downcast does not check the dynamic type. If the object is not actually of the derived type, behavior is undefined.

102.4.3.2 Prefer designs that avoid downcasts

If you often downcast, consider:

- adding a virtual function in the base,
- using a visitor,
- modeling alternatives with `std::variant` rather than a hierarchy.

102.4.3.3 When downcasting is truly needed: use `dynamic_cast`

`dynamic_cast` provides runtime checking *for polymorphic types* (a base with at least one virtual function). If a cast fails:

- pointer casts produce `nullptr`,
- reference casts throw `std::bad_cast`.

Clang-Tidy provides a direct check that enforces the Core Guidelines Type.2 rule.

```
#include <iostream>

struct Base { virtual ~Base() = default; };
struct D1 : Base { void f() const { std::cout << "D1\n"; } };
struct D2 : Base { void g() const { std::cout << "D2\n"; } };

void use(Base& b)
{
    if (auto* p = dynamic_cast<D1*>(&b)) {
        p->f();
    } else {
        // handle other types safely
    }
}
```

102.4.4 Casting away const (Type.3)

102.4.4.1 Why it breaks contracts

If an interface receives `const T&` or `const T*`, it promises not to modify the object. Casting away const to write breaks that promise and is often undefined behavior if the original object is truly const-qualified.

```
void bad(const int& r)
{
    int& x = const_cast<int&>(r);
    x = 42; // undefined behavior if r refers to a const object
}
```

102.4.4.2 Correct fix: make the API const-correct

If a function needs mutation, let the type system express it. If a legacy API is not const-correct, isolate the workaround in a wrapper and prefer fixing the API when possible.

102.4.5 C-style and functional casts (Type.4)

102.4.5.1 Why to avoid them

C-style casts can perform a `static_cast`, `const_cast`, or `reinterpret_cast` in one syntax, making code review and tooling less effective. Clang-Tidy provides a specific check to flag these cases.

```
struct B { virtual ~B() = default; };
struct D : B {};

void f(B* pb)
{
    D* p1 = (D*)pb;                // bad: ambiguous, unchecked
    D* p2 = dynamic_cast<D*>(pb); // good: checked
}
```

102.5 Initialization: prevent “use before set” (Type.5–Type.6)

102.5.1 Type.5: never read an uninitialized object

Reading an uninitialized object is undefined behavior for fundamental types and can cause unpredictable failures.

```
#include <iostream>

void g()
{
    int x;            // uninitialized
    // std::cout << x; // bad: read before initialization

    int y{};          // value-initialized to 0
```

```
    std::cout << y;
}
```

102.5.2 Prefer value initialization and invariants

Idiomatic modern C++ uses:

- `T{}` to value-initialize,
- constructors that establish class invariants,
- RAII types that cannot exist uninitialized in a meaningful “half-state”.

102.5.3 Type.6: always initialize every data member

For classes, ensure every data member is initialized:

- by a default member initializer, or
- in every constructor member initializer list.

This is important for correctness today and for maintainability when new members are added later. Microsoft documents this directly via warning C26495, mapped to Type.6.

```
#include <string>
#include <utility>

class Config {
public:
    Config() = default;
    explicit Config(int port) : port_(port) {}

    int port() const noexcept { return port_; }
    const std::string& host() const noexcept { return host_; }
```

```
private:
    int port_{8080};                 // always initialized
    std::string host_{"localhost"};  // always initialized
};
```

102.5.4 Tooling support

Clang-Tidy has a dedicated check that flags user-provided constructors that fail to initialize fields that would otherwise remain indeterminate.

102.6 Avoid raw unions; use `std::variant` (Type.7)

102.6.1 Why a naked union is unsafe for sum types

A raw union does not track which member is active. If you read a member that is not active, you can trigger undefined behavior and violate invariants. Maintaining a separate discriminator manually is error-prone.

102.6.2 Use `std::variant` for type-safe alternatives

`std::variant` is a discriminated union type that tracks the active alternative and offers safe access with `std::get`, `std::get_if`, and `std::visit`.

```
#include <variant>
#include <string>
#include <iostream>

using Value = std::variant<int, double, std::string>;

void print(const Value& v)
```

```
{
    std::visit([](const auto& x) { std::cout << x << '\n'; }, v);
}
```

102.6.3 Low-level note

Unions can be appropriate in tightly controlled low-level code with well-defined active-member management and strict invariants, but for general application modeling they are a poor default. Type.7 is about choosing the safe default.

102.7 Representation reinterpretation: prefer defined behavior

Many type-safety problems come from treating bytes as a different type. Prefer facilities with defined behavior:

- `std::bit_cast` (C++20) for safe bitwise copying between trivially copyable types of the same size,
- `std::memcpy` for byte-wise transfers,
- serialization formats rather than type punning.

Avoid `reinterpret_cast` except in well-justified, carefully reviewed low-level code.

```
#include <bit>
#include <cstdint>

float bits_to_float(std::uint32_t u) noexcept
{
    return std::bit_cast<float>(u); // defined behavior if sizes match
}
```

102.8 Type-safety profiles and modern enforcement

The Core Guidelines include a *Type-safety profile* that strengthens these rules so they can be enforced by tools. In parallel, WG21 papers propose standard "profiles" (including type safety) as a possible direction for future C++ safety enforcement, focusing on deterministically decidable rules that can be checked at compile time, sometimes with injected runtime checks.

102.9 Tooling: how to make type safety real in a codebase

102.9.1 Static analysis (MSVC)

Microsoft's C++ Core Guidelines Checkers map guideline rules to warnings, including:

- Type.1 narrowing and cast-related warnings (e.g., C26472),
- Type.5 uninitialized reads,
- Type.6 missing member initialization (C26495).

This helps enforce type safety systematically during builds and reviews.

102.9.2 Static analysis (Clang-Tidy)

Clang-Tidy provides many checks under the `cppcoreguidelines-` family, including type-safety related checks such as:

- flagging C-style casts,
- flagging missing member initialization,
- flagging other unsafe patterns aligned with Core Guidelines profiles.

102.10 Practical checklist

- **Initialization:** every variable and every member is initialized (prefer default member initializers).
- **Conversions:** brace initialization by default; make intentional narrowing explicit (prefer GSL helpers where used).
- **Casts:** remove unnecessary casts; avoid C-style casts; avoid `reinterpret_cast` in normal code.
- **Inheritance:** avoid downcasts; if unavoidable, use `dynamic_cast` and handle failure.
- **Sum types:** use `std::variant` rather than raw unions.
- **Bit reinterpretation:** prefer `std::bit_cast` or `memcpy` for defined behavior.

Chapter 103

Casts

103.1 Why casts matter to type safety

A *cast* is an explicit conversion that can bypass (or weaken) the C++ type system. Casts are sometimes necessary, but they are also a common source of:

- **type confusion** (treating an object as the wrong type),
- **undefined behavior** (invalid downcasts, invalid pointer reinterprets, modifying a truly `const` object),
- **silent loss of information** (narrowing numeric conversions),
- **fragile code** (a future type change makes a previously "safe" cast wrong).

The C++ Core Guidelines Type Safety rules start with a strong message: **avoid casts by default** (Type.1), and in particular avoid the cast forms that hide intent or remove safety checks (Type.2–Type.4).

103.2 Core Guidelines focus: Type.1–Type.4

This chapter is centered around these rules:

- **Type.1:** Avoid casts.
- **Type.2:** Don't use `static_cast` to downcast; use `dynamic_cast` instead.
- **Type.3:** Don't use `const_cast` to cast away `const`.
- **Type.4:** Don't use C-style casts; use named casts and construction instead.

103.3 First principle: prefer "no cast" solutions

Before choosing a cast, check whether you can make the conversion unnecessary.

103.3.1 Prefer correct interfaces

If the caller must cast to call a function, the function likely has the wrong parameter types.

```
void draw_circle(double radius);

void f(int r)
{
    draw_circle(static_cast<double>(r)); // suspicious: why does the caller cast?
}
```

Prefer:

```
void draw_circle(double radius);
void draw_circle(int radius); // overload if int is a common input
```

103.3.2 Prefer construction for value conversions

For value conversion, construction is usually clearer than casting.

```
double d = 3.7;

int a = static_cast<int>(d); // explicit, but still allows narrowing silently
int b{d};                    // prefers brace-init: rejects narrowing in many cases
```

103.3.3 Prefer library facilities for representation work

If you are converting between *representations* (bits/bytes), use well-defined facilities rather than pointer punning:

- `std::bit_cast` (C++20) for trivially copyable types of the same size,
- `std::memcpy` for byte movement.

103.4 A map of cast forms and what they mean

C++ has four named casts plus C-style/functional casts. The named casts exist because different conversions have very different safety properties.

103.4.1 `static_cast`

Use when you want a compile-time checked conversion that is well-defined by the language rules:

- numeric conversions (still can narrow),
- upcasts in class hierarchies (Derived* to Base*),
- user-defined conversions that are not `explicit` or are allowed in that context.

103.4.2 `dynamic_cast`

Use for safe downcasts in polymorphic hierarchies (types with at least one virtual function). It performs a runtime check.

103.4.3 `const_cast`

Use only to add or remove cv-qualification on a pointer/reference. **Never use it to cast away const in order to modify an object that is actually const.** (Type.3)

103.4.4 `reinterpret_cast`

Use only for low-level, carefully justified operations where you fully understand the object model and aliasing rules. In general application code, treat `reinterpret_cast` as a red flag.

103.4.5 C-style and functional casts

Avoid. They can perform multiple kinds of conversions with one syntax and can hide dangerous operations. (Type.4)

103.5 Type.4: Avoid C-style casts

103.5.1 Why they are dangerous

A C-style cast `(T)expr` can behave like a `static_cast`, `const_cast`, or `reinterpret_cast` (or a combination), depending on context. This makes it hard to see what actually happens.

103.5.2 Bad and good

```
struct B { virtual ~B() = default; };
```

```
struct D : B {};

void f(B* pb)
{
    D* p1 = (D*)pb;                 // bad: ambiguous cast
    D* p2 = dynamic_cast<D*>(pb);   // good: checked downcast
}
```

103.5.3 Functional casts vs construction

`T(expr)` sometimes means construction, but can also resemble a cast in code reviews. Prefer brace construction for clarity and narrowing protection:

```
int x1(3.7);  // may compile, narrows silently in many cases
int x2{3.7};  // error if narrowing
```

103.6 Type.1: Avoid casts (practical guidance)

103.6.1 Numeric conversions and narrowing

Narrowing conversions are especially common and especially error-prone. Prefer these options in order:

- use a wider type in the interface (avoid conversion),
- use brace initialization `T{expr}` where possible,
- if narrowing is truly intended, make it explicit and reviewable (for example via a checked narrowing helper such as `gsl::narrow` in projects that use the Guidelines Support Library).

```
#include <cstdint>
```

```
std::uint8_t to_byte(int x)
{
    // Prefer an explicit, reviewed decision for narrowing.
    // In some codebases, a checked helper is used here.
    return static_cast<std::uint8_t>(x);
}
```

103.6.2 Pointer conversions

Prefer to avoid pointer casts entirely. If you are converting:

- **Derived* to Base*:** no cast needed (implicit upcast).
- **Base* to Derived*:** use `dynamic_cast` if you must downcast (Type.2).
- **Unrelated pointer types:** question the design; this is often a bug.

103.7 Type.2: Downcasting safely (`dynamic_cast` vs `static_cast`)

103.7.1 The problem with `static_cast` downcasts

A downcast using `static_cast` performs no runtime check. If the object is not really the target derived type, the program has undefined behavior.

103.7.2 Checked downcast with `dynamic_cast`

```
#include <iostream>

struct Base { virtual ~Base() = default; };
```

```
struct D1 : Base { void f() const { std::cout << "D1\n"; } };
struct D2 : Base { void g() const { std::cout << "D2\n"; } };

void use(Base& b)
{
    if (auto* p = dynamic_cast<D1*>(&b)) {
        p->f(); // safe
    } else {
        // handle non-D1 case
    }
}
```

103.7.3 Design note

If downcasts are frequent, consider redesign:

- add a virtual function to the base,
- use a visitor,
- model alternatives with `std::variant`.

103.8 Type.3: `const_cast` and casting away const

103.8.1 The rule

Do not cast away `const` to modify an object. This breaks the type contract and can be undefined behavior when the original object is truly const.

103.8.2 What is actually safe

`const_cast` can be legitimate in narrow cases:

- to call a legacy API that incorrectly takes `T*` but does not modify the object,
- and only when you can prove the object is not actually const and the call is semantically read-only.

Even then, prefer isolating the workaround in one small wrapper instead of spreading casts.

```
void legacy_read_only_api(char* p); // legacy signature, but does not modify

void safe_wrapper(const char* p)
{
    // Keep the cast local and documented.
    legacy_read_only_api(const_cast<char*>(p));
}
```

103.8.3 What is not safe

```
void bad(const int& r)
{
    int& x = const_cast<int&>(r);
    x = 1; // undefined behavior if r refers to an actually-const object
}
```

103.9 `reinterpret_cast`: the most dangerous named cast

103.9.1 Why it is risky

`reinterpret_cast` can create pointers/references that do not actually refer to an object of the target type. Misuse can violate:

- the C++ object model,
- alignment requirements,

- strict aliasing rules.

In typical application code, it is rarely the right tool.

103.9.2 Prefer defined alternatives for bit reinterpretation

103.9.2.1 C++20: `std::bit_cast`

```
#include <bit>
#include <cstdint>

float u32_to_float(std::uint32_t u) noexcept
{
    return std::bit_cast<float>(u); // defined for trivially copyable, same-size types
}
```

103.9.2.2 Portable fallback: `std::memcpy`

```
#include <cstring>
#include <cstdint>

float u32_to_float_memcpy(std::uint32_t u) noexcept
{
    float f{};
    std::memcpy(&f, &u, sizeof(f));
    return f;
}
```

103.9.3 If you truly need `reinterpret_cast`

Keep it:

- rare,

- localized,
- heavily documented,
- reviewed by experts who understand object lifetime, aliasing, and alignment.

103.10 Casts and object lifetime

A cast does not create an object. It only changes how an expression is *viewed*. Type-safety bugs often happen when a cast is used to pretend an object exists where it does not.

103.10.1 Example: “pretending” bytes are an object

```
#include <cstddef>
#include <cstdint>

struct S { int x; };

void bad(std::uint8_t* bytes)
{
    // This does not create an S object in that storage.
    S* ps = reinterpret_cast<S*>(bytes); // dangerous
    // Using *ps as an S object can be undefined behavior.
}
```

Prefer explicit object construction and correct lifetime management instead of pointer reinterprets.

103.11 Tooling and enforcement (practical)

Most modern toolchains can flag many risky cast patterns:

- C-style casts,
- downcasts using `static_cast`,
- suspicious `reinterpret_cast` uses,
- narrowing conversions and implicit lossy conversions (especially in braces vs equals-init comparisons).

Use these diagnostics as build-breaking checks for new code, and as refactoring guides for older code.

103.12 Review checklist

When you see a cast in code review, ask:

- Can we remove this cast by changing the interface or using overloads?
- Is this a narrowing conversion? If yes, is it intentional and clearly expressed?
- Is this a downcast? If yes, can we redesign? If not, is `dynamic_cast` used and failure handled?
- Does this cast remove `const`? If yes, why is the API not const-correct?
- Does this cast reinterpret representation? If yes, can we use `std::bit_cast` or `memcpy` instead?
- Is the cast limited to a small, well-documented low-level boundary?

Chapter 104

`static_cast`

104.1 Role of `static_cast` in type safety

`static_cast<T>(expr)` is C++'s *named* cast for conversions that are checked at compile time according to the language rules. It is commonly used for:

- explicit numeric conversions,
- converting within an inheritance hierarchy where the relationship is known,
- converting `void*` to a typed pointer (in C++ this is not implicit),
- selecting a specific conversion when overload resolution or implicit conversions would be unclear.

It does *not* provide runtime checking, so it can still be unsafe when used to force a conversion the compiler cannot prove is valid at runtime (especially downcasts).

104.2 Core Guidelines position (Type Safety)

The C++ Core Guidelines Type Safety rules treat `static_cast` as useful but potentially dangerous:

- **Type.1: Avoid casts** (prefer designs that do not require casts; prefer brace initialization for numeric conversions and safer alternatives for narrowing).
- **Type.2: Don't use** `static_cast` **to downcast; use** `dynamic_cast` **instead.**

104.3 What `static_cast` can do (language-approved conversions)

The standard rules are detailed, but the important categories are:

104.3.1 Numeric conversions

You can explicitly convert between arithmetic types (integral and floating-point). The conversion is well-defined, but it may lose information (narrowing, truncation, overflow behavior depending on the target type rules).

104.3.2 Enumeration conversions

You can convert an enumeration value to an integral type (and, with restrictions, integral to enum).

104.3.3 Pointer and reference conversions within a class hierarchy

`static_cast` can:

- upcast (Derived → Base),
- downcast (Base → Derived) *without* runtime checking.

Upcasts are typically safe (and often implicit); downcasts are only safe if the dynamic object is truly of the derived type.

104.3.4 Converting `void*` to a typed pointer

In C++, converting from `void*` to `T*` is explicit; `static_cast` can be used for this. (The resulting pointer must still point to a real `T` object with correct alignment.)

104.3.5 Reference casts and value categories

`static_cast` can be used to cast to reference types (including rvalue references) in ways that affect value category, enabling moves when appropriate.

104.4 What `static_cast` cannot do

104.4.1 It cannot cast away `const`/`volatile`

`static_cast` cannot remove cv-qualification. Removing `const` requires `const_cast`, and doing so to mutate an actually-const object is a Type Safety violation (Type.3) and can be undefined behavior.

104.4.2 It is not a general-purpose "bit reinterpretation" tool

Using `static_cast` is not the right way to reinterpret an object's bytes as a different unrelated type. Such operations are governed by the object model and aliasing rules and typically require different techniques (for example `std::bit_cast` in C++20 for safe bitwise copying between trivially copyable types of the same size).

104.5 Best practices: use `static_cast` only when it makes code safer

104.5.1 Prefer construction and brace initialization for numeric conversions

If you are simply converting a value, prefer construction (especially braces) because it is clearer and can diagnose narrowing at compile time.

```
void demo(double d)
{
    int a{d};                    // preferred: narrowing is diagnosed
    int b = static_cast<int>(d); // explicit, but still allows narrowing silently
}
```

Microsoft's Core Guidelines warning C26472 explicitly recommends *not* using `static_cast` for arithmetic conversions and instead using brace initialization or explicit narrowing helpers (when a codebase uses them).

104.5.2 Use `static_cast` to document intentional, safe conversions

A good use case is when an implicit conversion exists but you want to make it *obvious* and searchable:

```
#include <cstddef>

void reserve_bytes(std::size_t n);

void f(int n)
{
    // If negative values are impossible by design, this documents intent:
    reserve_bytes(static_cast<std::size_t>(n));
}
```

Important: this is only safe if the program logic guarantees that n is non-negative. Otherwise, consider validating first.

104.5.3 Upcasting: prefer implicit conversions, but `static_cast` can clarify

```
struct Base { virtual ~Base() = default; };
struct Derived : Base {};

void takes_base(Base*);

void g(Derived* pd)
{
    takes_base(pd);                          // implicit upcast, fine
    takes_base(static_cast<Base*>(pd));      // sometimes used for clarity
}
```

104.6 The biggest pitfall: downcasting with `static_cast` (Type.2)

104.6.1 Why it is dangerous

A downcast with `static_cast` does not check the dynamic type. If the object is not actually the derived type, later access through the derived type is type confusion and can be undefined behavior.

104.6.2 Bad example

```
#include <iostream>

struct Base { virtual ~Base() = default; };
```

```
struct D1 : Base { void f() const { std::cout << "D1\n"; } };
struct D2 : Base { void g() const { std::cout << "D2\n"; } };

void use(Base& b)
{
    auto& d = static_cast<D1&>(b); // bad: unchecked downcast
    d.f();                         // undefined behavior if b is actually a D2
}
```

104.6.3 Preferred alternative: `dynamic_cast`

When you truly need a downcast in a polymorphic hierarchy, use `dynamic_cast` and handle failure.

```
void use_safe(Base& b)
{
    if (auto* p = dynamic_cast<D1*>(&b)) {
        p->f();
    } else {
        // handle other dynamic types safely
    }
}
```

This is exactly what the Core Guidelines Type.2 rule requires, and clang-tidy provides a dedicated check that flags `static_cast` downcasts and suggests replacing them with `dynamic_cast`.
Microsoft's Core Guidelines checker similarly flags `static_cast` downcasts of polymorphic types and points to Type.2.

104.6.4 Design note: frequent downcasts are a design smell

If code regularly needs downcasts, consider redesign:

- add a virtual function to the base interface,
- apply a visitor pattern,
- model alternatives using `std::variant` rather than inheritance.

These approaches keep behavior type-directed without unsafe casts.

104.7 Working with `void*`: what `static_cast` does and does not guarantee

104.7.1 Converting back from `void*`

```
#include <cstddef>

struct S { int x; };

void* store(S* p) { return p; }

S* load(void* vp)
{
    return static_cast<S*>(vp); // explicit in C++
}
```

This is safe only if:

- the original pointer really pointed to an `S`,
- the lifetime of that `S` is still active,
- alignment requirements are satisfied.

`static_cast` cannot validate these conditions at compile time or runtime.

104.8 Tooling: making `static_cast` usage enforceable

104.8.1 Clang-Tidy

Clang-Tidy includes checks aligned with the Core Guidelines Type Safety profile, including one that flags `static_cast` downcasts and suggests `dynamic_cast`.

104.8.2 MSVC Core Guidelines Checkers

Microsoft's Core Guidelines checkers include:

- guidance to avoid `static_cast` for arithmetic conversions (C26472, aligned with Type.1),
- a warning for `static_cast` downcasts on polymorphic types (aligned with Type.2).

104.9 Practical checklist for reviews

When you see `static_cast` in code:

- Is this conversion avoidable by changing the interface or using overloads?
- Is it a numeric conversion that might narrow? Prefer braces; if narrowing is intentional, make that decision explicit and reviewable.
- Is it a downcast (Base → Derived)? If yes, prefer `dynamic_cast` or redesign.
- Is the source `void*`? Ensure the pointer truly refers to the target type, with correct lifetime and alignment.

Chapter 105

dynamic_cast

105.1 Role of dynamic_cast in type safety

dynamic_cast is the C++ cast used for **runtime-checked** conversions within class hierarchies. It exists to prevent **type confusion** when you have a pointer or reference to a base class and need to safely determine whether the dynamic object is a particular derived type.
In the C++ Core Guidelines Type Safety rules, this is captured directly by **Type.2**:

> Do not use static_cast to downcast; use dynamic_cast instead.

This is enforced by tools such as clang-tidy, which recommends replacing unsafe static_cast downcasts with dynamic_cast.

105.2 When dynamic_cast is allowed

105.2.1 Polymorphic requirement

For casts within an inheritance hierarchy, `dynamic_cast` requires a **polymorphic** source type (a class with at least one virtual function, typically a virtual destructor). This requirement exists because the runtime check relies on runtime type information (RTTI).

```
struct Base { virtual ~Base() = default; }; // polymorphic
struct Derived : Base {};
```

105.2.2 Allowed target types

`dynamic_cast` can cast to:

- pointer or reference to a class type (within a class hierarchy),
- `void*` (to obtain a pointer to the most-derived object).

105.3 How failure is reported (pointers vs references)

105.3.1 Pointer form: returns null

If the target type is a pointer type and the runtime check fails, `dynamic_cast` returns the null pointer value of the target type.

```
struct Base { virtual ~Base() = default; };
struct D1 : Base {};
struct D2 : Base {};

void f(Base* pb)
{
    if (auto* p = dynamic_cast<D1*>(pb)) {
        // success: pb actually points to a D1 (or to a type derived from D1)
```

```
    } else {
        // failure: pb is not a D1
    }
}
```

105.3.2 Reference form: throws `std::bad_cast`

If the target type is a reference type and the runtime check fails, an exception matching `std::bad_cast` is thrown.

```
#include <typeinfo>   // std::bad_cast

struct Base { virtual ~Base() = default; };
struct D1 : Base {};

void g(Base& b)
{
    try {
        D1& d = dynamic_cast<D1&>(b); // throws if b is not a D1
        (void)d;
    } catch (const std::bad_cast&) {
        // handle failure
    }
}
```

105.3.3 Guideline: prefer pointer `dynamic_cast` in general code

For most application code, prefer the pointer form:

- it avoids exceptions for the normal “not that type” case,
- it encourages explicit handling of failure with a simple null check.

Use the reference form when failure is truly exceptional and exception-based flow is appropriate.

105.4 What `dynamic_cast` can do

105.4.1 Safe downcast (Base → Derived)

This is the primary use and the focus of Type.2.

```
#include <iostream>

struct Base { virtual ~Base() = default; };
struct Circle : Base { void draw() const { std::cout << "Circle\n"; } };
struct Square : Base { void draw() const { std::cout << "Square\n"; } };

void draw_if_circle(Base* p)
{
    if (auto* c = dynamic_cast<Circle*>(p)) {
        c->draw();
    }
}
```

105.4.2 Cross-cast and side-cast (between siblings via a shared base)

`dynamic_cast` can convert across a hierarchy when the runtime object contains the requested subobject unambiguously and publicly (common in multiple inheritance).

```
struct A { virtual ~A() = default; };
struct B { virtual ~B() = default; };

struct C : A, B {}; // multiple inheritance
```

```
void h(A* pa)
{
    if (auto* pb = dynamic_cast<B*>(pa)) {
        // pb points to the B subobject within the same most-derived C object
        (void)pb;
    }
}
```

105.4.3 Cast to `void*`: pointer to the most-derived object

If you cast a pointer to `void*`, the result is a pointer to the most-derived object (useful in some low-level interop and diagnostic scenarios).

```
struct Base { virtual ~Base() = default; };
struct D : Base { int x = 1; };

void demo(Base* pb)
{
    void* pmost = dynamic_cast<void*>(pb);
    (void)pmost;
}
```

105.5 What `dynamic_cast` does not do

105.5.1 It does not replace good design

If code frequently needs downcasts, it often indicates a design problem:

- missing virtual functions in the base interface,
- logic that should be expressed with polymorphism, a visitor, or `std::variant`,
- ad-hoc type switching that would be clearer as a discriminated union.

The Core Guidelines recommend using `dynamic_cast` only when class-hierarchy navigation is truly necessary.

105.5.2 It cannot be used for non-polymorphic base types

If the base is not polymorphic, `dynamic_cast` (for hierarchy conversions) is not available. Tools note this as a common reason a `static_cast` might appear, but the safer fix is usually to redesign or make the base polymorphic if runtime checking is required.

105.5.3 It is not for unrelated types or representation reinterpretation

`dynamic_cast` is strictly about navigating class hierarchies (and `void*` to most-derived pointer). It is not a tool for bit reinterpretation, numeric narrowing, or unrelated pointer conversions.

105.6 RTTI and build configuration considerations

105.6.1 RTTI must be enabled for the runtime check

The runtime check uses RTTI infrastructure. If RTTI is disabled in a build configuration, `dynamic_cast` within hierarchies is generally not usable (except for limited cases that do not require RTTI). When you plan to rely on `dynamic_cast`, treat RTTI as a required feature of the build for that component.

105.6.2 Interoperability note (exceptions on some implementations)

Some implementations document additional failure modes beyond the standard rules (for example, MSVC documents a `__non_rtti_object` exception in certain invalid-object

scenarios). This is not portable behavior and should not be used as a program contract; write code that respects the standard guarantees and avoids invalid pointers/references.

105.7 Performance and usage guidance

105.7.1 Cost model (high-level)

`dynamic_cast` performs a runtime check whose cost depends on the implementation and the class hierarchy (especially with multiple inheritance). It is typically fine for occasional checks at boundaries, but it should not be used as a hot-path type dispatch mechanism.

105.7.2 Preferred patterns

- Use `dynamic_cast` at **boundaries** (plugins, deserialization, framework interop) where you must validate a dynamic type.
- Keep the check **local** and handle failure immediately.
- If you have repeated type-switching, prefer **virtual functions**, a **visitor**, or `std::variant` to express alternatives directly.

105.8 Common mistakes and how to avoid them

105.8.1 Mistake 1: using `static_cast` for downcasts

This is precisely what Type.2 warns against. Use `dynamic_cast` when you need a checked downcast.

105.8.2 Mistake 2: assuming failure cannot happen

Even when “logically” a pointer should be a certain derived type, code evolves. Always code the failure path.

105.8.3 Mistake 3: using `dynamic_cast` as a design substitute

If your architecture depends on frequent downcasts, improve the interface rather than adding more casts.

105.9 Review checklist

When reviewing a `dynamic_cast`:

- Is RTTI enabled for this component/build?
- Is the source type polymorphic (virtual function present)?
- Is the cast truly necessary, or can a virtual function/visitor/`std::variant` express the logic more safely?
- Is failure handled correctly (`nullptr` for pointers, `std::bad_cast` handling for references)?
- Is the cast on a hot path where a different design would be better?

Chapter 106

Avoiding `reinterpret_cast`

106.1 Why `reinterpret_cast` is a type-safety hazard

`reinterpret_cast` is the most dangerous named cast in C++. It can produce a pointer or reference that *does not actually refer* to an object of the target type, and it can easily violate:

- **type-based aliasing rules** (often called the "strict aliasing rule"),
- **alignment requirements**,
- **object lifetime rules** (using storage as if an object exists there),
- **class invariants** (treating a base object as a derived object).

Because of this, the C++ Core Guidelines Type-safety profile **bans** `reinterpret_cast` as a default rule.

106.2 Core Guidelines perspective (Type Safety)

106.2.1 Type.1 and the type-safety profile

The Core Guidelines emphasize avoiding casts in general (Type.1) and explicitly state that the type-safety profile bans `reinterpret_cast`.

106.2.2 Reasoning

If a codebase allows widespread `reinterpret_cast`, it becomes difficult to trust types as contracts. Small refactors (changing a member type, adding padding, changing inheritance) can silently turn "working" code into undefined behavior.

106.3 What `reinterpret_cast` actually does (and does not do)

106.3.1 It changes the *view*, not the object

A cast does not create a new object. `reinterpret_cast` only changes how an expression is *treated*. If there is no real object of the target type at that address (with the right lifetime and alignment), using the result is undefined behavior.

106.3.2 Pointer round-trip is not permission to dereference

The standard permits converting a pointer to a different pointer type and converting back (a "round trip") under certain conditions, but *dereferencing the converted pointer* is only valid in very limited cases (notably, when the access is allowed by the aliasing rules and alignment is satisfied).

106.4 The three big failure modes

106.4.1 Failure mode 1: violating type-based aliasing rules

In C++, reading or writing an object through a glvalue of an unrelated type is typically undefined behavior. Compilers exploit this rule for optimizations (type-based alias analysis).

106.4.1.1 Example: “type punning” through a pointer cast (wrong)

```
#include <cstdint>

float pun(std::uint32_t u)
{
    // WRONG: treating u's storage as if it were a float object
    return *reinterpret_cast<float*>(&u); // undefined behavior
}
```

106.4.1.2 Correct alternatives

C++20: `std::bit_cast` `std::bit_cast` copies the object representation in a defined way (for trivially copyable types of equal size).

```
#include <bit>
#include <cstdint>

float u32_to_float(std::uint32_t u) noexcept
{
    return std::bit_cast<float>(u);
}
```

Portable fallback: `std::memcpy`

```
#include <cstdint>
#include <cstring>
```

```
float u32_to_float_memcpy(std::uint32_t u) noexcept
{
    float f{};
    std::memcpy(&f, &u, sizeof(f));
    return f;
}
```

106.4.2 Failure mode 2: alignment violations

Every object type has an alignment requirement. If you create a pointer to a type with stricter alignment than the underlying storage provides, dereferencing that pointer is undefined behavior.

106.4.2.1 Example: misaligned access via `reinterpret_cast`

```
#include <cstdint>

void bad(std::uint8_t* p)
{
    // p may not be aligned for std::uint32_t
    auto* u32 = reinterpret_cast<std::uint32_t*>(p); // dangerous
    *u32 = 0x12345678; // undefined behavior if p is misaligned
}
```

106.4.2.2 Correct alternatives

- Copy into an aligned object with `memcpy`.
- Use a serialization/deserialization layer that reads bytes safely.
- If you truly need an object in raw storage, ensure alignment and create the object (see lifetime section).

106.4.3 Failure mode 3: object lifetime violations

Even if the bytes exist, you cannot treat storage as if an object of type `T` exists there unless the lifetime of a `T` object has started in that storage. A pointer cast does not start an object lifetime.

106.4.3.1 Example: pretending bytes are an object (wrong)

```
#include <cstddef>
#include <cstdint>

struct S { int x; int y; };

void bad(std::uint8_t* bytes)
{
    auto* ps = reinterpret_cast<S*>(bytes); // does not create an S object
    ps->x = 1; // undefined behavior: no S object is alive there
}
```

106.4.3.2 Correct alternatives: create an object or treat as bytes

Create an object explicitly (placement new)

```
#include <new>
#include <cstdint>

struct S { int x; int y; };

void good(std::uint8_t* storage)
{
    // storage must be suitably aligned for S
    S* ps = ::new (storage) S{1, 2}; // starts lifetime of S
    (void)ps;
}
```

Or treat the data as bytes If you only need to transport/store bytes, do not pretend they are a `T`. Use `std::byte` or `unsigned char` buffers and explicit parsing.

106.5 Common misuse patterns and safe replacements

106.5.1 Pattern: C-style "type punning" for performance

106.5.1.1 Problem

People often use `reinterpret_cast` to reinterpret bits as another type.

106.5.1.2 Preferred replacement

- `std::bit_cast` when the intent is "same bits, different type" (C++20).
- `memcpy` otherwise.

106.5.2 Pattern: "view a struct header at the start of a byte buffer"

106.5.2.1 Problem

```
struct Header { std::uint32_t magic; std::uint16_t ver; };

const Header* bad_view(const std::uint8_t* buf)
{
    return reinterpret_cast<const Header*>(buf); // dangerous: alignment + lifetime
}
```

106.5.2.2 Preferred replacement: decode explicitly

```
#include <cstdint>
#include <cstring>
```

```
struct Header { std::uint32_t magic; std::uint16_t ver; };

Header parse_header(const std::uint8_t* buf) noexcept
{
    Header h{};
    std::memcpy(&h, buf, sizeof(h)); // still assumes layout matches the serialized format
    return h;
}
```

For robust binary formats, prefer field-by-field decoding with fixed endianness rules rather than assuming the in-memory layout matches the serialized layout.

106.5.3 Pattern: downcasting in polymorphic hierarchies

106.5.3.1 Problem

Using `reinterpret_cast` (or unchecked casts) to treat a base as a derived type is type confusion.

106.5.3.2 Preferred replacement

Use `dynamic_cast` for checked downcasts (Type.2) or redesign with virtual functions or `std::variant`.

106.6 What is allowed: byte-wise access and object representations

106.6.1 Accessing raw bytes of an object

C++ permits examining an object's representation through character-like byte types (such as `unsigned char`) for tasks like hashing, serialization, and debugging. This is fundamentally different from pretending the object is a different unrelated type.

106.6.2 Recommended practice

Prefer explicit byte views and conversion functions:

- keep representation code in a small, well-audited module,
- move "bit tricks" to named helper functions with tests,
- prefer standard library primitives (`bit_cast`, `memcpy`) over pointer punning.

106.7 When `reinterpret_cast` might be justified (rare)

The Core Guidelines treat `reinterpret_cast` as a last resort. If you decide it is necessary, apply all of the following constraints:

- **Localize it** to a tiny low-level boundary (FFI, OS, device register mapping).
- **Document the invariants**: alignment, lifetime, aliasing constraints, ownership.
- **Prefer read-only views** and avoid writing through reinterpreted pointers.
- **Add static assertions** for alignment/size assumptions where applicable.
- **Add tests** that exercise multiple compilers and optimization levels.

Even then, consider whether a different approach is possible (layout wrappers, explicit parsing, `bit_cast`, `memcpy`).

106.8 Tooling: enforce “no `reinterpret_cast`”

Many codebases make this practical by enabling Core-Guidelines-based checks. The Core Guidelines explicitly state the type profile bans `reinterpret_cast`, enabling automated enforcement.

106.9 Review checklist

When you see `reinterpret_cast` in code review, ask:

- Is the code trying to do type punning? If yes, can it use `std::bit_cast` or `memcpy`?
- Could alignment be violated? Is the underlying storage correctly aligned for the target type?
- Does an object of the target type actually exist there (lifetime started), or are we pretending?
- Are we relying on undefined aliasing behavior?
- Can we redesign the API to avoid the cast entirely?

Part XVIII

Object Lifetime Safety (Lifetime)

Chapter 107

Lifetime.1 – Lifetime.6

107.1 Overview: what the Lifetime safety profile is trying to guarantee

The Lifetime safety profile in the C++ Core Guidelines targets one core property:

> Do not access an object through a pointer (or pointer-like view) that might be invalid.

The Guidelines describe lifetime problems as a major source of errors (dangling pointers/references, invalid iterators, invalid views), and present **Pro.lifetime** as a profile intended to be enforced by a combination of style rules, static analysis, and library support.

107.1.1 Important note about numbering

In the current published Core Guidelines text, the **Lifetime safety profile summary** explicitly lists **Lifetime.1** and describes the overall impact. The broader lifetime profile work (as

implemented in tools and described in tool documentation and vendor posts) expands this into multiple practical rules that collectively enforce the same idea: do not use, pass, or return potentially dangling pointers/references/views.
This chapter presents **Lifetime.1 – Lifetime.6** as a coherent, tool-oriented breakdown that matches the published intent and common enforcement model.

107.2 Lifetime.1: Don't dereference a possibly invalid pointer (detect or avoid)

107.2.1 Rule

Never dereference a pointer (or pointer-like object) unless you can justify that it points to a live object (not uninitialized, not null, not out of bounds, not dangling, not invalidated).

107.2.2 Typical sources of invalid pointers

- Uninitialized pointers.
- `nullptr`.
- Pointer to a local that went out of scope.
- Pointer to deleted object (use-after-free).
- Pointer/iterator/view invalidated by container mutation or reallocation.

107.2.3 Bad example: pointer to a local that goes out of scope

```
int* leak_local()
{
```

```
    int x = 42;
    return &x; // BAD: returns pointer to dead object
}
```

107.2.4 Good patterns

- Return by value (often optimized by NRVO/move).
- Return an owning handle (`std::unique_ptr<T>`, `std::shared_ptr<T>`) when ownership must be transferred/shared.
- For non-owning access, pass references or views only when the caller can guarantee lifetime.

107.3 Lifetime.2: Don't create dangling pointers or references

107.3.1 Rule

Do not produce a pointer or reference that outlives the object it refers to.
This includes:

- storing addresses/references to locals that will go out of scope,
- storing pointers/references to objects managed elsewhere without an explicit lifetime relationship,
- binding references/views to temporaries that die immediately.

107.3.2 Bad example: returning a reference to a local

```
int& bad_ref()
{
```

```
    int x = 0;
    return x; // BAD: reference dangles after return
}
```

107.3.3 Bad example: binding a view to a temporary

```
#include <string>
#include <string_view>

std::string make();

void f()
{
    std::string_view sv = make(); // BAD: sv refers to destroyed temporary
    (void)sv;
}
```

This pattern is explicitly highlighted by lifetime-profile tooling discussions and examples.

107.3.4 Good alternatives

```
#include <string>
#include <string_view>

// Return owning object:
std::string make();

// Use owning string (safe):
void g()
{
    std::string s = make();
    std::string_view sv = s; // OK: view refers to s, which is alive here
    (void)sv;
```

```
}
```

107.4 Lifetime.3: Don’t pass a potentially dangling pointer/reference to another function

107.4.1 Rule

Do not pass a pointer/reference/view to a callee if it might be invalid at the call site, or if the callee could store it beyond the caller’s guarantee.
Lifetime-oriented tooling summarizes this contract as “don’t pass a potentially dangling pointer to another function.”

107.4.2 Bad example: passing a pointer to a local

```
void store_for_later(const int* p);

void h()
{
    int x = 7;
    store_for_later(&x); // BAD if store_for_later keeps p after h() ends
}
```

107.4.3 Preferred interface patterns

- Use `T&` or `const T&` for *non-storing* parameters whose lifetime is guaranteed for the duration of the call.
- If the callee must store the object, pass an *owner* type: `std::unique_ptr<T>` or `std::shared_ptr<T>`.

- For sequences, prefer `std::span<T>` to express a non-owning range with explicit size, but still require the underlying storage to outlive the span.

107.5 Lifetime.4: Don't return a potentially dangling pointer/reference/view

107.5.1 Rule

Never return a pointer/reference/view that refers to an object whose lifetime ends at function exit (locals, temporaries, or parameters that do not outlive the return value).
Lifetime-oriented tooling summarizes this contract as "don't return a potentially dangling pointer from any function."

107.5.2 Bad example: returning `std::string_view` to a local string

```
#include <string>
#include <string_view>

std::string_view bad()
{
    std::string s = "hello";
    return s; // BAD: view refers to s which is destroyed
}
```

107.5.3 Good alternatives

```
#include <string>
#include <string_view>

// 1) Return owning object:
```

```
std::string good_owning()
{
    return "hello";
}

// 2) Return view only when referring to storage that outlives the function:
std::string_view good_view(std::string_view input)
{
    // OK: returns a view into the caller-provided view (same lifetime as input)
    return input.substr(0, 3);
}
```

107.6 Lifetime.5: Avoid invalidation by mutation (iterators, pointers into containers, views)

107.6.1 Rule

Do not keep pointers/references/iterators/views into a container across operations that can invalidate them (reallocation, erase, insert, push_back, etc.), unless you can prove they remain valid.
Lifetime-checking examples explicitly call out invalidated iterators after container modification.

107.6.2 Bad example: iterator invalidation

```
#include <vector>

void bad()
{
    std::vector<int> v = {1,2,3};
```

```
    auto it = v.begin();
    v.push_back(4);  // may reallocate, invalidating it
    *it = 0;         // BAD: it may be invalid
}
```

107.6.3 Safer patterns

- Reacquire iterators after mutation.
- Use indices carefully (still beware of reallocation if you keep raw pointers).
- Prefer algorithms that do not require holding iterators across mutating calls.

107.7 Lifetime.6: Make ownership and lifetime relationships explicit

107.7.1 Rule

Prefer designs where ownership is visible in types and APIs, and non-owning access is clearly marked and short-lived.
The Guidelines emphasize making lifetime/ownership clear and enabling tool enforcement through conventions and supporting library types.

107.7.2 Practical guidance

- Use RAII owners (`std::vector`, `std::string`, `std::unique_ptr`, `std::shared_ptr`) to ensure automatic destruction at scope end.
- Avoid “naked” owning raw pointers; if legacy constraints force them, mark intent (e.g., with an ownership annotation type such as GSL’s `owner<T*>`) and centralize deletion.

- Keep non-owning pointers as *observers* and avoid storing them unless a stronger lifetime mechanism exists (object registry, stable storage, shared ownership, explicit parent/child lifetime).

107.7.3 Example: explicit ownership via `unique_ptr`

```
#include <memory>

struct Node
{
    std::unique_ptr<Node> left;
    std::unique_ptr<Node> right;
};

std::unique_ptr<Node> make_tree()
{
    auto root = std::make_unique<Node>();
    root->left  = std::make_unique<Node>();
    root->right = std::make_unique<Node>();
    return root; // OK: ownership transferred to caller
}
```

107.8 Enforcement model (what tools typically flag)

The Core Guidelines describe profiles as intended for tool enforcement. In practice, lifetime checking tools focus on:

- dereferencing pointers that might be dangling,
- passing/returning pointers that might dangle,
- common dangling patterns involving temporaries, iterators, and views.

Vendor documentation and examples summarize these as the core contracts and show typical diagnostics.

107.9 Review checklist

- Is every dereference provably safe (initialized, non-null if required, in-range, not dangling)?
- Are any pointers/references/views stored beyond the lifetime of their target?
- Are any views bound to temporaries?
- Are iterators/pointers into containers used after possible invalidation?
- Does the API make ownership explicit (value return, `unique_ptr`, `shared_ptr`) and non-ownership explicit (references/views) with clear lifetime constraints?

Chapter 108

Dangling References

108.1 Definition and core risk

A **dangling reference** is a reference that still exists after the object it refers to has already been destroyed (its lifetime ended). Using a dangling reference is undefined behavior: the program may appear to work, fail unpredictably, or become a security vulnerability.
A key fact (often underestimated) is that a reference has its own lifetime, and that lifetime can outlast the lifetime of the referred object. cppreference explicitly notes that the referred object may die earlier, making dangling references possible.

108.2 Where dangling references come from

Dangling references usually appear in a small number of recurring patterns.

108.2.1 Returning a reference to a local object

108.2.1.1 The bug

A local object dies when the function returns. Any reference to it becomes dangling.

```
int& bad()
{
    int x = 42;
    return x; // BAD: reference to local; x is destroyed on return
}
```

108.2.1.2 Guideline

The C++ Core Guidelines explicitly forbid returning a pointer or reference to a local object (directly or indirectly).

108.2.1.3 Fixes

```
int good_value()
{
    int x = 42;
    return x; // GOOD: return by value
}
```

If you must return a non-owning view, ensure it refers to storage that outlives the function (e.g., a parameter view, a static object, or an externally owned buffer) and document the contract.

108.2.2 Binding references to temporaries

108.2.3 Lifetime of temporaries and reference lifetime-extension

A temporary object usually dies at the end of the **full-expression**. Binding a reference can extend a temporary's lifetime, but only in specific cases and only for that reference.

108.2.3.1 Safe lifetime extension (simple local case)

```
#include <string>

void ok()
{
    const std::string& r = std::string("hello"); // temporary lifetime extended to r's
    ↪  lifetime
    (void)r;
} // temporary destroyed here, when r goes out of scope
```

108.2.3.2 The trap: you cannot extend a temporary's lifetime by "passing it on"

cppreference states a crucial rule: *in general, the lifetime of a temporary cannot be further extended by initializing a second reference from a reference that was bound to the temporary*.

```
#include <string>

const std::string& id(const std::string& s) { return s; }

void bad()
{
    const std::string& r = id(std::string("tmp"));
    // BAD: the temporary is destroyed at the end of the full-expression containing the
    ↪  call,
    // so r dangles immediately after the initialization statement ends.
    (void)r;
}
```

108.2.4 Dangling references via `std::string_view` and other non-owning views

`std::string_view` is a non-owning view. It does not extend lifetimes; it just stores a pointer and a length. cppreference includes a canonical example where a `string_view` is initialized from a temporary `std::string` and becomes dangling at the end of the statement.

108.2.4.1 The bug

```
#include <string>
#include <string_view>

using namespace std::string_literals;

void bad()
{
    std::string_view v = "temporary"s; // BAD: view refers to a temporary std::string
    // The temporary dies at the end of this statement, v dangles.
    (void)v;
}
```

108.2.4.2 Safe patterns

```
#include <string>
#include <string_view>

std::string make();

void good()
{
    std::string s = make();   // owning
    std::string_view v = s;   // view into a live object
    (void)v;
```

```
}
```

108.2.4.3 Rule of thumb for views

- Passing a view (`std::string_view`, `std::span`) *as a parameter* is usually safe when the function does not store the view beyond the call.
- Returning or storing a view is safe only if you can guarantee the referenced storage outlives the view.

This aligns with the Lifetime safety profile goal: don't use, pass, or return potentially dangling references/pointers/views.

108.2.5 Dangling references after container mutation (invalidation)

A reference to an element inside a container can become dangling after operations that invalidate references/iterators (e.g., reallocation in `std::vector`, erasure in many containers).

108.2.5.1 Example: reference invalidation in `std::vector`

```
#include <vector>

void bad()
{
    std::vector<int> v = {1,2,3};
    int& r = v[0];
    v.push_back(4); // may reallocate; r may now dangle
    r = 10;         // BAD: potentially writes through a dangling reference
}
```

108.2.5.2 Safer patterns

- Do not keep references/iterators across operations that may invalidate them.

- Reacquire the reference after mutation.
- Use stable containers when you truly need stable references (and still follow their rules).

108.3 Consequences: why dangling references are severe

Dangling references are not “just a crash risk.” They can:

- silently corrupt data,
- cause rare heisenbugs that appear only under optimization,
- become exploitable use-after-free bugs in security-sensitive code.

These are precisely the kinds of lifetime errors the C++ Core Guidelines Lifetime profile was created to prevent and diagnose.

108.4 How to prevent dangling references (engineering rules)

108.4.1 Prefer returning values

Return by value unless you have a strong reason not to. The Core Guidelines explicitly warn against returning references/pointers to locals and emphasize returning references only when you are returning a reference to an object that *will outlive* the call and the semantics are clear.

108.4.2 Make ownership explicit

If a function needs to produce an object that outlives the call:

- return the object by value, or
- return an owning smart pointer to transfer/share ownership.

```
#include <memory>

struct Widget { int x = 0; };

std::unique_ptr<Widget> make_widget()
{
    return std::make_unique<Widget>();
}
```

108.4.3 Treat references as non-owning and short-lived

A reference does not own. Do not store references unless you can prove and document that the referent outlives the reference (e.g., a member reference to a parent object with a strict construction/destruction order).

108.4.4 Be strict with non-owning views

- Accept `std::string_view` parameters freely for read-only string-like input when you do not store the view.
- Avoid storing `std::string_view` as a data member unless the object also owns (or otherwise guarantees) the underlying storage.
- Avoid returning `std::string_view` unless it refers to stable storage (e.g., a parameter view, string literal, or data owned elsewhere).

cppreference's examples show exactly why a view can dangle when created from a temporary.

108.4.5 Use tooling based on the Lifetime profile

The Lifetime profile work (as described by WG21 papers and adopted by analysis tools) aims to detect common local cases of dangling references/pointers/views at compile time using

deterministic rules.

108.5 Practical review checklist

When reviewing code that uses references, ask:

- Does the referenced object **definitely** outlive the reference?
- Are we binding to a temporary, and if so, is lifetime extension actually applicable in this form?
- Are we returning a reference/view that could refer to a local, temporary, or parameter that will not outlive the result?
- Are we holding references into containers across operations that might invalidate them?
- Could returning by value or returning an owner make the lifetime contract unambiguous?

Chapter 109

Lifetime Extension

109.1 What “lifetime extension” means

In C++, many expressions create **temporary objects** (often called *prvalues* that are materialized into temporary objects). By default, a temporary is destroyed at the end of the **full-expression** that created it (roughly: the end of the statement).
Lifetime extension is a special rule that can keep a temporary alive *longer than the end of the full-expression* when the temporary is **directly bound to a reference** in specific initialization forms. This rule is essential for safe and efficient code, but it also has important exceptions that can create dangling references and dangling views.

109.2 Why lifetime extension is important for lifetime safety

The C++ Core Guidelines Lifetime profile focuses on preventing dangling pointers, dangling references, and dangling non-owning views. A common root cause is misunderstanding when a temporary is destroyed and whether binding it to a reference actually extends its lifetime.

109.3 The default rule: temporaries die at end of the full-expression

Consider a temporary created during initialization:

```
#include <string>

void f()
{
    std::string s = std::string("hello"); // temporary used to initialize s
    // the temporary is destroyed at the end of the full-expression (this statement)
}
```

Here, lifetime extension is not relevant, because the temporary is used to initialize an owning object (`s`), and the program then uses `s`.

109.4 The main lifetime extension rule: binding to a reference

109.4.1 Const lvalue reference extends the temporary

If a temporary is **directly bound** to a **const lvalue reference**, the temporary's lifetime is extended to match the lifetime of that reference.

```
#include <string>

void ok()
{
    const std::string& r = std::string("hello");
    // The temporary std::string lives as long as r lives (until end of scope).
    (void)r;
}
```

109.4.2 Rvalue reference also extends the temporary (since C++11)

Since C++11, a temporary bound directly to an **rvalue reference** also has its lifetime extended to match the reference.

```
#include <string>

void ok2()
{
    std::string&& r = std::string("hello");
    // The temporary lives as long as r lives (until end of scope).
    (void)r;
}
```

109.4.3 Key constraint: the binding must be direct and in the right form

The lifetime extension rule is not "any time a reference refers to a temporary". It is triggered by specific *initialization* rules. Misunderstanding this leads to bugs.

109.5 The most important traps and exceptions

This section is the core of practical safety: situations where programmers expect lifetime extension but do not get it (or get a shorter lifetime than expected).

109.5.1 Trap 1: binding through a reference parameter does *not* extend beyond the call

A temporary bound to a reference parameter in a function call lives only until the end of the full-expression containing that call.

```
#include <string>
```

```
const std::string& id(const std::string& s) { return s; }

void bad()
{
    const std::string& r = id(std::string("tmp"));
    // The temporary bound to parameter s is destroyed at the end of this full-expression
    // (the initialization statement). r then dangles.
    (void)r;
}
```

109.5.2 Trap 2: lifetime generally cannot be “extended again” by returning a reference

A crucial rule (emphasized in reference initialization documentation) is that, in general, a temporary’s lifetime cannot be further extended by initializing a second reference from a reference that was bound to the temporary.
The example above (`id(std::string("tmp"))`) is a classic manifestation of this rule.

109.5.3 Trap 3: `std::string_view` and other non-owning views do not extend lifetimes

A view type (like `std::string_view`) typically stores a pointer and a length. It does not own the storage and does not extend lifetimes.

```
#include <string>
#include <string_view>

using namespace std::string_literals;

void bad_view()
```

```
{
    std::string_view v = "Hello"s; // temporary std::string
    // The temporary string is destroyed at the end of this statement; v dangles.
    (void)v;
}
```

Safe pattern:

```
#include <string>
#include <string_view>

std::string make();

void good_view()
{
    std::string s = make();      // owning
    std::string_view v = s;      // view into a live string
    (void)v;
}
```

This aligns directly with the Lifetime profile goal: avoid passing/returning/storing potentially dangling views.

109.5.4 Trap 4: reference members and `new`-initializers (not "as long as the object")

A temporary bound to a reference in the initializer used in a `new`-expression exists only until the end of the full-expression containing that `new`-expression, not as long as the dynamically allocated object. This can create a dangling reference member.

```
#include <string>

struct Holder
```

```
{
    const std::string& r;
};

Holder* make_bad()
{
    // The temporary std::string does NOT live as long as *Holder.
    // It lives only to the end of the full-expression containing this new-expression.
    return new Holder{ std::string("tmp") }; // r becomes dangling soon
}
```

109.5.5 Trap 5: aggregate initialization subtlety (braces vs parentheses, since C++20)

For an aggregate with a reference member, initializing with braces can extend the temporary's lifetime, but direct-initialization with parentheses can produce a dangling reference in cases described in the reference initialization rules (since C++20).

```
struct A { int&& r; };

void demo()
{
    A a1{7}; // OK: lifetime is extended
    A a2(7); // well-formed, but r is dangling
    (void)a1; (void)a2;
}
```

109.6 When lifetime extension *does* happen reliably

Use lifetime extension deliberately in these common, safe forms:

109.6.1 Local reference bound directly to a temporary

```
#include <vector>

void ok()
{
    const std::vector<int>& v = std::vector<int>{1,2,3};
    // The temporary vector lives until v goes out of scope.
    (void)v;
}
```

109.6.2 Rvalue reference bound directly to a temporary

```
#include <string>

void ok()
{
    std::string&& s = std::string("abc");
    // The temporary lives to end of scope, but treat s as a normal object reference.
    (void)s;
}
```

109.7 Engineering guidelines: using lifetime extension safely

109.7.1 Prefer owning results over reference tricks

If you need a value to live beyond a statement, prefer returning/storing an owning object (e.g., `std::string`) rather than relying on subtle lifetime extension rules.

109.7.2 Avoid returning references to objects that might be temporaries

The Lifetime profile work emphasizes that returning references can easily produce dangling when a reference binds to a temporary through a call chain. Prefer returning by value unless you are returning a reference to an object whose lifetime is clearly guaranteed.

109.7.3 Treat non-owning views as "borrowed" and short-lived

If you accept `std::string_view` or `std::span`:

- It is usually fine as an input parameter when you do not store it.
- Returning or storing it is safe only when the referred storage is guaranteed to outlive the view.

This is the practical heart of lifetime safety.

109.8 Review checklist (quick)

When you see a reference initialized from an expression, ask:

- Is the referent a temporary? If yes, is this one of the *direct binding* forms that extends lifetime?
- Is the reference obtained through a function call (reference parameter, returned reference)? If yes, assume the temporary dies at end of the full-expression unless you can prove otherwise.
- Is the program storing a view (`string_view`, `span`) to data produced by a temporary? If yes, it likely dangles.
- Is a reference member being initialized via `new` or an aggregate constructor-like form? Re-check the specific exception rules.

Chapter 110

Temporaries

110.1 What a temporary is

A **temporary object** is an unnamed object created by the language to hold a value for a limited time during expression evaluation. In modern C++, temporaries are closely tied to:

- **prvalues** (pure rvalues) and their **materialization** into objects when an object is actually needed,
- **full-expressions** (the usual boundary for destruction),
- **reference binding** (which may extend the temporary lifetime in specific cases).

The core language rules governing temporaries are part of the object lifetime model.

110.2 Why temporaries matter for lifetime safety

Most lifetime bugs in modern C++ are not caused by "raw memory" tricks, but by **borrowing from a temporary**:

- returning a reference/view to a temporary,
- storing a pointer/reference/view that was initialized from a temporary,
- binding a view (e.g., `std::string_view`, `std::span`) to data owned by a temporary.

The C++ Core Guidelines Lifetime profile and WG21 lifetime-safety work focus heavily on these patterns because they create **dangling references/views** with undefined behavior.

110.3 When temporaries are created

The standard's rules are detailed, but in practice, temporaries commonly arise from the following sources:

110.3.1 Temporary materialization (since C++17 wording model)

A prvalue is materialized into a temporary object when a real object is needed, for example:

- binding a reference to a prvalue,
- constructing an `std::initializer_list` from a braced-init-list,
- returning a prvalue from a function,
- performing certain conversions that produce prvalues.

110.3.2 Temporary results of conversions

A conversion may create a prvalue result that becomes a temporary object when used where an object is required (notably when binding a reference).

110.3.3 Braced initialization and `std::initializer_list`

List-initialization can create a temporary array or backing storage used by `std::initializer_list`. This is a well-known source of subtle lifetime issues when people store pointers/iterators into initializer-list backing storage.

110.4 Default destruction rule: end of the full-expression

Unless a lifetime-extension rule applies, a temporary is destroyed at the end of the **full-expression** that created it (typically, the end of the statement).

110.4.1 Example: temporary dies at end of statement

```
#include <string>

void f()
{
    const char* p = std::string("hello").c_str();
    // BAD: the temporary std::string is destroyed at the end of this statement,
    // so p dangles immediately after the statement ends.
    (void)p;
}
```

110.5 Lifetime extension: when a temporary can live longer

The most important exception is **binding directly to a reference** in certain initialization forms.

110.5.1 Direct binding extends lifetime

If a temporary is **directly bound** to a reference in a way covered by the reference-initialization rules, its lifetime is extended to match the lifetime of the reference (with important exceptions described below).

```
#include <string>

void ok()
{
    const std::string& r = std::string("hello");
    // The temporary std::string lives as long as r (until the end of scope).
    (void)r;
}
```

110.5.2 Rvalue references

In modern C++ rules, direct binding to an rvalue reference can also extend the temporary's lifetime in the covered forms.

```
#include <string>

void ok2()
{
    std::string&& r = std::string("hello");
    // The temporary lives as long as r (until the end of scope).
    (void)r;
}
```

110.6 The most important non-extension cases (common traps)

Understanding these traps is essential for the Lifetime safety profile.

110.6.1 Trap 1: binding through a function parameter does not extend past the full-expression

Passing a temporary to a function that takes a reference does *not* make it live beyond the full-expression containing that call. This becomes dangerous when you return that reference.

```
#include <string>

const std::string& id(const std::string& s) { return s; }

void bad()
{
    const std::string& r = id(std::string("tmp"));
    // The temporary bound to parameter s is destroyed at the end of this statement.
    // r dangles after the initialization statement completes.
    (void)r;
}
```

110.6.2 Trap 2: you generally cannot “extend again” by returning a reference

cppreference emphasizes that, in general, the lifetime of a temporary cannot be further extended by initializing another reference from a reference bound to that temporary. The example above is a direct consequence.

110.6.3 Trap 3: non-owning views do not extend lifetime

`std::string_view` (and similar view types) is non-owning: it does not keep the underlying object alive. cppreference includes canonical examples where a view is initialized from a temporary `std::string` and becomes dangling at the end of the statement.

```
#include <string>
#include <string_view>
using namespace std::string_literals;

void bad_view()
{
    std::string_view v = "hello"s; // temporary std::string
    // The temporary dies at the end of this statement; v dangles.
    (void)v;
}
```

Safe pattern:

```
#include <string>
#include <string_view>

std::string make();

void good_view()
{
    std::string s = make();     // owning
    std::string_view v = s;     // view into a live object
    (void)v;
}
```

110.6.4 Trap 4: `new`-initializers and reference members

A temporary used to initialize a reference member in a new-expression does not automatically live as long as the dynamically allocated object; the reference-initialization rules contain a specific exception here.

```
#include <string>

struct Holder { const std::string& r; };

Holder* make_bad()
{
    // The temporary does not live as long as *Holder.
    return new Holder{ std::string("tmp") }; // r may dangle soon
}
```

110.6.5 Trap 5: temporaries in `std::initializer_list`

Objects backing an `initializer_list` are temporary. Storing pointers/references into that backing storage is unsafe unless the storage is known to outlive the use.

```
#include <initializer_list>

const int* bad()
{
    std::initializer_list<int> il = {1,2,3};
    return il.begin(); // BAD: points to temporary backing array that will be destroyed
}
```

110.7 Temporaries, return values, and guaranteed copy elision

Since C++17, many return-by-value patterns do not create an extra temporary due to **guaranteed copy elision** in common cases. This strengthens the recommendation to **return by**

value rather than returning references to avoid lifetime issues.

```
#include <string>

std::string make_name()
{
    return std::string("name"); // returned object is constructed directly in the caller in
    ↪  common C++17 rules
}
```

110.8 Temporaries and sequencing: destruction order notes

Temporary destruction is tied to full-expressions and sequencing rules. Most code should not rely on subtle destruction order of multiple temporaries within complex expressions; keep expressions simple at lifetime boundaries and prefer named local variables for clarity and safety.

110.9 Guidelines for writing lifetime-safe code with temporaries

110.9.1 Prefer owning results to borrowed views

If you need data to outlive a statement, store an owning type (`std::string`, `std::vector`) rather than a borrowed pointer/view produced from a temporary.

110.9.2 Avoid returning references/views unless the referent lifetime is guaranteed

Returning a reference/view is safe only when it refers to storage that is guaranteed to outlive the returned reference/view (for example, data owned by the caller or stable global storage).

110.9.3 Use lifetime extension deliberately, not accidentally

Binding a temporary directly to a local reference can be safe, but avoid chains where a reference to a temporary is passed through functions, stored in objects, or returned.

110.9.4 Treat view types as "borrowed"

Using `std::string_view` or `std::span` as function parameters is often excellent for performance and clarity, but storing or returning them requires a clear lifetime contract.

110.10 Review checklist

When you see a temporary-producing expression, ask:

- Does any pointer/reference/view outlive the full-expression that created the temporary?
- Is lifetime extension actually guaranteed here, or are we binding through a parameter/return path?
- Are we creating a view (`string_view`, `span`) from a temporary owner?
- Are we taking addresses/iterators into `initializer_list` backing storage?

Chapter 111

Ownership Clarity

111.1 Purpose of this chapter

Lifetime safety problems in C++ are rarely caused by "mysterious compiler behavior." They are usually caused by unclear ownership:

- Who is responsible for destroying an object?
- How long is the object guaranteed to live?
- Is a pointer/reference/view merely observing, or is it owning?

The C++ Core Guidelines lifetime profile (`Pro.lifetime`) emphasizes preventing dangling pointers/references/views, and the fastest practical way to do that is to make ownership obvious in types and interfaces.

111.2 Definitions: ownership, borrowing, and lifetime

111.2.1 Ownership

An **owner** is responsible for an object's lifetime: it creates (or receives) the object and ensures it is destroyed exactly once, at the right time. In modern C++, ownership is typically represented by:

- automatic storage duration objects (RAII on the stack),
- standard library owning types (`std::string`, `std::vector`, etc.),
- smart pointers (`std::unique_ptr`, `std::shared_ptr`).

111.2.2 Borrowing (non-owning access)

A **borrower** (observer) uses an object but does not control its lifetime. Borrowing is usually represented by:

- references (`T&`, `const T&`),
- raw pointers used as observers (`T*`, `const T*`) when they do not imply ownership,
- views (`std::span`, `std::string_view`) which are explicitly non-owning.

Borrowing must be paired with a clear lifetime guarantee: the borrowed object must outlive the borrower.

111.2.3 Lifetime safety goal

The lifetime profile aims to ensure you do not dereference or propagate pointers/references/views that might be invalid (dangling, out of bounds, uninitialized, etc.). Ownership clarity makes such errors easier to prevent and easier for tools to diagnose.

111.3 Core Guidelines principle: encode ownership in types

The Core Guidelines strongly discourage “naked owning raw pointers” and encourage RAII and standard ownership types, because raw pointers do not communicate whether they own.

111.3.1 Rule of thumb

- If the function **creates** something and gives it to the caller: return an owning type (value, `unique_ptr`, or `shared_ptr`).
- If the function **uses** something but does not store it: take a reference or a view.
- If the function **stores** something beyond the call: take ownership explicitly (usually `unique_ptr` or `shared_ptr`).

111.4 Owning types: the standard options

111.4.1 Prefer value ownership when practical

Returning and storing objects by value is often the simplest and safest approach. In modern C++ (notably since C++17), returning by value is frequently efficient due to copy elision and moves.

```
#include <string>

std::string make_name()
{
    return "config.json"; // owning return by value
}
```

111.4.2 `std::unique_ptr`: single owner

`std::unique_ptr<T>` expresses exclusive ownership: exactly one `unique_ptr` owns a given object, and ownership can be transferred by move.

```
#include <memory>

struct Widget { int x = 0; };

std::unique_ptr<Widget> make_widget()
{
    return std::make_unique<Widget>();
}

void consume(std::unique_ptr<Widget> w) // takes ownership
{
    // use w, then it is destroyed automatically
}
```

111.4.3 `std::shared_ptr`: shared ownership

`std::shared_ptr<T>` expresses shared ownership: the object lives as long as at least one `shared_ptr` owns it. This is powerful but has costs and risks (cycles, unclear ownership boundaries). Use it when shared lifetime is truly needed.

```
#include <memory>

struct Session {};

std::shared_ptr<Session> make_session()
{
    return std::make_shared<Session>();
}
```

```
void use_session(std::shared_ptr<Session> s) // shares ownership
{
    // s keeps the Session alive during and after this call if stored
}
```

111.4.4 Avoid owning raw pointers

Owning with raw pointers (`T*` with `new`/`delete`) hides ownership and is a common source of leaks, double-deletes, and use-after-free. Prefer RAII owners.

111.5 Non-owning types: make borrowing explicit

111.5.1 References for mandatory non-null parameters

If a function requires a valid object, prefer a reference:

```
struct Config { int port = 0; };

void load(Config& cfg);           // must be valid
void print(const Config& cfg);    // read-only borrow
```

A reference expresses "must not be null," but it still requires a lifetime guarantee: the referent must outlive the reference use.

111.5.2 Raw pointers as observers for optional objects

Use `T*` when "no object" is meaningful (nullable), and document whether the pointer is non-owning.

```
struct Logger {};
```

```
void log(Logger* opt, const char* msg)
{
    if (opt) {
        // non-owning use
        (void)msg;
    }
}
```

111.5.3 `std::span` for non-owning ranges

`std::span<T>` is a non-owning view over contiguous elements; it makes size explicit and avoids pointer+length mistakes, but it does not own the data.

```
#include <span>
#include <cstddef>

void hash_bytes(std::span<const std::byte> bytes); // borrows
```

111.5.4 `std::string_view` for string-like input

`std::string_view` is a non-owning view; it is excellent for parameters but dangerous to store or return unless the underlying storage outlives the view.

```
#include <string>
#include <string_view>

void parse_line(std::string_view line); // safe if not stored

std::string_view bad()
{
    std::string s = "tmp";
    return s; // BAD: returning view to destroyed object
}
```

111.6 Ownership clarity in interfaces

111.6.1 Rule: express ownership transfer in the type

If a function takes ownership, the type should make it obvious.

111.6.1.1 Taking ownership with `unique_ptr`

```
#include <memory>

struct Node {};

class Tree
{
public:
    void set_root(std::unique_ptr<Node> n) // takes ownership
    {
        root_ = std::move(n);
    }

private:
    std::unique_ptr<Node> root_;
};
```

111.6.1.2 Sharing ownership with `shared_ptr`

```
#include <memory>

struct Image {};

class Cache
{
public:
```

```
    void store(std::shared_ptr<Image> img) // shares ownership
    {
        img_ = std::move(img);
    }

private:
    std::shared_ptr<Image> img_;
};
```

111.6.2 Rule: do not store borrowed pointers/views without a lifetime mechanism

Storing a `T*`, `T&`, `std::span`, or `std::string_view` as a member is safe only if the class also enforces that the referenced data outlives it (for example: the class owns the data, or it is tied to a parent object with a strict lifetime relationship). Otherwise, it is a common source of dangling.

111.7 Annotating legacy code: ownership markers and non-null contracts

Some codebases must interoperate with APIs that use raw pointers for ownership. The Guidelines Support Library (GSL) provides annotation types commonly used with the Core Guidelines:

- `gsl::owner<T*>` to mark an owning raw pointer in legacy interfaces,
- `gsl::not_null<T*>` to express a non-null pointer contract.

These annotations do not magically make code safe, but they make intent explicit and support review and analysis.

```
#include <gsl/gsl>

gsl::owner<int*> make_int_legacy()
{
    return new int{42}; // ownership is explicit in the type
}

void take_nonnull(gsl::not_null<int*> p)
{
    *p = 7; // contract: p is not null
}
```

111.8 Common ownership anti-patterns and safe replacements

111.8.1 Anti-pattern: function returns a borrowed pointer/reference to a local

```
int* bad()
{
    int x = 0;
    return &x; // BAD: dangling pointer
}
```

Replacement: return by value or return an owning smart pointer.

111.8.2 Anti-pattern: storing raw pointers without a lifetime policy

```
struct Engine {};
struct Car
{
```

```
    Engine* e; // unclear: owning or observing?
};
```

Replacement options:

- `std::unique_ptr<Engine>` if `Car` owns the engine,
- `Engine&` or `gsl::not_null<Engine*>` if `Car` borrows and requires it,
- `Engine*` if optional borrowing is intended, with explicit documentation.

111.8.3 Anti-pattern: returning or storing `string_view` to a temporary

Replacement: return/store an owning `std::string` or ensure the referred storage is stable and documented.

111.9 Design patterns that improve ownership clarity

111.9.1 Factory functions

Prefer factory functions that return owning types, making ownership transfer explicit:

```
#include <memory>

struct Service {};

std::unique_ptr<Service> make_service(); // caller owns result
```

111.9.2 Pimpl (ownership of implementation)

Pimpl typically stores the implementation as a `std::unique_ptr<Impl>` to make ownership explicit and keep lifetime automatic.

```
#include <memory>

class Widget
{
public:
    Widget();
    ~Widget(); // defined out-of-line
    Widget(Widget&&) noexcept;
    Widget& operator=(Widget&&) noexcept;

private:
    struct Impl;
    std::unique_ptr<Impl> p_;
};
```

111.9.3 Observer pointers vs. owning pointers (naming and conventions)

A practical convention used in many teams:

- owners are named like `owner`, `ptr`, `handle` and use RAII types,
- observers are named like `observer`, `view`, `opt` and never delete.

Conventions are not a substitute for types, but they support reviews and maintenance.

111.10 Ownership clarity checklist for reviews

- For every pointer/reference/view: is it owning or borrowing?
- If owning: is it represented by a standard owner type (value, `unique_ptr`, `shared_ptr`)?
- If borrowing: is the lifetime guarantee clear and enforceable (not “hope-based”)?

- Are any views (`span`, `string_view`) stored or returned without a stable backing store?
- Are raw pointers used in interfaces? If yes, can we replace them, or at least annotate (`owner`, `not_null`)?
- Does the design prevent dangling after container mutation (invalidation) or after temporary destruction?

Part XIX

Bounds Safety (Bounds)

Chapter 112

Bounds Safety (Bounds.1 – Bounds.4)

112.1 Scope and intent of the Bounds profile

The **Bounds safety** profile in the ISO C++ Core Guidelines targets one specific family of failures: **accessing memory outside the bounds of an allocated object** (most commonly, arrays and contiguous buffers). The profile focuses on removing (or strongly restricting) the two dominant root causes of bounds errors:

- pointer arithmetic (computing new pointers by adding/subtracting integers),
- unchecked indexing (using `operator[]` and similar operations without guaranteed checking).

The Core Guidelines define bounds safety as: *a program does not use an object to access memory outside of the range that was allocated for it*. This profile is intended to be complete only when combined with **Type safety** and **Lifetime safety**.

112.1.1 Why this matters

Out-of-bounds reads and writes are a major source of:

- correctness bugs that are difficult to reproduce,
- security vulnerabilities (notably buffer overflows),
- memory corruption and invariant violations.

Even an out-of-bounds *read* can break program correctness by observing bytes that are not part of the intended object, possibly interpreting them as the wrong type.

112.2 Core idea: represent ranges explicitly

The profile's practical direction is:

- Treat raw pointers as **pointers to single objects**, not as **implicit ranges**.
- When you have a range, use a **range type** (such as `span`) that carries both **address** and **size**.
- Prefer operations that can be checked (statically where possible; otherwise at runtime).

112.2.1 Key vocabulary and types

Contiguous range A sequence of elements stored in adjacent memory (e.g., `T[N]`, `std::array<T,N>`, `std::vector<T>`, `std::span<T>`, a buffer returned by an API).

`std::span` **vs** `gsl::span` The Guidelines historically use the Guidelines Support Library (GSL) to provide lightweight helper types. They note that `std::span` was adopted into the standard library, but `std::span` does *not* guarantee bounds checking on element access, whereas GSL's `span` is described as bounds-checked by default in the guideline text.

`gsl::at` Microsoft's C++ Core Guidelines checker explicitly recommends using `gsl::at()` as a bounds-checked access alternative to unchecked subscripting in contexts where the checker can help enforce the profile.

112.3 Rule Bounds.1: Don't use pointer arithmetic. Use `span` instead

112.3.1 Rule statement

Bounds.1: *Don't use pointer arithmetic. Use* `span` *instead.*

112.3.2 Rationale

Pointer arithmetic silently turns a single pointer into a *range cursor* without carrying the range bounds. This makes it easy to compute a pointer that points:

- before the beginning of an array,
- past the end of an array,
- into unrelated storage (especially after refactoring).

A `span` (or another explicit range abstraction) preserves both *start* and *extent*, enabling:

- safer iteration,
- clearer contracts at interfaces,
- better tool enforcement.

The Guidelines explicitly describe pointer arithmetic as best done *within* spans.

112.3.3 Bad examples

```
void f(int* p, int n)
{
    // p is being used as if it were a range (but n is separate and unchecked)
    for (int* it = p; it != p + n; ++it) {   // pointer arithmetic
        *it = 0;
    }

    int* q = p + n; // may point one past the end (valid to form, unsafe to dereference)
    *q = 42;        // out-of-bounds write (undefined behavior)
}
```

112.3.4 Preferred approach: take a span (range-as-a-value)

```
#include <span>

void f(std::span<int> s)
{
    for (int& x : s) {
        x = 0;
    }

    // If you need a subrange, create one explicitly.
    // (subspan performs range computations in a single place, and expresses intent)
    if (s.size() >= 1) {
        auto tail = s.subspan(1);
        (void)tail;
    }
}
```

112.3.5 Practical guidance

- For APIs: prefer `std::span<T>` or `std::span<const T>` parameters when the function expects a contiguous range.
- For raw buffers from C APIs: wrap them immediately in a span *at the boundary*.
- Prefer `for (auto& e : range)` and algorithms over manual indexing.

112.3.6 Enforcement and tool support

A tool can flag pointer arithmetic by detecting arithmetic on pointer-typed expressions producing pointer results. Microsoft's CppCoreCheck maps this to diagnostics such as `C26481 NO_POINTER_ARITHMETIC` and recommends using span instead.

112.4 Rule Bounds.2: Only index into arrays using constant expressions

112.4.1 Rule statement

Bounds.2: *Only index into arrays using constant expressions.*

112.4.2 Rationale

Unchecked dynamic indexing into a raw array (including `T[N]` and `std::array<T,N>`) is a classic source of out-of-bounds access. If the index is a compile-time constant, tools (and compilers) can often prove it is within range and diagnose mistakes early.
This rule is intentionally strict: it pushes you away from raw array indexing for variable indices and toward safer abstractions (`span`, `vector` with `at()`, validated indices, or algorithmic iteration).

112.4.3 Bad examples

```
#include <array>

int read_bad(const std::array<int, 4>& a, int i)
{
    return a[i]; // i is dynamic and unchecked
}
```

112.4.4 Better options

112.4.4.1 Option A: prefer range iteration (often simplest)

```
#include <array>

int sum(const std::array<int, 4>& a)
{
    int r = 0;
    for (int x : a) r += x;
    return r;
}
```

112.4.4.2 Option B: if you need indexing, make bounds checking explicit

```
#include <array>
#include <stdexcept>

int read_checked(const std::array<int, 4>& a, int i)
{
    if (i < 0 || i >= static_cast<int>(a.size())) {
        throw std::out_of_range("index");
    }
    return a[static_cast<std::size_t>(i)];
```

```
}
```

112.4.4.3 Option C: use a checked access helper (when available in your environment)

Many environments provide `gsl::at(container, index)` as a checked access helper and tools can recommend it.

112.4.5 Enforcement and tool support

The Guidelines describe an enforcement strategy:

- Flag indexing into array types where the index is not a compile-time constant within `[0, bound)`.

Microsoft's checker maps this to diagnostics such as `C26482 NO_DYNAMIC_ARRAY_INDEXING` and `C26483 STATIC_INDEX_OUT_OF_RANGE`.

112.5 Rule Bounds.3: No array-to-pointer decay

112.5.1 Rule statement

Bounds.3: *No array-to-pointer decay.*

112.5.2 Rationale

In C and C++, an array expression often converts implicitly ("decays") to a pointer to its first element. The moment decay happens:

- the **size information is lost**,
- the callee sees only `T*` and cannot know the bounds unless a separate parameter is correctly provided and consistently used,

- tools have less information to prove correctness.

This is a major reason why APIs like `void f(int*, int)` are fragile: the relationship between pointer and length is not enforced by the type system.

112.5.3 Bad examples

```
#include <cstddef>

void g(int* p, std::size_t n);

void f()
{
    int a[10]{};
    g(a, 10); // a decays to int*, size is now separate and can get out of sync
    g(a, 12); // compiles, but is wrong
}
```

112.5.4 Preferred approach: preserve bounds in the type

112.5.4.1 Use `std::span` to pass the array with its extent

```
#include <span>
#include <cstddef>

void g(std::span<int> s);

void f()
{
    int a[10]{};
    g(a);       // span knows extent (10)
    g(a + 0);   // does not compile: avoids accidental decay patterns at the interface
}
```

112.5.4.2 Use `std::array` or `std::vector` as ownership/range carriers

```
#include <array>
#include <vector>

void use_fixed(const std::array<int, 10>& a);
void use_dyn(const std::vector<int>& v);
```

112.5.5 Enforcement and tool support

The Guidelines propose flagging any expression relying on implicit conversion of an array type to a pointer type. Microsoft's checker maps this to `C26485 NO_ARRAY_TO_POINTER_DECAY`.

112.6 Rule Bounds.4: Don't use standard-library functions and types that are not bounds-checked

112.6.1 Rule statement

Bounds.4: *Don't use standard-library functions and types that are not bounds-checked.*

112.6.2 Rationale

The standard library offers both:

- operations that are **checked** (commonly throwing `std::out_of_range` on failure), and
- operations that are **unchecked** (often `operator[]`, raw-pointer-based algorithms, or legacy C interfaces).

Bounds.4 tells you to prefer the checked forms when bounds errors are plausible and costly, especially at:

- module boundaries,
- parsing, decoding, protocol handling,
- file/network input processing,
- security-sensitive code paths.

112.6.3 Common examples in practice

112.6.3.1 `operator[]` vs `at()`

```
#include <vector>
#include <stdexcept>

int read_vec(const std::vector<int>& v, std::size_t i)
{
    // Unchecked:
    // return v[i];

    // Checked (throws std::out_of_range on invalid i):
    return v.at(i);
}
```

112.6.3.2 Prefer checked access helpers where available

Tools can recommend `gsl::at(container, index)` as a safer alternative to unchecked subscripting in many contexts.

112.6.3.3 Avoid raw pointer + length APIs when a range type is available

```
#include <span>
#include <cstddef>
```

```
int parse_packet(std::span<const std::byte> bytes); // carries bounds

// Avoid: int parse_packet(const std::byte* p, std::size_t n);
```

112.6.4 Performance reality: checking can be optimized

Bounds checking is not automatically "slow." In many real cases:

- checks are hoisted out of loops,
- checks are eliminated when provably unnecessary,
- checks can be enabled in debug/test builds to catch bugs early.

The profile approach is specifically designed to be enforceable and adoptable incrementally, using tools and libraries to obtain safety with acceptable cost.

112.6.5 Enforcement and tool support

Microsoft's CppCoreCheck includes a specific diagnostic encouraging `gsl::at()` and references Bounds.4 directly.

112.7 Profile-wide enforcement, suppression, and engineering workflow

112.7.1 Mechanical enforcement ideas (as described by the Guidelines)

The Bounds profile suggests checks such as:

- flag pointer arithmetic that yields pointer results,
- flag non-constant indexing into array types,

- flag implicit array-to-pointer decay.

112.7.2 Suppression: use sparingly and document why

The Guidelines define a standard suppression mechanism:

```
[[gsl::suppress("tag")]]
[[gsl::suppress("tag", justification: "message")]]
```

where `"tag"` can be a profile name like `"bounds"` or a specific rule like `"bounds.2"`.

Example:

```
[[gsl::suppress("bounds.1", justification: "interop with a legacy C API requires pointer
↪  arithmetic here")]]
void legacy_bridge(char* p, int n);
```

Use suppression only when you have a clearly bounded, reviewed, and tested reason (e.g., ABI constraints, vetted low-level code, or performance-critical inner loops with proven invariants).

112.8 Summary: applying Bounds.1–Bounds.4 in modern C++

- **Bounds.1** removes the most dangerous implicit-range technique: pointer arithmetic; prefer explicit range types (`span`).
- **Bounds.2** makes raw array indexing a compile-time-only operation; otherwise use safer iteration or explicit checked access.
- **Bounds.3** prevents silent loss of bounds by rejecting array-to-pointer decay; keep size with the data.
- **Bounds.4** guides you toward bounds-checked library operations and away from unchecked access patterns, especially at boundaries.

Together, these rules form a coherent, tool-friendly subset that targets a large class of real-world defects and vulnerabilities, and aligns with current C++ safety-profile efforts.

Chapter 113

Out-of-bounds access

113.1 What is out-of-bounds access?

113.1.1 Definition

Out-of-bounds access happens when a program reads from or writes to memory *outside* the storage that belongs to an object. For a contiguous sequence (an array, a `std::vector`, a buffer), valid element indices are `0` to `size-1`. Anything else is out-of-bounds.
The C++ Core Guidelines treat out-of-bounds access as one of the most important classes of defects because it can cause:

- silent data corruption,
- crashes that appear far from the bug,
- security vulnerabilities (e.g., buffer overflows),
- broken type-based assumptions (reading bytes that are not the intended object).

Bounds safety is the profile whose goal is: *do not access beyond the memory allocation of an object, especially arrays and other contiguous ranges*. This chapter focuses on how out-of-bounds access happens, how it looks in real code, and how the Bounds profile prevents it.

113.1.2 A crucial point: C++ does not automatically protect you

In standard C++, most out-of-bounds element accesses have **undefined behavior** (UB). UB means the standard imposes no required outcome: the program may appear to work, may crash, may corrupt data, or may behave differently under optimization.

113.2 How out-of-bounds bugs are created

113.2.1 Unchecked indexing

The most common form is an unchecked index used with `operator[]` or raw arrays.

113.2.1.1 Typical bug: off-by-one

```
#include <vector>
#include <cstddef>

int sum_bad(const std::vector<int>& v)
{
    int s = 0;
    for (std::size_t i = 0; i <= v.size(); ++i) { // BUG: should be i < v.size()
        s += v[i]; // out-of-bounds when i == v.size()
    }
    return s;
}
```

113.2.1.2 Better: range-for or algorithms

```
#include <vector>

int sum_good(const std::vector<int>& v)
{
    int s = 0;
    for (int x : v) s += x;
    return s;
}
```

113.2.2 Pointer arithmetic used as implicit range traversal

Pointer arithmetic can silently walk outside the object because the pointer does not carry bounds. This is exactly why the Bounds profile starts with **Bounds.1: no pointer arithmetic; use** span **instead**.

```
void fill_bad(int* p, int n)
{
    // p is used like the start of a range, but nothing enforces the range
    for (int* it = p; it != p + n; ++it) { // pointer arithmetic
        *it = 0;
    }
}
```

113.2.3 Array-to-pointer decay loses the size

When an array decays to a pointer, the bound is dropped. If a separate length is passed, it can be wrong, mismatched, or forgotten.

```
#include <cstddef>

void process_bad(const int* p, std::size_t n);
```

```
void f()
{
    int a[10]{};
    process_bad(a, 10); // ok by convention only
    process_bad(a, 12); // compiles, but is wrong
}
```

113.2.4 Signed/unsigned and integer narrowing bugs

Even careful code can go out-of-bounds due to type issues:

- negative indices converted to huge unsigned values,
- narrowing from a 64-bit size to a 32-bit int,
- overflow in `i * stride` or pointer offset calculations.

```
#include <vector>
#include <cstddef>

int read_bad(const std::vector<int>& v, int i)
{
    // If i is negative, it becomes a huge size_t after conversion.
    return v[static_cast<std::size_t>(i)];
}
```

113.3 Why it is dangerous even when it “seems to work”

113.3.1 Undefined behavior is optimization-visible

Compilers assume your program follows the language rules. If your code performs UB, the optimizer may transform surrounding code in ways that look surprising. This is why many

out-of-bounds bugs disappear in debug builds and appear in release builds (or only with certain compiler flags).

113.3.2 Out-of-bounds reads can still break invariants

Developers sometimes underestimate reads: “it is only reading one element past the end.” But reading outside the object may read unrelated bytes and interpret them as the wrong type or value, which can break invariants and lead to incorrect decisions.

113.4 Bounds profile strategy against out-of-bounds access

The Core Guidelines **Bounds Safety (Bounds)** profile reduces out-of-bounds access by removing the most common unsafe mechanisms and forcing range information to be explicit and checkable.

113.4.1 Principle 1: represent ranges as a single object

Prefer `std::span<T>` (or a similar range type) to pass a contiguous range:

- the pointer and the size travel together,
- interfaces document intent (a range, not “a pointer to something”),
- tools can enforce profile rules more effectively.

```
#include <span>

void fill(std::span<int> s)
{
    for (int& x : s) x = 0;
}
```

113.4.2 Principle 2: avoid the unsafe building blocks

The Bounds profile's first rules address the direct causes of out-of-bounds access:

113.4.2.1 Bounds.1: Don't use pointer arithmetic; use `span` instead

Pointer arithmetic is treated as a primary "enemy" of bounds safety, because it makes out-of-bounds pointers easy to compute.

113.4.2.2 Bounds.2: Only index into arrays using constant expressions

This rule pushes variable indexing away from raw arrays and toward safer alternatives. If the index is a compile-time constant, tools and compilers can often prove it is in range.

113.4.2.3 Bounds.3: No array-to-pointer decay

Prevent silent loss of size information.

113.4.2.4 Bounds.4: Don't use library functions and types that are not bounds-checked

Prefer checked forms when the correctness and safety benefits matter.

113.5 Checked access: what "checked" means in practice

113.5.1 `at()` as a standard checked access

Several standard containers provide `at()`, which performs bounds checking and reports errors (typically by throwing `std::out_of_range`). This makes the failure mode explicit and testable.

```
#include <vector>
#include <cstddef>
```

```
int read_checked(const std::vector<int>& v, std::size_t i)
{
    return v.at(i); // checked access
}
```

113.5.2 Prefer expressing the loop so no per-iteration check is needed

Checked indexing inside a hot loop can be the wrong structure. Often, you can restructure code so that:

- one check happens before the loop,
- the loop body is then safe without repeated checks,
- compilers can optimize better.

```
#include <span>
#include <cstddef>

int dot(std::span<const int> a, std::span<const int> b)
{
    if (a.size() != b.size()) return 0; // policy choice for mismatch
    int s = 0;
    for (std::size_t i = 0; i < a.size(); ++i) {
        s += a[i] * b[i]; // indexing is now safe by construction (same bound)
    }
    return s;
}
```

113.6 Bridging to legacy C and low-level APIs safely

113.6.1 Wrap at the boundary

When a C API gives you `T*` + length, immediately wrap it into a range object and keep the raw pair local.

```
#include <span>
#include <cstddef>

void consume_buffer(std::span<const unsigned char> bytes);

extern "C" void legacy_api(const unsigned char* p, std::size_t n);

void bridge(const unsigned char* p, std::size_t n)
{
    consume_buffer(std::span<const unsigned char>(p, n));
}
```

113.6.2 When pointer arithmetic is unavoidable

Some low-level code must do pointer arithmetic (device registers, custom allocators, packet decoding with fixed formats). In such code, keep the unsafe operations:

- small,
- isolated,
- heavily tested,
- documented with clear preconditions.

The Guidelines ecosystem supports suppressing specific warnings when justified (for example, for a vetted low-level bridge), but suppression should be rare and explained.

113.7 Tooling and enforcement

113.7.1 Static analysis

Static analyzers can detect many patterns strongly associated with out-of-bounds access:

- pointer arithmetic producing pointers,
- dynamic indexing into array types,
- array-to-pointer decay at call sites and assignments.

Microsoft's C++ Core Guidelines checker (CppCoreCheck) includes specific diagnostics for the Bounds profile, such as warnings that correspond to **Bounds.1** and **Bounds.3**, and guidance to use `span` and bounds-checked access helpers where appropriate.

113.7.2 Runtime sanitizers and hardened builds

Runtime tools can catch out-of-bounds access during testing:

- address-based checking tools (commonly used in CI) detect heap/stack/global buffer overflows,
- undefined-behavior checking tools detect various forms of UB, including some out-of-bounds patterns.

These are test-time defenses, not a substitute for correct interfaces and safe coding rules.

113.8 Practical checklist for preventing out-of-bounds access

- Use **range types** to carry pointer + size together: `std::span` for non-owning contiguous ranges.

- Prefer **range-for** loops and standard algorithms over manual indexing.
- If you must index, ensure the index is provably in-range and consider **checked access** (`at()` or equivalent) at boundaries.
- Avoid **pointer arithmetic** in application code; keep it isolated in low-level, reviewed components.
- Avoid **array-to-pointer decay** in interfaces; accept arrays/ranges as `span`, `std::array`, or containers.
- Enable static analysis rules that enforce the Bounds profile, and use sanitizers in testing.

113.9 Summary

Out-of-bounds access is one of the most damaging categories of C++ bugs because it is typically undefined behavior and can corrupt memory or enable vulnerabilities. The Core Guidelines Bounds Safety profile reduces these failures by:

- discouraging pointer arithmetic and implicit ranges,
- forbidding array-to-pointer decay at interfaces,
- restricting unsafe indexing patterns,
- preferring explicit ranges and checked operations where appropriate.

Chapter 114

span

114.1 Purpose and role in the Bounds profile

114.1.1 What `span` is

`std::span<T, Extent>` is a lightweight, **non-owning view** over a **contiguous** sequence of objects. It stores (conceptually) a pointer to the first element and a size (unless the extent is known at compile time). A span does *not* allocate, free, copy, or move the elements it refers to.

114.1.2 Why the Core Guidelines emphasize `span`

The Bounds Safety profile aims to prevent out-of-bounds access by making ranges explicit and checkable. A raw pointer does not carry bounds, so it is easy to compute invalid addresses via pointer arithmetic or to mismatch pointer+length pairs at interfaces. The Guidelines therefore recommend using `span` as the primary vocabulary type for passing contiguous ranges.

114.1.3 When to use `span`

Use `span` when a function:

- needs read/write access to a contiguous sequence, but does **not** take ownership,
- should accept multiple container types uniformly (C arrays, `std::array`, `std::vector`, etc.),
- must carry a pointer and size together in a single argument.

`span` is the array/range analogue of `std::string_view`: a view, not an owner.

114.2 Basic model and key properties

114.2.1 Type parameters: element type and extent

`std::span` has two template parameters:

- `T`: the element type (e.g., `int`, `const std::byte`),
- `Extent`: either a compile-time constant (`N`) or `std::dynamic_extent`.

If the extent is static, it is encoded in the type (`span<T, N>`). If the extent is dynamic, the size is stored at runtime (`span<T>`).

114.2.2 Contiguity requirement

A span can refer only to ranges whose elements are stored contiguously. This includes:

- built-in arrays `T[N]`,
- `std::array<T,N>`,

- `std::vector<T>`, `std::basic_string<T>` (when appropriate),
- raw buffers `T*` + length (when wrapped explicitly).

It does not model linked lists or tree traversals because those are not contiguous.

114.2.3 Non-owning and lifetime

A span is valid only as long as the referenced elements remain valid. If the underlying storage is destroyed, reallocated, or moved in a way that invalidates pointers, the span becomes invalid (dangling).

114.3 Construction and conversions

114.3.1 Default construction

A default-constructed span is an empty view. For dynamic extent, that typically means `data()==nullptr` and `size()==0`.

114.3.2 From pointer and count

```
#include <span>
#include <cstddef>

void use(std::span<int> s);

void f(int* p, std::size_t n)
{
    use(std::span<int>(p, n)); // explicit: pointer + count
}
```

114.3.3 From C arrays and `std::array`

```
#include <span>
#include <array>

void g(std::span<int> s);

void f()
{
    int a[4] = {1,2,3,4};
    std::array<int, 3> b = {5,6,7};

    g(a); // extent is known at compile time for the source array
    g(b);
}
```

114.3.4 From contiguous containers

C++ provides constructors that accept a contiguous range whose `data()` and `size()` are usable for constructing a span, subject to constraints on the element type and contiguity.

```
#include <span>
#include <vector>

void h(std::span<const int> s);

void f()
{
    std::vector<int> v = {1,2,3};
    h(v); // non-owning view over v's elements
}
```

114.3.5 Static extent size requirements

When constructing `span<T, N>`, the source range must have exactly `N` elements (unless `N==0` in the special empty cases). If the sizes do not match, the construction is ill-formed (fails to compile), which is an important safety property.

```
#include <span>

void take3(std::span<int, 3> s);

void f()
{
    int a[3]{};
    int b[4]{};

    take3(a); // OK
    // take3(b); // ERROR: extent mismatch
}
```

114.3.6 Initializer-list construction (C++26)

Historically, `std::span` could not be constructed directly from a braced initializer list because of lifetime pitfalls. C++26 adds a converting constructor that enables `std::span<const T>` from an appropriate `std::initializer_list<T>`.

```
#include <span>

void read(std::span<const int> s);

void f()
{
    read({1,2,3}); // C++26: span<const int> over initializer_list
}
```

Important: this is available only for `span<const T>` (not writable spans), consistent with `initializer_list` providing only constant access.

114.3.7 Deduction guides and C++26 improvement

Span has deduction guides that allow the element type (and sometimes static extent) to be deduced from the source. C++26 extends deduction from iterator-sentinel pairs to also deduce static extent when the size argument is "integral-constant-like," and requires contiguous iterators for that guide.

114.4 Access, iteration, and slicing

114.4.1 Core observers

A span provides:

- `size()`, `empty()`,
- `data()`,
- `size_bytes()` for byte size computation.

These allow you to write uniform, safe interfaces without losing bounds.

114.4.2 Element access: `operator[]`

In C++20 and C++23, `std::span::operator[]` is **unchecked** (like many standard containers). That means an invalid index is undefined behavior. This is why the Bounds profile often pairs `span` with tooling and/or checked access helpers (such as GSL utilities).

114.4.3 `span::at()` (C++26)

C++26 adds `std::span::at()` as a bounds-checked access function (feature-test macro guarded), aligning span with other standard containers that provide checked access.

114.4.4 Subspans and view-based slicing

Slicing is one of span's most important features because it reduces error-prone pointer arithmetic. Span provides:

- `first(n)`, `last(n)`,
- `subspan(offset, count)`.

```
#include <span>
#include <cstddef>

int checksum(std::span<const unsigned char> bytes)
{
    int s = 0;
    for (auto b : bytes) s += b;
    return s;
}

int parse(std::span<const unsigned char> packet)
{
    if (packet.size() < 8) return -1;

    auto header = packet.first(8);
    auto body   = packet.subspan(8);

    return checksum(header) + checksum(body);
}
```

Span slicing expresses intent (“header” and “body”) without performing raw pointer arithmetic in user code.

114.5 Byte views and object representation

114.5.1 `as_bytes` and `as_writable_bytes`

Span supports converting a span of objects into a span of bytes representing their object representation:

- `std::as_bytes(span<T>)` returns `span<const std::byte>`,
- `std::as_writable_bytes(span<T>)` returns `span<std::byte>` and participates only if `T` is not `const`.

The resulting span’s extent follows the rules described for `dynamic_extent` vs static extents.

```
#include <span>
#include <cstddef>
#include <cstdint>

std::uint32_t crc32(std::span<const std::byte> bytes);

std::uint32_t crc_of_ints(std::span<const int> ints)
{
    auto bytes = std::as_bytes(ints);
    return crc32(bytes);
}
```

Note: byte views are about object representation, not about “reinterpreting” elements as a different type.

114.6 Using `span` to design safe interfaces

114.6.1 Replacing pointer+length parameters

A classic fragile interface:

```
void process_bad(const int* p, std::size_t n);
```

A safer interface that keeps bounds with the pointer:

```
#include <span>

void process(std::span<const int> s);
```

This directly supports Bounds safety goals by eliminating mismatched pointer-length pairs and making ranges explicit at the type level.

114.6.2 Const-correctness patterns

- `std::span<const T>` for read-only views,
- `std::span<T>` for read-write views.

```
#include <span>

int sum(std::span<const int> s)
{
    int r = 0;
    for (int x : s) r += x;
    return r;
}

void scale(std::span<int> s, int k)
{
    for (int& x : s) x *= k;
}
```

114.6.3 Returning spans: when it is safe

Returning span is safe only if the referenced data outlives the returned span. Safe examples include:

- returning a span into a member-owned buffer *when the object outlives the span usage*,
- returning a span into static storage.

Unsafe is returning a span into a local variable or a temporary container.

```
#include <span>
#include <array>

struct Packet
{
    std::array<unsigned char, 128> buf{};

    std::span<unsigned char> payload() { return buf; } // OK: view into member storage
};

std::span<int> bad()
{
    int a[4] = {1,2,3,4};
    return a; // ERROR: span would dangle (a is local)
}
```

114.7 Safety pitfalls and how to avoid them

114.7.1 Dangling spans

A span is only as safe as the lifetime of its source. Common ways to create dangling spans:

- span referring to a local array that goes out of scope,
- span referring to a `std::vector` that later reallocates,
- span referring to a temporary container returned by value.

The Bounds profile addresses *out-of-bounds*; Lifetime profile addresses *dangling*. They are complementary.

114.7.2 Reallocation invalidation

If a span refers to `std::vector<T>` elements, operations that reallocate the vector's storage invalidate the span (because the underlying pointer becomes invalid).

```
#include <span>
#include <vector>

void f()
{
    std::vector<int> v = {1,2,3};
    std::span<int> s = v;

    v.push_back(4); // may reallocate and invalidate s
    // using s here may be undefined behavior
}
```

114.7.3 Bounds checking expectations: `std::span` vs GSL

The Core Guidelines and tooling often describe `span` as a safer range type in the Bounds profile context. However, `std::span` itself historically provides unchecked `operator[]` (C++20/23), while the Microsoft GSL implementation emphasizes bounds checking in its `gsl::span`. With C++26 adding `span::at()`, the standard library offers a built-in checked access option.

114.8 Tooling support for the Bounds profile

114.8.1 Clang-tidy (Core Guidelines checks)

Clang-tidy provides checks aligned with the Core Guidelines, including checks that flag pointer arithmetic and recommend using span as a safer alternative (Bounds.1 enforcement style).

114.8.2 MSVC C++ Core Guidelines checker

Microsoft's code analysis includes rules that map directly to the Core Guidelines' Bounds profile; for example, warning C26481 states "Don't use pointer arithmetic. Use span instead (bounds.1)."

114.9 Design guidance: choosing between `span`, containers, and other views

114.9.1 Use `span` when you do not own

If your API does not allocate or manage lifetime, take a `span`. This typically reduces interface complexity and removes pointer+length mismatches.

114.9.2 Use an owning container when you own

If your function must store data beyond the call, or allocate and manage it, use an owning type such as: `std::vector<T>`, `std::string`, `std::array<T,N>`, or a domain-specific owning type.

114.9.3 Use `std::string_view` for text

For read-only textual data, `std::string_view` is often the better vocabulary type. Use `std::span<const char>` only when you explicitly want a byte/character sequence with size semantics similar to other binary buffers.

114.10 Summary

`std::span` is the standard vocabulary type for a non-owning view of contiguous data. It supports Bounds Safety by making pointer+size a single, explicit object, enabling safer interfaces and tool enforcement. To use it correctly:

- keep lifetimes valid (a span never extends the lifetime of what it views),
- avoid relying on unchecked indexing; prefer iteration, slicing, and (where available) checked access such as `at()` in C++26,
- wrap legacy pointer+length pairs into spans at module boundaries.

This approach aligns with the C++ Core Guidelines' Bounds profile and with mainstream tool diagnostics that encourage replacing pointer arithmetic with span-based range handling.

Chapter 115

Safe indexing

115.1 What “safe indexing” means

115.1.1 Definition

Safe indexing means accessing elements of a sequence only when the index is guaranteed to be within the valid range `[0, size)`, so that no out-of-bounds read or write is possible.
In the C++ Core Guidelines, **bounds safety** focuses on preventing out-of-bounds access, with two main “enemies” repeatedly highlighted by the profile: **pointer arithmetic** and **array indexing**.

115.1.2 Why unchecked indexing is risky

In standard C++, most out-of-bounds accesses via `operator[]` (raw arrays, `std::span::operator[]`, `std::vector::operator[]`, etc.) are **undefined behavior**. For example, `std::span::operator[]` explicitly has undefined behavior when `idx < size()` is false.

Safe indexing is therefore about choosing designs and techniques that:

- **make the bounds explicit** (so they can be checked),
- **make invalid access hard to write** (so errors are prevented),
- **make failures observable** (so bugs are caught early).

115.2 Bounds profile rules that define safe indexing expectations

Safe indexing in the Core Guidelines is primarily expressed through the Bounds profile rules and their enforcement.

115.2.1 Bounds.2: Constant-expression indexing for arrays

Bounds.2 requires: *Only index into arrays using constant expressions*. Tooling maps this directly (for example, MSVC warning C26482).
The intent is strict: it pushes dynamic indexing away from raw arrays and toward safer abstractions (ranges, spans, checked access, or algorithms).

115.2.2 Bounds.4: Avoid unchecked standard access

Bounds.4 requires: *Don't use standard-library functions and types that are not bounds-checked*.
Tooling guidance for Bounds.4 explicitly recommends using a checked access utility such as `gsl::at()` instead of unchecked subscripting, and also mentions using standard `at()` member functions where available.

115.3 The safest pattern: avoid indexing entirely when possible

Indexing is error-prone; many bugs are simply off-by-one mistakes. A strong "safe indexing" technique is to write code that does not need indices.

115.3.1 Prefer range-for and algorithms

```
#include <vector>

int sum_range_for(const std::vector<int>& v)
{
    int s = 0;
    for (int x : v) s += x; // no index, no bounds risk
    return s;
}
```

This aligns with the Bounds profile goal: fewer opportunities to write unchecked subscripts.

115.4 When you must index: safe indexing strategies

Sometimes indexing is necessary (tables, random access, parallel arrays, protocols). In that case, choose one of the strategies below depending on performance and error-handling needs.

115.4.1 Strategy A: Use checked access (`at()`) when available

Many standard containers provide `at(pos)`, which performs bounds checking and throws `std::out_of_range` if `pos` is invalid.

```
#include <vector>
```

```cpp
#include <cstddef>

int read_checked(const std::vector<int>& v, std::size_t i)
{
    return v.at(i); // checked access; throws std::out_of_range on invalid i
}
```

115.4.1.1 C++26: `std::span::at()`

Historically, `std::span` had unchecked `operator[]` (UB if out of range). C++26 adds `std::span::at()` as a checked alternative, making spans support standard-style checked element access.

115.4.2 Strategy B: Use a checked helper (`gsl::at`) under the Bounds profile

MSVC's Core Guidelines checker explicitly recommends `gsl::at()` for uniform bounds-checked access in the Bounds profile (Bounds.4), and notes that standard `at()` should be used where available.

```cpp
#include <gsl/gsl>
#include <vector>

int read_gsl_at(const std::vector<int>& v, int i)
{
    return gsl::at(v, i); // checked; recommended by Bounds tooling
}
```

115.4.3 Strategy C: Validate once, then index safely

Sometimes throwing per access is undesirable, but you still want safety. A safe approach is:

- validate the index against the range once,
- then use unchecked indexing only after the invariant is established.

```
#include <span>
#include <cstddef>

int read_validated(std::span<const int> s, std::size_t i)
{
    if (i >= s.size()) return 0; // policy: handle error without exceptions
    return s[i];                 // now safe because invariant is checked
}
```

This method keeps safety while allowing you to choose an error policy (return code, optional, expected-like, terminate, etc.).

115.5 Index type safety: avoid signed/unsigned traps

Even with checks, the wrong index type can create bugs.

115.5.1 Use `size_t` for sizes, but be careful with negative indices

Most container `size()` and indexing types are unsigned (`size_t`). A negative `int` index converted to `size_t` becomes a very large number. Safe indexing therefore must ensure:

- user-provided indices are validated for negativity *before* conversion,
- conversions are explicit and done after checks.

```
#include <vector>
#include <cstddef>

int read_user_index(const std::vector<int>& v, int i)
```

```
{
    if (i < 0) return 0;
    auto ui = static_cast<std::size_t>(i);
    if (ui >= v.size()) return 0;
    return v[ui];
}
```

115.5.2 Use `std::ssize` for signed loop indices when needed

C++20 provides `std::ssize` (and `std::ranges::ssize`) to get a **signed** size, which can simplify certain loops (especially reverse loops) while staying consistent with the container size.

```
#include <vector>
#include <iterator> // std::ssize
#include <cstddef>

int last_nonzero_index(const std::vector<int>& v)
{
    for (auto i = std::ssize(v) - 1; i >= 0; --i) {
        if (v[static_cast<std::size_t>(i)] != 0) return static_cast<int>(i);
    }
    return -1;
}
```

Note: even when using a signed index for loop control, convert to `size_t` only after ensuring the value is non-negative.

115.6 Safe indexing for arrays and `std::array`

115.6.1 Follow Bounds.2 strictly for raw arrays

For raw arrays, the Core Guidelines' Bounds.2 is intentionally strict: only constant-expression indices are accepted. Clang-tidy provides a dedicated check for this rule (`cppcoreguidelines-pro-bounds-constant-array-index`) and describes it as part of Bounds.2.

115.6.2 Prefer `std::array` + algorithms, or `span`

If you need dynamic indexing, prefer:

- `std::array` or `std::vector` with `at()` where it fits,
- `std::span` at interfaces to preserve bounds and prevent array-to-pointer decay.

115.7 Safe indexing for containers under Bounds.4

115.7.1 Avoid unchecked container access when tools are enforcing Bounds.4

Clang-tidy provides a Bounds.4-related check for avoiding unchecked container access (`cppcoreguidelines-pro-bounds-avoid-unchecked-container-access`) and documents it as part of the Bounds Safety (Bounds.4) profile.

115.7.2 Practical rule of thumb

- Prefer `at()` when the index may be influenced by external input or complex logic.
- Prefer validation-once + `operator[]` when performance is critical and invariants are clear and enforced.

- Prefer no indexing (range loops/algorithms) whenever possible.

115.8 Designing interfaces that make safe indexing the default

115.8.1 Take `span` instead of pointer+length

`std::span` is a standard vocabulary type that represents a contiguous range with a pointer and size kept together.

```
#include <span>
#include <cstddef>

int parse(std::span<const std::byte> bytes)
{
    if (bytes.size() < 4) return -1; // validate once
    // safe indexing thereafter
    return 0;
}
```

This reduces mismatches and simplifies safe validation patterns.

115.8.2 Prefer slicing over pointer arithmetic

Span supports view operations like `subspan` that help avoid manual offset calculations, reducing the chance of indexing errors.

115.9 Checklist: safe indexing in real code

- Prefer **range-for** and algorithms to eliminate indexing.
- For indexing that must remain:

  - Use **checked access** (`at()`) where available.
  - Under Bounds.4 tooling, prefer `gsl::at()` for uniform checked access.
  - Or validate once (`i < size()`) and then index under the established invariant.

- For arrays, follow Bounds.2: constant-expression indices only; otherwise move to a safer abstraction.
- Avoid signed/unsigned mistakes; use `std::ssize` for signed loop control when it improves correctness.
- Remember: `std::span::operator[]` is unchecked (UB if out-of-range) in C++20/23; C++26 adds `span::at()` for checked access.

Part XX

GSL (Guidelines Support Library)

Chapter 116

GSL.1–GSL.6 (Guidelines Support Library)

116.1 Purpose and scope of the GSL

The **Guidelines Support Library (GSL)** is a small, header-only set of types and utilities intended to make the ISO C++ Core Guidelines practical in real code bases. In particular, it provides lightweight vocabulary types to express intent (ownership, non-nullness, and views over contiguous memory) and a few focused utilities (pre/postcondition checks, narrowing helpers, RAII cleanup helpers, etc.). The Core Guidelines describe the GSL as essential support that prevents the guidelines from becoming overly restrictive on language details.

116.1.1 What "GSL.1–GSL.6" means in this manual

The Core Guidelines section titled *"GSL: Guidelines support library"* is primarily a *specification of components* (views, ownership pointers, assertions, utilities, concepts) rather than a numbered set of rules. In this reference manual, we group the most important, directly

actionable recommendations about using those components into six rules, named **GSL.1** through **GSL.6**, to match the structure used elsewhere in this book.

116.2 GSL.1: Use GSL vocabulary types to make safety assumptions explicit

Rule (GSL.1): Prefer expressing intent in types (not comments): use GSL vocabulary types where they clarify ownership, nullability, and bounds.

116.2.1 Rationale

Many defects come from implicit assumptions:

- "This pointer is never null."
- "This pointer points to exactly one object, not an array."
- "This pointer + length pair must stay consistent."

The GSL provides vocabulary types (notably `not_null` and `span`) to state these assumptions directly in function signatures and local variables. The guidelines explicitly list these view/ownership-related types and explain that views are not owners.

116.2.2 Key idea

- Use `T*` and `T&` as *non-owning* by default.
- Mark exceptional cases (ownership, non-nullness, array views) explicitly with GSL types.

116.3 GSL.2: Use `gsl::not_null` to rule out null pointers

Rule (GSL.2): When null is not a valid value, use `gsl::not_null<T>` to make that guarantee explicit.

116.3.1 Rationale

A `not_null<T>` encodes the invariant that the stored pointer-like value is not `nullptr`. The Core Guidelines list `not_null<T>` as the recommended way to say "this must not be null." In practice, this improves readability, reduces defensive checks, and enables static analysis to detect violations earlier. Microsoft documentation for C++ Core Guidelines checking specifically recommends using `gsl::not_null` to mark values that should never be null.

116.3.2 Example: safer interface

```
#include <gsl/gsl>
#include <string>

struct Logger {
    void write(const std::string& s);
};

void log_line(gsl::not_null<Logger*> lg, const std::string& msg)
{
    lg->write(msg);
}
```

116.3.3 Notes

- `not_null` is about *validity*, not ownership.

- It is appropriate for parameters, returns, and member fields that must never be null.

116.4 GSL.3: Use `gsl::span` for contiguous sequences (avoid pointer+length)

Rule (GSL.3): Represent a contiguous buffer using `gsl::span<T>` (or `std::span<T>` where appropriate), not raw pointer + size pairs.

116.4.1 Rationale

The Core Guidelines describe `span<T>` as a view over a contiguous sequence and state that pointer arithmetic is best done within spans. This directly supports the **Bounds Safety** profile (which aims to eliminate bounds violations) by making the range explicit.

116.4.2 Important compatibility note: `gsl::span` vs `std::span`

The Core Guidelines explain that `std::span` was standardized based on the earlier GSL design, but `std::span` does not guarantee bounds-checking by default; the guidelines describe `gsl::span` as the bounds-safe-by-default counterpart intended to track `std::span`'s interface.

116.4.3 Example: replace pointer+count

```
#include <gsl/span>
#include <cstdint>

std::uint32_t checksum(gsl::span<const std::uint8_t> bytes)
{
    std::uint32_t s = 0;
    for (auto b : bytes) s = (s * 131u) + b;
    return s;
```

```
}
```

116.4.4 Example: subspans instead of pointer arithmetic

```
#include <gsl/span>
#include <cstddef>

int sum_first_k(gsl::span<const int> a, std::size_t k)
{
    if (k > a.size()) k = a.size();
    auto head = a.first(k);
    int s = 0;
    for (int x : head) s += x;
    return s;
}
```

116.5 GSL.4: Mark ownership explicitly (prefer RAII; use `owner<T*>` only for legacy)

Rule (GSL.4): Prefer RAII owners (`std::unique_ptr`, `std::shared_ptr`, containers). If you must keep an owning raw pointer (legacy/ABI/handle internals), mark it as `gsl::owner<T*>`.

116.5.1 Rationale

The Core Guidelines describe:

- `owner<T*>` as a way to mark an owning raw pointer when code cannot be upgraded to a proper resource handle,
- and list typical reasons: conversion cost, ABI constraints, or being part of a resource-handle implementation.

116.5.2 Example: legacy boundary with explicit ownership

```
#include <gsl/gsl>
#include <cstdlib>

// Legacy C API:
extern "C" void* legacy_alloc(std::size_t);
extern "C" void  legacy_free(void*);

gsl::owner<unsigned char*> make_buffer(std::size_t n)
{
    return static_cast<unsigned char*>(legacy_alloc(n));
}

void consume_and_free(gsl::owner<unsigned char*> p)
{
    // ... use p ...
    legacy_free(p);
}
```

116.5.3 Rule of thumb

- If you can, replace `owner<T*>` with a real owner type (smart pointer with correct deleter, or a container).

- If you cannot, `owner<T*>` makes ownership visible and tool-checkable.

116.6 GSL.5: Use `Expects` and `Ensures` to document and enforce contracts

Rule (GSL.5): Use `Expects(expr)` for preconditions and `Ensures(expr)` for postconditions to state local correctness assumptions.

116.6.1 Rationale

The Core Guidelines list `Expects` and `Ensures` as assertion facilities of the GSL and explain they are macros intended as a temporary mechanism pending standard evolution of contracts.

116.6.2 Example: checked preconditions and postconditions

```
#include <gsl/assert>
#include <gsl/span>

int max_value(gsl::span<const int> a)
{
    Expects(!a.empty());

    int m = a[0];
    for (int x : a) if (x > m) m = x;

    Ensures(m >= a[0]);
    return m;
}
```

116.6.3 Guidance

- Use contract checks to defend *interface assumptions*, not to replace normal error handling.

- Prefer expressing assumptions in types first (for example, `not_null`, `span`); then use `Expects`/`Ensures` for the remaining conditions that types cannot express.

116.7 GSL.6: Prefer GSL utilities for safe cleanup and safe narrowing; integrate tool support thoughtfully

Rule (GSL.6): Use GSL utilities (`finally`, `narrow`, `narrow_cast`, `index`) to make intent explicit and reduce common correctness hazards; use tool suppressions only with justification.

116.7.1 Safe cleanup: `finally`

The GSL provides `finally` as an RAII helper: it creates an object whose destructor runs a cleanup action.

```
#include <gsl/util>
#include <cstdio>

void write_file(const char* path)
{
    std::FILE* f = std::fopen(path, "wb");
    if (!f) return;

    auto close_file = gsl::finally([&] { std::fclose(f); });

    // ... write using f ...
}
```

116.7.2 Safe narrowing: `narrow` and `narrow_cast`

The guidelines list:

- `narrow_cast<T>(x)` as an explicit request for a narrowing `static_cast`,
- `narrow<T>(x)` as a checked narrowing that throws on value loss.

```
#include <gsl/narrow>
#include <cstdint>

std::uint8_t to_u8_checked(int x)
{
    return gsl::narrow<std::uint8_t>(x); // throws if x does not fit
}

std::uint8_t to_u8_explicit(int x)
{
    return gsl::narrow_cast<std::uint8_t>(x); // explicit narrowing request
}
```

116.7.3 Indexing consistency: `gsl::index`

The Core Guidelines list `gsl::index` as a dedicated indexing type (currently an alias for `ptrdiff_t`) to support consistent indexing decisions across code.

```
#include <gsl/util>
#include <vector>

int sum_vec(const std::vector<int>& v)
{
    int s = 0;
    for (gsl::index i = 0; i < static_cast<gsl::index>(v.size()); ++i)
        s += v[i];
    return s;
}
```

116.7.4 Tool integration: checked access and suppression with justification

Many teams adopt the Core Guidelines using static analysis tools (for example, Microsoft C++ Core Check and clang-tidy). The Bounds profile emphasizes avoiding unchecked operations and recommends using uniformly checked access patterns, such as `gsl::at()` (or container `at()`) for bounds-checked indexing.

```
#include <gsl/gsl>
#include <vector>

int get_checked(const std::vector<int>& v, int i)
{
    return gsl::at(v, i); // checked access (throws or terminates depending on config)
}
```

When a rule is intentionally not applied in a specific location, Microsoft documentation describes using the `[[gsl::suppress]]` attribute to suppress a specific code analysis warning (optionally with justification).

```
#include <gsl/gsl>

int legacy_adapter(int* p, int n)
{
    // Example: suppression for a carefully reviewed legacy pattern
    [[gsl::suppress("26481", justification: "Legacy API requires pointer+length; validated
    ↪  by tests")]]
    return p[n - 1]; // still dangerous if preconditions are not ensured
}
```

116.7.5 Minimal policy for suppressions

- Suppress *as locally as possible*.

- Always provide a short justification.
- Prefer refactoring to a GSL vocabulary type (often `span` or `not_null`) over suppressing.

116.8 Summary checklist (practical adoption)

- Use `gsl::not_null` when null is invalid.
- Use `gsl::span` to pass buffers; avoid pointer arithmetic and pointer+count interfaces.
- Prefer RAII owners; reserve `gsl::owner<T*>` for legacy/ABI edges.
- Use `Expects`/`Ensures` for pre/postconditions that types cannot express.
- Use `finally` and narrowing helpers to make cleanup and conversions explicit.
- Integrate static analysis; suppress only with precise scope and justification.

Chapter 117

span (GSL)

117.1 What `gsl::span` is

117.1.1 Definition

`gsl::span<T>` is a lightweight, non-owning **view** over a **contiguous** sequence of `T`. Conceptually, it is a pair:

- a pointer to the first element, and
- a length (number of elements).

A span does not allocate, free, copy, or own the elements it references. It only *describes* a range.

117.1.2 Why the C++ Core Guidelines want it

The Core Guidelines emphasize bounds safety: avoid out-of-bounds access by making ranges explicit. Raw pointers do not carry bounds, so they are easily misused as implicit ranges

(especially with pointer arithmetic and mismatched pointer+length parameters). The Guidelines therefore recommend `span` as a primary vocabulary type for passing contiguous sequences safely.

117.2 Relationship to `std::span`

117.2.1 Two spans, one idea

`std::span` is the standard library vocabulary type (C++20) with the same fundamental idea: a non-owning view of contiguous memory. The GSL `span` predates standardization and heavily influenced `std::span`.

117.2.2 A key practical difference: bounds checking

In typical implementations and in the Guidelines/tooling ecosystem, `gsl::span` is used as the **bounds-checked** span:

- it is designed to detect invalid indexing early (fail-fast in contract violations),
- it is often treated specially by static analysis rules that enforce the Bounds profile.

For example, Microsoft's C++ Core Guidelines checking includes a rule that warns on unchecked `operator[]` use and treats indexing through `gsl::span` differently than indexing through other containers. This is aligned with the Bounds profile intent: reduce out-of-bounds errors.

117.2.3 Choosing between them in real projects

- Prefer `std::span` when you want a fully standard, portable vocabulary type and you manage bounds via design, validation, and testing.

- Prefer `gsl::span` when you want the Guidelines-oriented, bounds-safety posture (especially in code bases adopting the Core Guidelines profiles with tooling).

117.3 Construction: how to obtain a `gsl::span`

117.3.1 From pointer + count

This is the most direct way to wrap a legacy pointer+length pair.

```
#include <gsl/span>
#include <cstddef>

void consume(gsl::span<int> s);

void f(int* p, std::size_t n)
{
    consume(gsl::span<int>(p, n));
}
```

117.3.2 From C arrays and `std::array`

A span can be formed from fixed-size arrays without losing the bound.

```
#include <gsl/span>
#include <array>

void use(gsl::span<const int> s);

void f()
{
    int a[4] = {1,2,3,4};
    std::array<int, 3> b = {5,6,7};
```

```
    use(a);
    use(b);
}
```

117.3.3 From contiguous containers

Many code bases create spans from contiguous containers (notably `std::vector` and `std::string`) by using their `data()` and `size()`.

```
#include <gsl/span>
#include <vector>

void read_only(gsl::span<const int> s);

void f()
{
    std::vector<int> v = {10, 20, 30};
    read_only(gsl::span<const int>(v.data(), v.size()));
}
```

117.4 Element access and bounds safety

117.4.1 The problem: unchecked indexing is a common bug source

Out-of-bounds access through indexing is a frequent cause of undefined behavior, memory corruption, and security vulnerabilities. Safe indexing requires that the index is always within `[0, size)`.

117.4.2 GSL approach: contract-checked access

A core goal of the GSL implementation is to make span usage safer by providing **bounds checking** so violations can be detected immediately during testing and in hardened builds.

117.4.3 Recommended checked access: `gsl::at`

The MSVC Core Guidelines checker explicitly recommends `gsl::at()` as a checked alternative to unchecked subscripting, aligning with Bounds.4 (avoid unchecked access).

```
#include <gsl/gsl>
#include <vector>

int read_checked(const std::vector<int>& v, int i)
{
    return gsl::at(v, i); // checked access
}
```

117.4.4 Safe indexing pattern: validate once, then work

Even with checked tools, it is good practice to validate boundary conditions once and then operate under a clear invariant.

```
#include <gsl/span>
#include <cstddef>

int first_or_zero(gsl::span<const int> s)
{
    if (s.empty()) return 0;
    return s[0]; // safe because we validated non-emptiness
}
```

117.5 Slicing and subranges

117.5.1 Why slicing matters

A large fraction of bounds bugs come from manual offset calculations and pointer arithmetic. Span slicing expresses intent directly and localizes range computations.

117.5.2 `first`, `last`, and `subspan`

```
#include <gsl/span>
#include <cstddef>

int checksum(gsl::span<const unsigned char> bytes)
{
    int s = 0;
    for (auto b : bytes) s += b;
    return s;
}

int parse_packet(gsl::span<const unsigned char> packet)
{
    if (packet.size() < 8) return -1;

    auto header = packet.first(8);
    auto body   = packet.subspan(8);

    return checksum(header) + checksum(body);
}
```

117.6 Byte views and object representation

117.6.1 Viewing memory as bytes safely

Many low-level tasks (hashing, checksums, serialization boundaries) need to view objects as bytes. Span-based byte views help keep pointer+size together and reduce indexing mistakes.

```
#include <gsl/span>
#include <cstddef>
#include <cstdint>

std::uint32_t crc32(gsl::span<const std::byte> bytes);

std::uint32_t crc_of_words(gsl::span<const std::uint32_t> words)
{
    auto p = reinterpret_cast<const std::byte*>(words.data());
    auto n = words.size_bytes();
    return crc32(gsl::span<const std::byte>(p, n));
}
```

Important: this is about object representation, not arbitrary type-punning. Keep such conversions localized and well-documented.

117.7 Designing interfaces with `gsl::span`

117.7.1 Replace pointer+length parameters

A fragile interface:

```
void process_bad(const int* p, std::size_t n);
```

A safer interface:

```
#include <gsl/span>

void process(gsl::span<const int> s);
```

Benefits:

- the bound travels with the pointer,
- the signature documents intent (a range),
- it supports the Bounds profile rules (especially reducing pointer arithmetic and decay-style mistakes).

117.7.2 Const-correctness

- Use `gsl::span<const T>` for read-only input.
- Use `gsl::span<T>` when the function modifies elements.

```
#include <gsl/span>

int sum(gsl::span<const int> s)
{
    int r = 0;
    for (int x : s) r += x;
    return r;
}

void scale(gsl::span<int> s, int k)
{
    for (int& x : s) x *= k;
}
```

117.8 Lifetime rules: bounds safety is not lifetime safety

117.8.1 A span never extends lifetime

`gsl::span` does not own data. If the underlying storage becomes invalid, the span becomes dangling.
Common dangling cases:

- returning a span to a local array,
- storing a span to a `std::vector` and then resizing the vector (reallocation),
- building a span from a temporary container.

```
#include <gsl/span>
#include <vector>

gsl::span<int> bad()
{
    std::vector<int> v = {1,2,3};
    return gsl::span<int>(v.data(), v.size()); // BAD: v is destroyed on return
}
```

Bounds safety prevents out-of-range access *assuming the data is valid*; lifetime safety prevents dangling references. The Core Guidelines treat these as complementary profiles.

117.9 Tooling: why `gsl::span` fits the Guidelines ecosystem

117.9.1 Bounds profile enforcement

Static analysis tools that implement the C++ Core Guidelines profiles explicitly encourage span-based code:

- MSVC's C++ Core Guidelines checkers include Bounds rules such as *Bounds.1: Don't use pointer arithmetic. Use span instead.*

- MSVC also provides a bounds-checking-related rule that recommends `gsl::at()` instead of unchecked subscripting.

117.9.2 Practical consequence

When a team adopts the Bounds profile, using `gsl::span` in interfaces and using checked access patterns (`gsl::at`, validated indexing, slicing) typically reduces warnings and improves safety consistency across the code base.

117.10 Performance notes

117.10.1 Span is lightweight

A span is small (pointer + size for dynamic extent) and designed to be passed by value. The safety gains often come from better structure:

- fewer pointer+length mismatches,
- fewer manual offset computations,
- earlier detection of invalid indexing.

117.10.2 Checks vs structure

In hot paths, do not rely on repeated per-element checks as your main design. Prefer:

- validating preconditions once,
- iterating safely using range-for or algorithms,
- slicing into subspans to keep offsets correct.

117.11 Summary

`gsl::span` is the Core Guidelines vocabulary type for representing a contiguous range **without owning it**. It is central to Bounds Safety because it keeps pointer and size together and supports safer patterns than raw pointer arithmetic. Use `gsl::span` to:

- replace pointer+length interfaces,
- express intent (a range, not a single object pointer),
- reduce out-of-bounds risk via slicing and checked access patterns,
- integrate cleanly with Core Guidelines static analysis (Bounds rules).

Chapter 118

not_null (GSL)

118.1 What gsl::not_null is

118.1.1 Definition

gsl::not_null<T> is a small wrapper type that represents a **pointer-like value that must never be null**. It is a **vocabulary type**: its main purpose is to express an invariant in the type system and make interfaces self-documenting.
In the C++ Core Guidelines GSL section, not_null<T> is presented as the way to say *"this value is not supposed to be* nullptr*"*. The type T is usually a pointer type (for example int*), but it may be any type for which comparing to nullptr is meaningful.

118.1.2 What not_null does and does not do

What it guarantees:

- A not_null object is constructed only from a non-null value.
- The program fails fast (contract violation) if construction is attempted with a null value.

- The non-null assumption becomes visible at the API boundary (parameters, return types, data members).

What it does not guarantee:

- It does not prove the pointer is *valid* (e.g., not dangling, not pointing to freed memory).
- It does not provide ownership; it only constrains nullness.
- It does not automatically solve lifetime problems; it is complementary to lifetime safety rules.

118.2 Why `not_null` exists

118.2.1 The real problem: "nullable by default"

In many code bases, raw pointers are used for multiple meanings:

- optional value (`nullptr` means "absent"),
- mandatory non-owning reference (`nullptr` is a bug),
- ownership (`delete` required),
- array/range pointer.

When a pointer is *not* supposed to be null, leaving it as `T*` hides that assumption. Developers then add repeated checks, comments, or assertions, which are:

- easy to forget,
- easy to remove accidentally,
- not visible at the interface.

`gsl::not_null` solves the interface problem by moving the assumption into the type.

118.2.2 Tooling benefit (Core Guidelines checkers)

Static analysis tools that implement Core Guidelines checks (for example, MSVC C++ Core Guidelines checking) can:

- recommend marking a pointer as `not_null` when it is never checked against null,
- warn if code redundantly checks a `not_null` value for null,
- improve reasoning about control-flow and contracts.

118.3 Core semantics and operations

118.3.1 Construction and enforcement

A `not_null` object is created from a value of type `T` that must not be null. The Microsoft GSL documentation notes that the non-null checks are performed in the **constructor** and that there is **no overhead when retrieving or dereferencing** the stored pointer-like value.

118.3.2 Access

A `not_null<T>` typically provides:

- implicit conversion to `T`,
- `get()` to access the underlying `T`,
- `operator*` and `operator->` where meaningful.

118.3.3 Assignment and rebinding

Unlike references, a `not_null<T>` object can be *rebound* (assigned a different non-null value) unless you make the wrapper itself `const`. This is a key reason it is not equivalent to using a reference.

118.4 When to use `not_null`

118.4.1 Best use cases

Use `gsl::not_null` when `nullptr` is not a meaningful value and should be treated as a bug:

- function parameters that require a valid object pointer,
- return values that must never be null,
- data members that must always refer to something once constructed,
- pointers stored for later use where you want to prevent accidental null assignment.

118.4.2 When not to use it

Do not use `not_null` when null is a normal, meaningful state:

- optional relationships (`nullptr` means “not present”),
- sentinel logic where null is a legitimate value,
- APIs that must accept null for interoperability (prefer wrapping inside the boundary).

118.5 Comparing `not_null` with alternatives

118.5.1 Why not just use references?

A reference `T&` expresses:

- non-null (cannot be null),
- and **non-rebindable** (cannot later refer to a different object).

`not_null<T*>` expresses:

- non-null,
- and can be rebindable (unless `const`).

Therefore, `T&` and `not_null<T*>` are not interchangeable. In particular, references cannot represent “a non-null pointer that may later be reset to point elsewhere”.

118.5.2 Why not just use assertions?

Assertions can enforce a check at one point in the code, but they do not:

- express the rule in the type of the parameter/return,
- prevent misuse by callers,
- support tool reasoning as strongly as a vocabulary type.

118.5.3 What about smart pointers?

`not_null` is designed to wrap *pointer-like* types where comparison to `nullptr` is meaningful. That can include raw pointers and (in many real code bases) smart pointers that support null states. This can be useful for APIs that require, for example, a non-null `std::shared_ptr<T>`.

118.6 Practical usage patterns

118.6.1 Non-null parameter

```
#include <gsl/gsl>
#include <string>

struct Logger {
    void write(const std::string& s);
};

void log_line(gsl::not_null<Logger*> lg, const std::string& msg)
{
    lg->write(msg); // no need for null checks
}
```

118.6.2 Non-null return value

```
#include <gsl/gsl>

struct Node { int value; };

gsl::not_null<Node*> find_root(gsl::not_null<Node*> start)
{
    // example only; in real code you would follow actual parent pointers
    return start;
}
```

118.6.3 Data member that must never be null

```
#include <gsl/gsl>
```

```
struct Engine { void run(); };

class Controller {
public:
    explicit Controller(gsl::not_null<Engine*> e) : eng_(e) {}

    void step() { eng_->run(); }

private:
    gsl::not_null<Engine*> eng_;
};
```

118.6.4 Rebinding control with `const`

```
#include <gsl/gsl>

struct Widget { int id; };

void example(gsl::not_null<Widget*> w)
{
    // w may be rebound to another non-null Widget* (if assigned)
    (void)w;

    const gsl::not_null<Widget*> fixed = w;
    // fixed cannot be rebound after construction
    (void)fixed;
}
```

118.7 Common pitfalls and how to avoid them

118.7.1 Mistaking non-null for valid

`not_null` does not prevent dangling pointers. A pointer can be non-null and still invalid (freed object, moved-from storage, lifetime ended). Use lifetime-safe design (RAII owners, spans for ranges, avoiding returning pointers to locals) alongside `not_null`.

118.7.2 Leaking nullable semantics into non-null APIs

Do not "force" an optional pointer into `not_null` to satisfy an interface. If a value can be absent, use a representation that expresses that:

- keep it as a nullable pointer at the boundary and validate,
- or use an explicit optional type where appropriate.

118.7.3 Overusing `not_null` in low-level glue

When interfacing with legacy C APIs that accept null for special meanings, keep that logic localized. Convert to `not_null` only after you establish the invariant.

118.8 Integration with Core Guidelines tools (MSVC examples)

MSVC Core Guidelines checking provides diagnostics that encourage correct `not_null` usage patterns, including:

- recommending `gsl::not_null` when a pointer is never tested for null,
- warning when an expression is of type `gsl::not_null` and is redundantly tested against null.

```
#include <gsl/gsl>

struct X { int n; };

int use(gsl::not_null<X*> p)
{
    // Under Core Guidelines checking, this kind of test is redundant:
    // if (p == nullptr) return 0;

    return p->n;
}
```

The intent is clear: once a value is `not_null`, your code should treat nullness as impossible, and checks become noise.

118.9 Performance and build-mode considerations

118.9.1 Cost model

`not_null` is designed to be essentially a zero-cost wrapper in normal use:

- a construction-time check enforces the invariant,
- the stored pointer-like value is then accessed and dereferenced without extra checking overhead.

118.9.2 Engineering guidance

- Use `not_null` primarily to improve correctness and clarity.
- In performance-critical code, focus on using it at *interfaces* and important invariants; do not micro-optimize away correctness.

118.10 Summary

`gsl::not_null<T>` is the Core Guidelines vocabulary type for expressing **non-nullness** in C++ interfaces. Use it when null is a bug, not a state:

- it documents intent directly in the type,
- it enforces the rule at construction (fail-fast on violation),
- it improves static analysis and reduces redundant null checks.

Remember that it is not a lifetime or ownership tool; it must be used together with proper resource management and lifetime-safe design.

Chapter 119

`owner` (GSL)

119.1 What `gsl::owner` is

119.1.1 Definition

`gsl::owner<T>` is a **vocabulary annotation** used by the ISO C++ Core Guidelines to mark **owning** raw pointers in code that must still use raw pointers. In practice, it is most commonly used as:

```
gsl::owner<T*>
```

meaning: *this* `T*` *owns the object it points to, and the owner is responsible for releasing it*. The Core Guidelines explicitly describe using `owner<X*>` to make ownership transfer clear when returning raw pointers is unavoidable, and they emphasize that such pointers must be `deleted` or transferred to another owner.

119.1.2 Key point: `owner` is not RAII

`gsl::owner<T*>` is **not** a smart pointer and does not automatically delete anything. It is still a raw pointer that **requires explicit release** (for example, `delete` for memory allocated with `new`) or explicit transfer to another owner.

119.1.3 Nullability

The Core Guidelines explicitly note that `owner<T*>` **might be** `nullptr`. Owning pointers may be nullable; if an owning pointer must never be null, combine the concepts using a non-null owner type in your own design policy, or avoid nullable ownership entirely by using proper owners (recommended).

119.2 Why `owner` exists

119.2.1 The Guidelines default: raw pointers are non-owning

A central assumption in the Core Guidelines is:

- `T*` is **non-owning** by default (a pointer to an object, not a resource handle).

When you must use a raw pointer that *does* own, you should say so explicitly using `owner<T*>`.

119.2.2 The preferred alternative: use RAII owners

The Guidelines recommend using resource handles such as:

- `std::unique_ptr` for exclusive ownership,
- `std::shared_ptr` for shared ownership,
- standard containers (e.g., `std::vector`) for owning sequences.

`owner<T*>` is primarily for cases where you cannot adopt a proper RAII owner yet.

119.2.3 When `owner<T*>` is acceptable

The Core Guidelines list typical reasons you may need `owner<T*>` instead of upgrading to a proper resource handle:

- cost of conversion in large legacy code,
- ABI constraints (interfaces fixed to raw pointers),
- the pointer is part of the internal implementation of a resource handle.

In these cases, `owner<T*>` makes ownership visible and tool-checkable.

119.3 Core semantics: what `owner<T*>` means

119.3.1 Ownership responsibility

If you hold an `owner<T*>`, then you are responsible for one of the following:

- releasing the resource (e.g., `delete p;`) exactly once, or
- transferring ownership to another `owner<T*>` in a way that preserves the single-release rule.

This is exactly the meaning the Core Guidelines attach to the annotation: it tells both readers and tools that the pointer must be released or moved to another owner.

119.3.2 Transfer, not sharing

An `owner<T*>` is best understood as *exclusive ownership by convention* (one owner responsible for release). If you want shared ownership semantics, you should use `std::shared_ptr<T>` or another explicit shared ownership mechanism.

119.3.3 What `owner` does not provide

- It does not automatically delete.
- It does not prevent leaks by itself.
- It does not prevent double delete by itself.
- It does not guarantee lifetime safety for non-owners that observe the pointer.

It is primarily a **semantic marker** for analysis tools and human readers.

119.4 Recommended usage patterns

119.4.1 Pattern 1: Mark ownership transfer at an interface boundary

```
#include <gsl/gsl>

struct X { int v; };

gsl::owner<X*> make_x()
{
    gsl::owner<X*> p = new X{42};
    return p; // ownership transferred to caller
}

void consume_x(gsl::owner<X*> p)
{
    // ... use *p ...
    delete p; // explicit release required
}
```

This is the style explicitly shown by the Core Guidelines for code that must return a raw pointer but wants ownership transfer to be unambiguous.

119.4.2 Pattern 2: Use a real owner internally; expose `owner<T*>` only when forced

Prefer to keep RAII internally and only annotate raw-pointer ownership at a legacy boundary.

```
#include <gsl/gsl>
#include <memory>

struct X { int v; };

gsl::owner<X*> legacy_api_returning_owner()
{
    // Prefer: keep RAII as long as possible
    std::unique_ptr<X> u = std::make_unique<X>(X{7});

    // Forced boundary: release ownership to raw pointer
    return u.release(); // now the caller must delete
}
```

Rule of thumb: If you can avoid returning ownership as a raw pointer, do so. If you cannot, use `owner<T*>` to make the contract explicit.

119.4.3 Pattern 3: Owning pointers inside resource-handle implementations

The Guidelines explicitly mention `owner` as useful inside the implementation of resource handles.

```
#include <gsl/gsl>
#include <utility>

class XHandle {
```

```
public:
    explicit XHandle(gsl::owner<int*> p) noexcept : p_(p) {}
    ~XHandle() { delete p_; }

    XHandle(const XHandle&) = delete;
    XHandle& operator=(const XHandle&) = delete;

    XHandle(XHandle&& other) noexcept : p_(std::exchange(other.p_, nullptr)) {}
    XHandle& operator=(XHandle&& other) noexcept
    {
        if (this != &other) {
            delete p_;
            p_ = std::exchange(other.p_, nullptr);
        }
        return *this;
    }

private:
    gsl::owner<int*> p_ = nullptr; // owning raw pointer inside a handle
};
```

119.5 Tooling: how analyzers interpret `owner`

119.5.1 MSVC C++ Core Guidelines checker: ownership diagnostics

Microsoft's Core Guidelines checkers include a group of rules that treat `owner<T*>` specially:

- **C26400** warns when the result of `operator new` (or a function returning `owner<T>`) is assigned to a plain raw pointer, because ownership is being hidden.
- **C26401** warns when code deletes a raw pointer that is *not* an `owner<T*>`, because non-owner raw pointers should not be deleted.

- **C26406** warns when assigning a raw pointer into an `owner<T*>`, because raw pointers do not guarantee a safe ownership transfer (the original owner may still exist).

These diagnostics reflect the Guidelines intent: make ownership explicit and make ownership transfer mechanically checkable.

119.5.2 clang-tidy: ownership semantics check

clang-tidy provides `cppcoreguidelines-owning-memory`, which implements the type-based semantics of `gsl::owner<T*>` to support static analysis in code bases that still use raw pointers for resources.

119.6 Common mistakes and how to avoid them

119.6.1 Mistake 1: Thinking `owner` makes a raw pointer safe

`owner<T*>` does not add automatic deletion, does not prevent leaks, and does not prevent double-free. It only marks intent. The code still must obey correct release/transfer discipline.

119.6.2 Mistake 2: Assigning raw pointers into owners

This is exactly what MSVC warning C26406 flags: raw pointers do not guarantee ownership transfer, so converting a raw pointer to an owner without a proven transfer point can create double-delete or leak risk.

```
#include <gsl/gsl>

gsl::owner<int*> bad(int* p) // p may already be owned elsewhere
{
    return p; // risky: ownership not guaranteed
}
```

119.6.3 Mistake 3: Deleting non-owners

If a pointer is not an `owner<T*>`, the Core Guidelines model treats it as non-owning, and tools can warn when such a pointer is deleted.

```
void f(int* p) // by default, non-owning
{
    // delete p; // should be rejected by guideline-based tooling
}
```

119.6.4 Mistake 4: Using `owner` instead of upgrading to RAII

`owner<T*>` is a transitional tool for legacy constraints. If you can replace owning raw pointers with `std::unique_ptr` (or another handle), you should.

119.7 Practical guidance for real projects

119.7.1 Adoption strategy

- Treat plain `T*` as non-owning by default.
- Introduce `owner<T*>` only at the boundaries where ownership is truly represented by raw pointers.
- Prefer to refactor owning raw pointers into resource handles (`unique_ptr`, containers) as soon as feasible.
- Enable static analysis rules that understand `owner<T*>` so mistakes become warnings instead of latent bugs.

119.7.2 Interoperability note

When an ABI forces a raw-pointer ownership convention (common in C APIs), `owner<T*>` is a useful way to annotate that convention in C++ code so that humans and tools can see it immediately.

119.8 Summary

`gsl::owner<T*>` is the Core Guidelines annotation for **owning raw pointers**. Use it only when you cannot use a proper RAII owner. It makes ownership transfer and responsibility explicit and enables static analysis tools (MSVC and clang-tidy) to enforce key rules:

- do not hide ownership by assigning allocations to plain raw pointers,
- do not delete non-owners,
- do not assign raw pointers into owners without a guaranteed ownership transfer.

For new code, prefer real resource handles; for legacy constraints, use `owner<T*>` to make dangerous ownership explicit and checkable.

Chapter 120

`index` (GSL)

120.1 What `gsl::index` is

120.1.1 Definition

`gsl::index` is the Guidelines Support Library (GSL) **vocabulary type for indexing**. In Microsoft's reference implementation of the GSL, it is defined as:

```
using index = std::ptrdiff_t;
```

The intent is to provide one consistent, well-named type to use for:

- container indices (subscripts),
- loop counters used to index,
- sizes that must interact safely with indices (especially when negative values are meaningful).

120.1.2 What it is not

`gsl::index` is **not** a replacement for every size type in the standard library. Standard containers typically use `size_t` for `size()` and `operator[]` parameters. `gsl::index` is a practical GSL convention for writing *indexing code* more safely and consistently.

120.2 Why the Guidelines use a signed index type

120.2.1 The core problem: signed/unsigned traps

A large number of indexing bugs come from mixing:

- **signed** integers (like `int`) for indices, and
- **unsigned** sizes (like `size_t`) from `size()`.

Common failure modes:

- A negative index converted to `size_t` becomes a very large value and can pass naive checks.
- Comparisons like `i < v.size()` cause signed/unsigned conversion warnings and can hide logic errors.
- Reverse loops with unsigned indices easily underflow and become infinite loops.

120.2.2 Why `ptrdiff_t`?

`std::ptrdiff_t` is the standard signed integer type intended to represent pointer differences and distances between elements in memory. Using it for indexing has two practical benefits:

- it can represent negative sentinel values (for example, “not found” as `-1`),
- it reduces accidental unsigned underflow patterns in reverse loops and arithmetic.

120.3 When to use `gsl::index`

120.3.1 Use it for indexing loops

Use `gsl::index` as the loop counter when the counter is used to subscript a sequence.

```
#include <gsl/util>
#include <vector>

int sum_indexed(const std::vector<int>& v)
{
    int s = 0;
    for (gsl::index i = 0; i < static_cast<gsl::index>(v.size()); ++i) {
        s += v[static_cast<std::size_t>(i)];
    }
    return s;
}
```

120.3.2 Use it for “not found” and similar sentinel results

Returning `-1` is a common, compact pattern for “not found”. A signed index type supports that naturally.

```
#include <gsl/util>
#include <vector>

gsl::index find_index(const std::vector<int>& v, int value)
{
    for (gsl::index i = 0; i < static_cast<gsl::index>(v.size()); ++i) {
        if (v[static_cast<std::size_t>(i)] == value) return i;
    }
    return -1; // not found
}
```

120.3.3 Use it for reverse indexing loops

Reverse loops are a classic source of unsigned underflow bugs. A signed index type makes a correct reverse loop straightforward.

```
#include <gsl/util>
#include <vector>

int last_nonzero(const std::vector<int>& v)
{
    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;
}
```

Important: This loop must handle the empty-vector case correctly. A safe pattern is to check emptiness first.

```
#include <gsl/util>
#include <vector>

int last_nonzero_safe(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;
}
```

120.4 How to use `gsl::index` safely with standard containers

120.4.1 Dealing with `size()` returning `size_t`

Most standard containers return `size_t` from `size()`. If you use `gsl::index` for indexing, you will often want a single explicit conversion at the boundary:

```
#include <gsl/util>
#include <vector>

void f(const std::vector<int>& v)
{
    const gsl::index n = static_cast<gsl::index>(v.size()); // one explicit conversion

    for (gsl::index i = 0; i < n; ++i) {
        (void)v[static_cast<std::size_t>(i)];
    }
}
```

120.4.2 Prefer expressing the algorithm without indexing

Even when using `gsl::index`, the safest indexing is often the one you do not write. Prefer range-for and algorithms when possible:

```
#include <vector>

int sum_range_for(const std::vector<int>& v)
{
    int s = 0;
    for (int x : v) s += x;
    return s;
}
```

120.4.3 When a checked access fits better

If the index is influenced by external input or complex logic, consider checked access such as `at()` (or `gsl::at()` in Guidelines-oriented code) to avoid undefined behavior on mistakes.

120.5 Common mistakes and how to avoid them

120.5.1 Mistake 1: converting a negative index to `size_t`

If you accept user indices as signed values, validate negativity before converting.

```
#include <vector>
#include <cstddef>

int read_user_index(const std::vector<int>& v, int i)
{
    if (i < 0) return 0;
    const std::size_t ui = static_cast<std::size_t>(i);
    if (ui >= v.size()) return 0;
    return v[ui];
}
```

120.5.2 Mistake 2: using `gsl::index` for sizes that cannot be negative

`gsl::index` is mainly for *indexing* operations. For purely non-negative sizes, it is still valid to use the standard library's `size_t`. Use `gsl::index` when it makes the indexing logic safer and clearer.

120.5.3 Mistake 3: assuming `gsl::index` is always wide enough

`gsl::index` is typically the same width as a pointer difference (`ptrdiff_t`). On mainstream 64-bit platforms it is usually 64-bit, but it is not guaranteed to hold every possible `size_t` value on every platform. If a container can conceptually exceed `ptrdiff_t`'s max value on your platform, you must design accordingly (usually by avoiding such enormous in-memory containers, or using a larger integral type where the platform provides it).

120.6 How `gsl::index` fits with modern standard C++

120.6.1 `std::ssize` as a standard alternative for signed sizes

Modern C++ provides `std::ssize` (C++20), which returns a signed integer type corresponding to a container's size. In code bases that prefer standard facilities, `std::ssize` is often used for loop counters to avoid signed/unsigned mismatch.

Practical guideline:

- If your project is strongly Guidelines/GSL-based: use `gsl::index` consistently.
- If your project prefers standard-only vocabulary: use `std::ssize` for signed sizing and choose an index type carefully.

120.7 Summary

`gsl::index` is the GSL vocabulary type for indexing, defined (in Microsoft GSL) as `std::ptrdiff_t`. It exists to make indexing code safer and clearer by:

- encouraging a **signed** index type,
- reducing signed/unsigned mismatch pitfalls,

- supporting negative sentinel values such as `-1`,
- making reverse loops and index arithmetic less error-prone.

Use `gsl::index` for index variables and index-returning functions, keep conversions from `size_t` explicit and localized, and prefer non-indexing algorithms when possible.

Chapter 121

`gsl::Expects` / `gsl::Ensures`

121.1 Purpose and intent

121.1.1 What these facilities are

`gsl::Expects` and `gsl::Ensures` are the GSL assertion facilities used by the ISO C++ Core Guidelines to express:

- **preconditions** (requirements that must be true before a function continues), and
- **postconditions** (guarantees that must be true before a function returns normally).

In the Core Guidelines GSL specification, `Expects(p)` is described as a precondition assertion that *terminates the program unless* `p == true`. The same section lists `Ensures` as a postcondition assertion, and notes that both are currently macros placed in function bodies pending standard language support for contracts.

121.1.2 What they are not

These contract checks are **not** a general-purpose error-handling mechanism. They are intended for **programmer errors** (violations of an interface contract), not for routine runtime failures such as:

- file-not-found,
- network timeouts,
- invalid user input that must be reported and handled.

If a condition can reasonably be violated in normal operation and must be handled, use regular error handling (return values, error objects, exceptions, etc.) instead of contract termination.

121.2 Semantics and failure behavior

121.2.1 Fail-fast contract checking

The Guidelines describe `Expects` as terminating the program when violated. This is a deliberate **fail-fast** design: continuing after a broken precondition is often unsafe because the function may rely on that condition for memory safety and correctness.
The Guidelines discussion clarifies that "terminate" is intended to mean `std::terminate`, allowing a `terminate_handler` to perform last-chance logging or diagnostics.

121.2.2 Microsoft GSL implementation: termination is the standard behavior

Microsoft's GSL project is the reference implementation most commonly used with the Core Guidelines. In the GSL 3.0.0 release notes, Microsoft states that **contract violations are no**

longer configurable and that they **always result in termination** (rather than throwing or being ignored).

121.2.3 Why termination (and not exceptions)

A contract violation indicates that the program has reached a state where assumptions required for safe execution no longer hold. Attempting to recover by throwing can be misleading because it suggests the caller can continue safely after violating a contract. The fail-fast approach aims to prevent silent corruption and security issues by stopping immediately.

121.3 How to use `Expects` correctly

121.3.1 Place preconditions at the top of the function body

Put `Expects` checks early, before dereferencing pointers, indexing, or performing operations that rely on the precondition.

```
#include <gsl/assert>
#include <gsl/span>

int sum(gsl::span<const int> a)
{
    Expects(a.size() >= 0);      // trivial here, shown as structure
    Expects(!a.empty());         // requires at least one element

    int s = 0;
    for (std::size_t i = 0; i < a.size(); ++i) {
        s += a[i];
    }
    return s;
}
```

121.3.2 Prefer expressing contracts using safer types

Often you can remove whole classes of preconditions by changing the interface:

- Use `gsl::not_null<T*>` when null is forbidden.
- Use `gsl::span<T>` to bind a pointer with its bounds.
- Use strong types or enums to restrict invalid states.

```
#include <gsl/assert>
#include <gsl/pointers>   // not_null

int read_first(gsl::not_null<const int*> p)
{
    // No need for Expects(p != nullptr); the type expresses it.
    return *p;
}
```

121.3.3 Keep contracts simple, cheap, and side-effect free

A contract expression should be:

- **free of side effects** (do not call functions that mutate state),
- **fast** (do not perform heavy computations),
- **deterministic** and easy to understand.

121.4 How to use `Ensures` correctly

121.4.1 Use postconditions to document and verify guarantees

`Ensures` expresses what the function guarantees on normal return.

```
#include <gsl/assert>
#include <vector>

std::vector<int> make_nonempty(std::size_t n)
{
    Expects(n > 0);

    std::vector<int> v(n, 0);

    Ensures(!v.empty());
    Ensures(v.size() == n);
    return v;
}
```

121.4.2 Avoid postconditions that are expensive or redundant

If the postcondition is trivially guaranteed by construction and adds no value, omit it. Use `Ensures` when it improves correctness, documents subtle behavior, or guards against future maintenance errors.

121.5 Relationship to standard C++ contracts

121.5.1 Why `Expects/Ensures` are macros today

The Core Guidelines explicitly state that these assertions are currently macros and are intended to move toward standard contract syntax once available. The Guidelines mention a direction where `Expects(p)` could become a contract attribute form such as `[[expects: p]]` when the language supports it.

121.5.2 Practical takeaway

Today:

- Put `Expects` / `Ensures` in function bodies.
- Treat violations as **fatal contract breaks**.

If future standardized contracts are adopted widely, the same logical contracts can migrate from bodies to declarations, while keeping the same intent.

121.6 Debugging and diagnostics strategies

121.6.1 Use a terminate handler for last-chance logging

Because contract violations terminate, debugging often relies on:

- a debugger break on `std::terminate`,
- logging inside a custom `std::terminate_handler`,
- crash dumps / core dumps in production environments.

Microsoft explicitly suggests using a terminate handler to log relevant information before termination executes (especially after removal of throwing behavior).

```
#include <gsl/assert>
#include <exception>
#include <iostream>
#include <cstdlib>

static void my_terminate() noexcept
{
```

```
    // Keep it minimal and async-signal-safe in real systems if possible.
    std::cerr << "Fatal: contract violation or unhandled exception\n";
    std::abort(); // end quickly after logging
}

int main()
{
    std::set_terminate(my_terminate);

    int x = 0;
    Expects(x != 0); // will terminate
}
```

121.6.2 Do not use `Expects/Ensures` to validate user input

If the program must report an error and continue, use explicit validation and return an error result:

```
#include <optional>
#include <string>

std::optional<int> parse_positive_int(const std::string& s)
{
    // user input: validate and return a result, do not terminate
    try {
        int v = std::stoi(s);
        if (v <= 0) return std::nullopt;
        return v;
    } catch (...) {
        return std::nullopt;
    }
}
```

121.7 Common patterns

121.7.1 Indexing and bounds assumptions

When using span, bounds are part of the interface. Still, when an index comes from complex logic, a precondition can make the assumption explicit:

```
#include <gsl/assert>
#include <gsl/span>

int get_at(gsl::span<const int> a, std::size_t i)
{
    Expects(i < a.size());
    return a[i];
}
```

121.7.2 Non-null assumptions

Instead of:

```
#include <gsl/assert>

int f(const int* p)
{
    Expects(p != nullptr);
    return *p;
}
```

Prefer an interface that encodes the contract:

```
#include <gsl/pointers>

int f(gsl::not_null<const int*> p)
```

```
{
    return *p;
}
```

121.8 Common mistakes and how to avoid them

121.8.1 Mistake: using contracts for recoverable conditions

If failure is expected sometimes and the caller must react, do not use `Expects/Ensures`. Use a return-based or exception-based error model instead.

121.8.2 Mistake: writing contracts with side effects

Avoid:

```
Expects(++i < n); // side effect: changes program state
```

Write:

```
Expects(i + 1 < n); // no side effects
```

121.8.3 Mistake: relying on contracts for program logic

Contracts are for stating requirements and guarantees, not for implementing normal branches. Do not write code whose correctness depends on whether contracts are enabled/disabled in some configuration. Modern guidance (including Microsoft GSL's direction) is to treat contract violations as fatal termination.

121.9 Summary

`gsl::Expects` and `gsl::Ensures` are GSL contract assertion facilities used by the Core Guidelines to state preconditions and postconditions. A violated contract is treated as a **fatal**

error (termination), and Microsoft's GSL explicitly documents termination as the consistent behavior for contract violations. Use them to catch programmer mistakes early, and prefer expressing contracts through types (`not_null`, `span`) when possible.

Part XXI

Profiles

Chapter 122

Type Safety Profile (Pro.safety)

122.1 Overview

122.1.1 What a *profile* is

In the ISO C++ Core Guidelines, a *profile* is a coherent, tool-enforceable subset of rules designed to achieve a specific safety guarantee (for example, “static type safety”). The intent is gradual adoption: enable a profile in tools, fix diagnostics, and gain a clear, measurable improvement.

122.1.2 Goal of the Type Safety Profile

The Type Safety Profile makes it easier to construct code that:

- uses types correctly,
- avoids inadvertent type punning,

- avoids the major sources of type violations, especially unsafe casts and unsafe union usage.

For the purposes of this profile, **type-safety** means: a variable is not used in a way that violates the rules of its declared type; and memory accessed as type `T` should not actually hold an unrelated type `U`.

122.1.3 Completeness requires Bounds and Lifetime profiles too

The Core Guidelines explicitly note that complete safety is intended when Type safety is combined with:

- the Bounds Safety Profile, and
- the Lifetime Safety Profile.

Without bounds and lifetime guarantees, memory may be accessed beyond an object or after an object ends, undermining any type-based reasoning.

122.2 Profile summary (Type.1 – Type.8)

A conforming implementation of the profile recognizes the following patterns as non-conforming and emits diagnostics.

122.2.1 Type.1: Avoid casts

Avoid casts because they can silently subvert the type system and enable accidental type punning. The profile defines a strict set of cast bans/restrictions:

1. **Type.1.1** Don't use `reinterpret_cast`.
2. **Type.1.2** Don't use `static_cast` for arithmetic types.

3. **Type.1.3** Don't cast between pointer types where the source type and the target type are the same.

4. **Type.1.4** Don't cast between pointer types when the conversion could be implicit.

Example: `reinterpret_cast` is a red flag

```
#include <cstdint>

std::uint32_t read_u32_bad(const unsigned char* p) {
    // Strict-aliasing and alignment hazards; violates type safety.
    return *reinterpret_cast<const std::uint32_t*>(p);
}
```

Safer approach: copy bytes into a real object

```
#include <cstdint>
#include <cstring>

std::uint32_t read_u32_ok(const unsigned char* p) {
    std::uint32_t x{};
    std::memcpy(&x, p, sizeof(x)); // reads representation safely
    return x;
}
```

Tool enforcement (Clang-Tidy) Clang-Tidy provides checks that directly correspond to the Type profile cast restrictions, including: `cppcoreguidelines-pro-type-reinterpret-cast` (flags all `reinterpret_cast` use).

122.2.2 Type.2: Don't use `static_cast` to downcast

Downcasting with `static_cast` can produce undefined behavior if the dynamic object is not actually of the derived type. The profile bans this pattern and directs you to use `dynamic_cast`

when hierarchy navigation is unavoidable.

Bad: unchecked downcast

```
struct Base { virtual ~Base() = default; };
struct Derived : Base { void f(); };

void g(Base& b) {
    auto& d = static_cast<Derived&>(b); // unsafe if b is not a Derived
    d.f();
}
```

Good: checked downcast (when you must navigate a hierarchy)

```
#include <stdexcept>

void g(Base& b) {
    if (auto* pd = dynamic_cast<Derived*>(&b)) {
        pd->f();
        return;
    }
    throw std::runtime_error("Bad downcast: Base is not a Derived");
}
```

Tool enforcement (Clang-Tidy) `cppcoreguidelines-pro-type-static-cast-downcast` flags `static_cast` downcasts and can suggest a fix-it to `dynamic_cast`.

122.2.3 Type.3: Don't use `const_cast` to cast away `const`

Casting away `const` is almost always a design error. If you then modify an object that was originally declared `const`, the behavior is undefined. The profile forbids casting away `const`.

Bad: removing constness

```
void h(const int* p) {
    int* q = const_cast<int*>(p); // breaks the contract
    *q = 42;                      // undefined if *p was originally const
}
```

Better: fix the interface

```
void h(int* p) {        // caller must provide mutable data
    *p = 42;
}
```

Tool enforcement (Clang-Tidy) `cppcoreguidelines-pro-type-const-cast` flags `const_cast` usage and is documented as part of Type.3.

122.2.4 Type.4: Don't use C-style or functional casts

C-style casts `(T)expr` and functional casts `T(expr)` can perform different kinds of conversions (including dangerous ones) and hide intent. The profile bans these forms and prefers:

- direct construction when you want a value of type `T`,
- or a named cast (`static_cast`, `dynamic_cast`, `const_cast`, `reinterpret_cast`) when you genuinely need one, and
- brace initialization `T{expr}` where appropriate.

Bad: ambiguous C-style cast

```
double d = 3.14;
int i = (int)d; // hides narrowing and intent
```

Better: explicit intent

```
double d = 3.14;
int i = static_cast<int>(d); // still a narrowing conversion, but explicit
```

Tool enforcement (Clang-Tidy) `cppcoreguidelines-pro-type-cstyle-cast` flags C-style casts that perform unsafe casts and documents that a C-style cast may attempt several conversions in order, which is exactly why it can obscure danger.

122.2.5 Type.5: Don't use a variable before it has been initialized

Reading an uninitialized object is undefined behavior for many types and is a frequent source of miscompilations, intermittent bugs, and security issues. The profile requires that variables are initialized before use.

Bad: uninitialized read

```
int f(bool cond) {
    int x;           // uninitialized
    if (cond) x = 1;
    return x;        // UB if cond is false
}
```

Good: initialize and then update

```
int f(bool cond) {
    int x = 0;
    if (cond) x = 1;
    return x;
}
```

Tool enforcement (Clang-Tidy) `cppcoreguidelines-init-variables` checks for local variables declared without an initial value and explicitly states it is part of the Type safety (Type.5) profile.

122.2.6 Type.6: Always initialize a data member

If a constructor fails to initialize members, objects can contain indeterminate values, causing undefined behavior when those values are read. The profile requires consistent member initialization, typically using:

- default member initializers, and/or
- constructor initializer lists, and/or
- constructors that establish class invariants.

Bad: member left uninitialized

```
struct S {
    int n;      // must be initialized
    S() {}      // leaves n indeterminate
};
```

Good: default member initializer

```
struct S {
    int n{0};
    S() = default;
};
```

Tool enforcement (Clang-Tidy) `cppcoreguidelines-pro-type-member-init` flags constructors that fail to initialize members that would otherwise be left in an undefined state, and documents that it is part of Type.6.

122.2.7 Type.7: Avoid naked `union`

A “naked union” relies on programmer discipline to track the active member. Incorrect reads/writes easily violate type rules and object lifetime rules. The profile bans unsafe union-member access and points you toward safer alternatives, such as `std::variant`.

Bad: fragile union usage

```
union U {
    int i;
    double d;
};

double bad() {
    U u;
    u.i = 1;
    return u.d; // reads inactive member: violates the type model
}
```

Good: tagged union with `std::variant`

```
#include <variant>

using V = std::variant<int, double>;

double ok(const V& v) {
    if (auto pd = std::get_if<double>(&v)) return *pd;
    return static_cast<double>(std::get<int>(v));
}
```

Tool enforcement (Clang-Tidy) `cppcoreguidelines-pro-type-union-access` flags accesses to union members and explains why such access is fragile.

122.2.8 Type.8: Avoid varargs

C-style varargs (`...` and `va_arg`) are not type-safe: the callee assumes the caller passed arguments of the correct types, and mismatches lead to undefined behavior. The profile bans varargs and encourages typed alternatives such as overloads, variadic templates, and type-safe formatting libraries.

Bad: C varargs

```
#include <cstdarg>

int sum(int n, ...) {
    va_list ap;
    va_start(ap, n);
    int s = 0;
    for (int i = 0; i < n; ++i) {
        s += va_arg(ap, int); // assumes caller passed int
    }
    va_end(ap);
    return s;
}
```

Good: variadic templates (type-safe)

```
template<class... Ints>
int sum(Ints... xs) {
    static_assert((std::is_same_v<Ints, int> && ...), "ints only");
    return (0 + ... + xs);
}
```

Tool enforcement (Clang-Tidy) `cppcoreguidelines-pro-type-vararg` flags calls to C-style vararg functions and use of `va_arg`, and explains the underlying fragility.

122.3 Adoption strategy

122.3.1 Start with vocabulary and interfaces

Type safety becomes easier when interfaces express intent:

- prefer strongly typed enums and domain-specific types over raw integers,
- avoid untyped "byte blobs" unless the representation is explicit,
- avoid "raw memory" views that require `reinterpret_cast`.

122.3.2 Use tools to enforce the profile

The Core Guidelines explicitly list tool support for adoption:

- **Clang-Tidy** provides checks named `cppcoreguidelines-*` to enforce guideline rules.
- **Microsoft C++ Code Analysis (CppCoreCheck)** supports a subset of the Core Guidelines rules inside Visual Studio.

Clang-Tidy's published check list includes the Type-safety related checks (reinterpret-cast, const-cast, cstyle-cast, member-init, static-cast-downcast, union-access, vararg), making enforcement practical in CI.

122.3.3 Know what the profile does *not* guarantee

The Type Safety Profile aims to eliminate key sources of type violations (casts, unions, uninitialized use), but it does not by itself guarantee:

- absence of out-of-bounds access (handled by the Bounds profile),
- absence of dangling-pointer access (handled by the Lifetime profile).

The Guidelines explicitly state that completeness is intended when combined with Bounds and Lifetime safety.

122.4 Practical notes and exceptions

122.4.1 Interoperability with C APIs and low-level code

Some domains require controlled violations (e.g., ABI boundaries, OS/hardware interfaces). In such cases:

- isolate the unsafe code in small, well-reviewed modules,
- wrap raw operations in typed, checked interfaces,
- document invariants, alignment requirements, and ownership/lifetime assumptions,
- keep the rest of the codebase profile-clean.

122.4.2 About future profiles (safe arithmetic)

The Profiles section notes candidates for future profiles, including narrowing arithmetic promotions/conversions and certain problematic casts. Treat these as separate concerns: keep Type safety focused on type correctness and avoiding type punning.

122.5 Summary

The Type Safety Profile (Pro.safety) is an enforceable subset of the ISO C++ Core Guidelines designed to make code statically safer with respect to:

- unsafe casts (`reinterpret_cast`, unsafe arithmetic casts, C-style casts),

- unsafe downcasts,
- casting away `const`,
- uninitialized variables and members,
- unsafe unions,
- C-style varargs.

When adopted with tool enforcement (Clang-Tidy, CppCoreCheck) and combined with Bounds and Lifetime profiles, it forms a practical foundation for modern C++ safety without changing the language.

Chapter 123

Bounds Safety Profile (Pro.bounds)

123.1 Overview

123.1.1 What a profile is

In the ISO C++ Core Guidelines, a *profile* is a coherent, tool-enforceable set of rules that targets a specific safety goal. Profiles are designed for incremental adoption: enable the profile in tools, fix the diagnostics, and steadily improve the code base.

123.1.2 Goal of the Bounds Safety Profile

The **Bounds Safety Profile (Pro.bounds)** aims to eliminate **out-of-bounds access** (reading or writing outside the valid range of an object or allocated region). Out-of-bounds access is a major source of undefined behavior, memory corruption, and security vulnerabilities.

123.1.3 The two main enemies

The profile highlights two recurring causes of bounds violations:

- **Pointer arithmetic** (`p + i`, `++p`, `p[i]` where `p` is treated like an array pointer).
- **Unchecked indexing** (especially raw arrays and `operator[]` on containers/views without checks).

123.1.4 Completeness requires Type and Lifetime profiles too

Bounds safety alone does not guarantee overall program safety. For robust safety, combine it with:

- **Type Safety Profile (Pro.safety)**: reduce unsafe casts, union hazards, uninitialized use.
- **Lifetime Safety Profile (Pro.lifetime)**: reduce dangling pointers/references and use-after-free.

If an object's lifetime is invalid, or its type assumptions are broken, bounds checks cannot make the program correct.

123.2 Profile summary (Bounds.1 – Bounds.4)

The Bounds Safety Profile is summarized by four rules, each intended to be enforceable by static analysis tools.

123.2.1 Bounds.1: Don't use pointer arithmetic. Use `span` instead

Rule: Avoid pointer arithmetic on raw pointers. Use a range type such as `gsl::span` (or `std::span` where appropriate) so the pointer is carried together with its bounds.

123.2.1.1 Rationale

Raw pointers do not carry size information. Once pointer arithmetic is allowed, it becomes easy to:

- compute the wrong address,
- walk past the end,
- lose the original base pointer needed for validation.

123.2.1.2 Bad example: pointer arithmetic

```
#include <cstddef>

int sum_bad(const int* p, std::size_t n)
{
    int s = 0;
    for (std::size_t i = 0; i < n; ++i) {
        s += *(p + i);           // pointer arithmetic
    }
    return s;
}
```

123.2.1.3 Better: pass a span

```
#include <gsl/span>

int sum_ok(gsl::span<const int> a)
{
    int s = 0;
    for (int x : a) s += x;      // no pointer arithmetic
    return s;
}
```

123.2.2 Bounds.2: Only index into arrays using constant expressions

Rule: For **built-in arrays** (and typically fixed-size arrays), only use constant-expression indices.

123.2.2.1 Rationale

Built-in arrays provide no runtime bounds information. Restricting indexing to constant expressions eliminates an entire class of dynamic indexing mistakes for raw arrays and nudges code toward safer abstractions.

123.2.2.2 Example

```
void f()
{
    int a[4] = {1,2,3,4};

    int x = a[2];     // OK (constant)
    // int y = a[i];  // discouraged by Bounds.2 (dynamic index into raw array)
    (void)x;
}
```

123.2.3 Bounds.3: No array-to-pointer decay

Rule: Avoid implicit array-to-pointer decay. Do not pass an array as a raw pointer when a range is intended.

123.2.3.1 Rationale

When an array decays to a pointer, its size is lost. That makes correct bounds handling the caller's responsibility, often leading to mismatches and off-by-one errors.

123.2.3.2 Bad: array decays to pointer

```
#include <cstddef>

void g_bad(const int* p);  // loses size
```

```
void f()
{
    int a[4] = {1,2,3,4};
    g_bad(a);              // array-to-pointer decay
}
```

123.2.3.3 Better: accept a span

```
#include <gsl/span>

void g_ok(gsl::span<const int> a);

void f()
{
    int a[4] = {1,2,3,4};
    g_ok(a);               // size preserved
}
```

123.2.4 Bounds.4: Don't use standard-library types and functions that are not bounds-checked

Rule: Prefer bounds-checked access patterns instead of unchecked subscripting.

123.2.4.1 Rationale

Unchecked indexing via `operator[]` is a common source of undefined behavior when the index is wrong. Bounds.4 aims to make out-of-range access detectable early, and to encourage designs that do not rely on unchecked subscripts.

123.2.4.2 Example: unchecked `operator[]`

```
#include <vector>
```

```
int read_bad(const std::vector<int>& v, std::size_t i)
{
    return v[i];   // unchecked; out-of-range is undefined behavior
}
```

123.2.4.3 Better: checked access (`at()` or `gsl::at`)

```
#include <vector>
#include <gsl/gsl>

int read_ok(const std::vector<int>& v, int i)
{
    return gsl::at(v, i);  // checked access (tool-friendly in Guidelines code bases)
}
```

123.2.4.4 Even better: avoid indexing

When possible, prefer range-for and algorithms:

```
#include <vector>
#include <numeric>

int sum_ok2(const std::vector<int>& v)
{
    return std::accumulate(v.begin(), v.end(), 0);
}
```

123.3 Design guidance: making bounds safety the default

123.3.1 Prefer `span` at interfaces

A strong bounds-safety design practice is to use `span` for contiguous buffers at API boundaries:

- eliminates pointer+length mismatches,
- makes slicing explicit (`first`, `subspan`, `last`),
- reduces the need for pointer arithmetic.

```
#include <gsl/span>
#include <cstddef>

int parse_header(gsl::span<const unsigned char> packet)
{
    if (packet.size() < 8) return -1;

    auto header = packet.first(8);
    auto body   = packet.subspan(8);

    int s = 0;
    for (auto b : header) s += b;
    for (auto b : body)   s += b;
    return s;
}
```

123.3.2 Validate once, then operate under an invariant

Bounds safety improves when code establishes a local invariant (e.g., `i < size`) and then uses it consistently.

```
#include <gsl/span>
#include <cstddef>

int get_checked(gsl::span<const int> a, std::size_t i)
{
    if (i >= a.size()) return 0;     // error policy chosen by the project
    return a[i];                     // now safe under the invariant
}
```

123.3.3 Choose the right error model

Bounds.4 encourages checked access, but the project must decide the error model:

- `at()` throws `std::out_of_range` (exception-based policy).
- `gsl::at()` is typically used in Guidelines-oriented code bases (often with fail-fast contracts).
- Validation + return-value policy (no exceptions) can still be bounds-safe if consistently applied.

123.4 Tool enforcement (practical adoption)

123.4.1 Clang-Tidy (cppcoreguidelines Pro.Bounds checks)

Clang-Tidy provides checks that directly support the bounds profile, including:

- `cppcoreguidelines-pro-bounds-pointer-arithmetic` (pointer arithmetic),
- `cppcoreguidelines-pro-bounds-constant-array-index` (Bounds.2),
- `cppcoreguidelines-pro-bounds-array-to-pointer-decay` (Bounds.3),
- `cppcoreguidelines-pro-bounds-avoid-unchecked-container-access` (Bounds.4 style guidance, e.g. flag `operator[]`).

123.4.2 MSVC C++ Core Guidelines checkers (CppCoreCheck)

Microsoft's code analysis includes Core Guidelines checks and maps several diagnostics directly to Bounds rules, for example:

- **C26481** corresponds to Bounds.1 (no pointer arithmetic; use span),

- additional diagnostics support array indexing/decay rules and bounds-related guidance.

123.5 Common exceptions and how to handle them safely

123.5.1 Low-level code and ABI boundaries

Some code must interact with:

- C APIs,
- OS/hardware interfaces,
- packed wire formats,
- performance-critical inner loops.

In those cases:

- isolate the unsafe operations in a small, reviewed translation unit,
- convert raw pointers to `span` as early as possible,
- keep pointer arithmetic local and minimal,
- validate sizes once at boundaries.

123.5.2 Performance notes

Bounds safety is primarily achieved by **structure**, not by repeatedly checking every subscript:

- Use spans and slicing to keep offsets correct.
- Prefer algorithms and range-for to eliminate indexing.
- Validate boundary conditions once, then run tight loops over well-defined ranges.

123.6 Summary

The Bounds Safety Profile (Pro.bounds) is an enforceable subset of the ISO C++ Core Guidelines intended to prevent out-of-bounds access by:

- banning pointer arithmetic (Bounds.1) and favoring `span`,
- restricting raw array indexing (Bounds.2),
- preventing array-to-pointer decay (Bounds.3),
- discouraging unchecked container access and encouraging safe alternatives (Bounds.4).

In practice, you get the best results by adopting `span`-based interfaces, minimizing indexing, and enabling tools (Clang-Tidy / MSVC CppCoreCheck) to enforce the profile consistently across the code base.

Chapter 124

Lifetime Safety Profile (Pro.lifetime)

124.1 Overview

124.1.1 What a profile is

In the ISO C++ Core Guidelines, a *profile* is a coherent, deterministic, and portably enforceable subset of rules designed to achieve a specific safety guarantee, and to support gradual adoption by enabling tool diagnostics for a focused set of problems.

124.1.2 Goal of the Lifetime Safety Profile

The **Lifetime Safety Profile (Pro.lifetime)** targets one of the hardest classes of C++ bugs:

> *Accessing through a pointer that doesn't point to anything.*

The Guidelines list typical reasons a pointer may be invalid: it may be uninitialized, be `nullptr`, point beyond an array range, or point to a deleted object.

124.1.3 Profile summary

The profile is summarized by a single, central rule:

- **Lifetime.1:** Don't dereference a possibly invalid pointer: detect or avoid.

124.1.4 Impact (why this matters)

The Guidelines state that, when completely enforced through a combination of style rules, static analysis, and library support, the profile:

- eliminates one major source of nasty errors in C++,
- eliminates a major source of potential security violations,
- can improve performance by eliminating redundant "paranoia" checks,
- increases confidence in correctness,
- avoids undefined behavior by enforcing a key C++ language rule.

124.2 Lifetime.1 in practice: what counts as a "possibly invalid pointer"

124.2.1 Common sources of invalid pointers and references

A pointer/reference can become invalid in many ways. The most common patterns include:

- **Uninitialized pointers** (indeterminate value).
- `nullptr` **dereference**.
- **Dangling stack address** (pointer/reference to an object that went out of scope).

- **Use-after-free** (after `delete` or after the last owning smart pointer releases).
- **Pointer/iterator invalidation by container operations** (e.g., storing a pointer/iterator/reference to a `std::vector` element, then calling `push_back` and later dereferencing the old handle).
- **Dangling view types** (e.g., `std::string_view`, spans, iterators) that refer to temporary or short-lived storage.

124.2.2 A key guideline: the profile is about dereference

The profile's essential requirement is not "never have pointers" but:

> *Never dereference a pointer (or pointer-like handle) that may be invalid.*

That focus makes it possible for tools to detect common local lifetime bugs at compile time.

124.3 Typical lifetime bugs (with safe rewrites)

124.3.1 Dangling pointer to a local object

124.3.1.1 Bad: address of a local escapes its scope

```
int* bad()
{
    int x = 7;
    return &x;              // returns pointer to a dead object
}

int use()
{
    int* p = bad();
```

```
    return *p;              // Lifetime.1 violation: dereference may be invalid
}
```

124.3.1.2 Good: return by value (prefer)

```
int good()
{
    int x = 7;
    return x;               // value is copied/moved safely
}
```

124.3.1.3 Good: return an owning smart pointer (when dynamic lifetime is intended)

```
#include <memory>

std::unique_ptr<int> good2()
{
    return std::make_unique<int>(7);
}
```

124.3.2 Use-after-delete

124.3.2.1 Bad: manual delete then dereference

```
int use_after_free()
{
    int* p = new int(9);
    delete p;
    return *p;              // Lifetime.1 violation: p may be invalid
}
```

124.3.2.2 Good: use RAII

```
#include <memory>
```

```
int ok()
{
    auto p = std::make_unique<int>(9);
    return *p;              // valid while p owns the object
}
```

124.3.3 Iterator/reference invalidation

124.3.3.1 Bad: store a reference, then grow the vector

```
#include <vector>

int bad_vec()
{
    std::vector<int> v = {1,2,3};
    int& r = v[0];          // reference to an element
    v.push_back(4);         // may reallocate -> r may dangle
    return r;               // Lifetime.1 violation: dereference may be invalid
}
```

124.3.3.2 Good: avoid keeping handles across invalidating operations

```
#include <vector>

int ok_vec()
{
    std::vector<int> v = {1,2,3};
    v.push_back(4);
    return v[0];            // obtain the reference after mutation
}
```

124.3.4 `std::string_view` bound to a temporary

124.3.4.1 Bad: view points into a temporary string

```
#include <string>
#include <string_view>

std::string_view bad_view()
{
    std::string_view sv = std::string("hello"); // view into temporary
    return sv;                                  // sv dangles immediately
}
```

124.3.4.2 Good: return an owning string

```
#include <string>

std::string good_view()
{
    return std::string("hello");
}
```

124.3.4.3 Good: view into storage owned by the caller (explicit contract)

```
#include <string>
#include <string_view>

std::string_view ok_view(const std::string& s)
{
    return std::string_view(s);  // valid as long as s outlives the returned view
}
```

124.4 How the profile is intended to be enforced

124.4.1 What tools should diagnose

The Guidelines describe enforcement goals for lifetime checking, including flagging dereferences of pointers that:

- point to an object that has gone out of scope,
- might have been invalidated by assigning `nullptr`,
- might have been invalidated by a `delete`,
- refer to a container element that might have been invalidated by container operations.

124.4.2 Microsoft lifetime analysis (practical status)

Microsoft has shipped and evolved lifetime checking in Visual Studio as part of the broader Core Guidelines checking effort, with published updates that focus specifically on detecting dangling pointers/references using static analysis and contracts between functions.

124.4.3 Relationship to other profiles

The Guidelines emphasize that safety is strongest when the profiles are combined:

- **Type safety** reduces unsafe casts, unsafe unions, and use of uninitialized objects.
- **Bounds safety** reduces out-of-bounds access (buffer overflows).
- **Lifetime safety** reduces use-after-scope and use-after-free.

124.5 Design techniques that make Lifetime.1 easier to satisfy

124.5.1 Prefer RAII and owning smart pointers

Use RAII to tie resource lifetime to object lifetime. Prefer:

- `std::unique_ptr<T>` for exclusive ownership,
- `std::shared_ptr<T>` only when shared ownership is truly required.

This reduces manual `new`/`delete` and shrinks the surface area where dangling pointers can occur.

124.5.2 Avoid returning references/pointers to locals

Prefer returning by value. Return references only when the referenced object's lifetime is clearly owned by the caller and guaranteed to outlive the reference, and when that is part of the interface contract.

124.5.3 Be careful with non-owning "view" types

Types such as `std::string_view`, iterators, and spans do not own storage. Treat them as *borrowing* handles:

- Never bind them to temporaries unless the use stays within the same full-expression.
- Make the lifetime contract explicit in the API: "this view is valid only as long as the underlying storage lives".

The lifetime profile explicitly targets common dangling-view mistakes.

124.5.4 Avoid storing references/iterators across invalidating operations

Container operations may invalidate pointers/references/iterators. When you must keep handles:

- prefer indices or stable containers where appropriate,
- or re-acquire iterators/references after the operation,
- or reserve capacity up front when using `std::vector` and you must keep references.

124.6 Summary

The Lifetime Safety Profile (Pro.lifetime) is centered on a single rule:

- **Lifetime.1:** Don't dereference a possibly invalid pointer: detect or avoid.

It targets dangling pointers/references, use-after-free, and invalidated handles (iterators/views/spans). The Guidelines describe concrete enforcement goals for tools, and Visual Studio has published multiple updates describing practical lifetime checking aimed at these defect classes.

Chapter 125

Enforcement and Tooling (Profiles)

125.1 Why enforcement matters

125.1.1 Guidelines become real only when enforced

The ISO C++ Core Guidelines include many rules that are easy to agree with but hard to follow consistently in a large code base. Manual code review alone cannot reliably catch:

- subtle lifetime issues (dangling pointers, iterator invalidation),
- accidental bounds violations (off-by-one, pointer arithmetic),
- type-safety holes (unsafe casts, union misuse, varargs),
- ownership confusion (who deletes, who borrows).

Tooling turns guidance into a repeatable, measurable process.

125.1.2 Profiles are designed for tool enforcement

The Profiles in Part 21 are not just "advice collections"; they are intended to be **checkable subsets**:

- **Pro.safety** (Type Safety): reduce unsafe casts, unsafe unions, uninitialized use, varargs.
- **Pro.bounds** (Bounds Safety): reduce out-of-bounds by banning pointer arithmetic and discouraging unchecked access.
- **Pro.lifetime** (Lifetime Safety): reduce dereferences of possibly invalid pointers and dangling views.

A profile is successful when a project can say: "this profile is enabled, and violations are treated as defects."

125.2 Enforcement layers

A practical enforcement strategy uses multiple layers, because no single tool finds everything.

125.2.1 Layer 1: Compile-time diagnostics (warnings as errors)

Compilers catch many issues early:

- suspicious conversions,
- uninitialized variables (in many cases),
- missing `return`,
- unreachable code and logic mistakes.

125.2.1.1 Guideline-oriented configuration

- Enable a strict warning set and treat it as part of the definition of “clean build”.
- Elevate important warnings to errors in CI.
- Use consistent warning baselines across platforms (MSVC/Clang/GCC) to avoid drift.

125.2.2 Layer 2: Static analysis (semantic and dataflow checking)

Static analysis goes beyond compiler warnings by modeling control flow and data flow:

- potential null dereference,
- resource leaks,
- use-after-free patterns (limited but valuable),
- bounds-safety patterns (pointer arithmetic, unchecked indexing),
- guideline-specific rules.

125.2.3 Layer 3: Dynamic analysis (sanitizers and runtime checks)

Sanitizers and runtime checkers detect issues that are hard or impossible to prove statically:

- out-of-bounds reads/writes,
- use-after-free and double free,
- data races (thread sanitizer),
- undefined behavior classes (UB sanitizer).

They are especially effective when combined with good test coverage and fuzzing.

125.2.4 Layer 4: Code review with rule-based checklists

Code review is still essential, but it should focus on what tools cannot reliably decide:

- correct invariants and API contracts,
- correct ownership design,
- correct lifetime documentation and assumptions,
- architectural decisions (where unsafe code is isolated).

Reviews become more productive when tools handle the mechanical checks.

125.3 Core Guidelines tool ecosystem

125.3.1 Clang-Tidy (cross-platform, CI-friendly)

Clang-Tidy provides a large set of checks, including the `cppcoreguidelines-*` family designed to implement many Core Guidelines rules and profiles. For the Profiles chapter, the most relevant groups include:

- `cppcoreguidelines-pro-type-*` (Type Safety profile),
- `cppcoreguidelines-pro-bounds-*` (Bounds Safety profile),
- additional `cppcoreguidelines-*` checks that support broader guideline categories.

125.3.1.1 Typical project setup: `.clang-tidy`

A minimal example that enables the profile-related checks and treats them as high priority:

```
Checks: >
  -*,
  cppcoreguidelines-pro-type-*,
  cppcoreguidelines-pro-bounds-*,
  cppcoreguidelines-*
WarningsAsErrors: ''
HeaderFilterRegex: '.*'
AnalyzeTemporaryDtors: false
FormatStyle: file
```

Engineering notes

- Start with a small, high-value set (the profiles), then expand gradually.
- Use a baseline approach: enable checks, fix existing issues, then block regressions.
- Prefer local suppression (narrow scope) over global suppression.

125.3.2 Microsoft Visual Studio: C++ Code Analysis and CppCoreCheck

Visual Studio includes Microsoft C++ Code Analysis, and it supports a subset of the C++ Core Guidelines rules (often referred to as `CppCoreCheck` in documentation). This is particularly relevant for teams building on Windows and MSVC.

125.3.2.1 How it is typically used

Two common ways:

- Run analysis in the IDE and treat warnings as actionable defects.
- Run analysis in CI using the command line build with analysis enabled.

125.3.2.2 Example: enabling analysis in MSBuild

```
<PropertyGroup>
  <RunCodeAnalysis>true</RunCodeAnalysis>
  <EnableCppCoreCheck>true</EnableCppCoreCheck>
</PropertyGroup>
```

What you gain

- diagnostics directly mapped to Guidelines rules (including profile-relevant warnings),
- consistent integration with MSVC builds and Visual Studio workflows,
- actionable messages for patterns such as pointer arithmetic (Bounds.1) and unsafe casts (Type.1).

125.3.3 Sanitizers (runtime defect detection)

Sanitizers are a key enforcement tool for Profiles because they detect real executions of undefined behavior:

- AddressSanitizer (ASan): out-of-bounds, use-after-free, double free.
- UndefinedBehaviorSanitizer (UBSan): undefined behavior categories (e.g., misaligned access, invalid shifts).
- MemorySanitizer (MSan): use of uninitialized memory (requires instrumented libs).
- ThreadSanitizer (TSan): data races and some thread issues.

125.3.3.1 Recommended practice

- Run sanitizers in CI on at least one configuration (e.g., Clang + ASan/UBSan).

- Treat sanitizer findings as correctness defects, not test flakiness.
- Add regression tests for every sanitizer-discovered bug.

125.3.4 Debug-mode checked interfaces

For bounds and lifetime issues, checked interfaces amplify the effect of testing:

- prefer checked access (`at()` / `gsl::at`) where indexes are not trivially safe,
- use contract checks (`gsl::Expects`/`gsl::Ensures`) to fail fast on broken assumptions,
- use `span` to keep pointer+size together and reduce indexing mistakes.

125.4 Mapping tools to profiles

125.4.1 Type Safety Profile (Pro.safety)

A practical enforcement set typically includes:

- Clang-Tidy: `cppcoreguidelines-pro-type-*` checks (reinterpret cast, const cast, C-style casts, unsafe union access, vararg usage, member initialization, etc.).
- Compiler warnings for narrowing conversions and suspicious casts.
- UBSan for runtime detection of undefined behavior classes.

125.4.2 Bounds Safety Profile (Pro.bounds)

A practical enforcement set typically includes:

- Clang-Tidy: `cppcoreguidelines-pro-bounds-*` checks (pointer arithmetic, array-to-pointer decay, constant array indexing, unchecked container access).

- Visual Studio CppCoreCheck diagnostics related to bounds rules (notably the pointer arithmetic rule aligned with Bounds.1).
- ASan for runtime detection of out-of-bounds.

125.4.3 Lifetime Safety Profile (Pro.lifetime)

A practical enforcement set typically includes:

- Visual Studio lifetime analysis (where available) to detect common dangling patterns.
- ASan for use-after-free and use-after-scope detection at runtime.
- Project conventions that prevent returning views to temporaries and storing borrowing handles beyond safe scopes.

125.5 A workable adoption plan for real code bases

125.5.1 Step 1: Establish a baseline

- Run the tools once on the whole code base.
- Export the diagnostics and classify them (true defect, acceptable exception, false positive).
- Fix the highest-impact categories first: bounds and lifetime issues typically give the largest safety return.

125.5.2 Step 2: Prevent regressions

- Make the tool run part of CI.

- Fail the build on new violations (“no new warnings”).
- Keep the baseline shrinking over time.

125.5.3 Step 3: Refactor toward profile-friendly code

The profiles become much easier when the code uses safe vocabulary types:

- replace pointer+length with `span`,
- mark non-null pointers with `not_null`,
- represent ownership with RAII (`unique_ptr`, containers); use `owner<T*>` only when forced,
- reduce casts by redesigning APIs.

125.5.4 Step 4: Isolate unavoidable unsafe code

Some code must do low-level operations (ABI boundaries, hardware I/O, wire formats). Policy for such code:

- isolate it in small modules,
- document invariants (alignment, lifetime, ownership),
- provide a safe wrapper API to the rest of the project,
- add tests and sanitizer runs specifically targeting this module.

125.6 Suppressions and exceptions

125.6.1 Prefer local, documented suppressions

When you must violate a rule, do it deliberately:

- keep the exception narrowly scoped,
- add a short comment explaining why it is safe here,
- prefer tool-supported suppression mechanisms rather than disabling whole rule groups.

125.6.2 Avoid blanket disabling of profile checks

Disabling a whole category (e.g., all bounds checks) removes the value of adopting the profile. If a check is noisy, tune configuration (scope, header filters, excluded third-party code) before turning it off.

125.7 Measuring success

A practical definition of success for “Profiles enforcement” includes:

- zero new profile violations in CI,
- a steadily decreasing baseline of legacy violations,
- sanitizer builds that run clean on regression tests,
- fewer production crashes linked to memory corruption and undefined behavior.

125.8 Summary

Enforcement and tooling is the bridge from written guidelines to reliable software. For the Profiles:

- use **Clang-Tidy** to enforce profile rules across platforms,
- use **MSVC CppCoreCheck** for Windows/MSVC-centric teams,
- use **sanitizers** to detect real runtime undefined behavior,
- refactor toward safe vocabulary types (`span`, `not_null`, RAII owners),
- adopt a CI baseline strategy: fix, prevent regressions, shrink the baseline.

Part XXII

Enforcement and Tooling

Chapter 126

clang-tidy

126.1 What clang-tidy is

clang-tidy is a Clang/LLVM *LibTooling*-based C++ "linter" and static-analysis driver. It provides an extensible framework for diagnosing common programming errors, guideline violations, interface misuse, and issues that can be deduced by static analysis. It can also run many checks from the Clang Static Analyzer.

126.2 Where clang-tidy fits in Core Guidelines enforcement

126.2.1 Core Guidelines and profiles

The ISO C++ Core Guidelines include rules that are deliberately chosen to be enforceable by tools where possible. In practice, clang-tidy is one of the primary cross-platform toolchains used to enforce Core Guidelines recommendations in CI, especially for:

- **Type Safety Profile** checks (e.g., unsafe casts, union misuse),

- **Bounds Safety Profile** checks (e.g., pointer arithmetic, array-to-pointer decay, unchecked access),
- broader guideline-related checks in the `cppcoreguidelines-` family.

126.2.2 Check naming and families

Each check has a stable name (for example, `cppcoreguidelines-init-variables`). Checks are grouped by prefixes such as: `clang-analyzer-`, `bugprone-`, `modernize-`, `readability-`, `performance-`, and `cppcoreguidelines-`.

126.3 Fundamental requirement: compile commands

126.3.1 Why a compilation database is important

`clang-tidy` needs the same compilation options your project uses (include paths, macros, language standard, target options). The most reliable way is a *compilation database* named `compile_commands.json`.

126.3.2 Generating `compile_commands.json` with CMake

A common workflow is to ask CMake to export the compilation database:

```
cmake -S . -B build -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
cmake --build build
```

Then run `clang-tidy` with `-p` pointing to that build directory.

126.3.3 If you do not have a compilation database

You can still run `clang-tidy` by passing compilation arguments after `--`, but this is harder to scale for large projects:

```
clang-tidy src/file.cpp -- -Iinclude -DMY_DEFINE=1 -std=c++23
```

This is valid, but a compilation database is the preferred approach for real code bases.

126.4 Selecting checks

126.4.1 Enabling and disabling checks with `-checks=`

Checks are selected with `-checks=` using a comma-separated list of globs, where:

- globs without a leading `-` enable matching checks,
- globs prefixed with `-` disable matching checks,
- globs are processed in order.

126.4.1.1 Examples

Disable all checks and enable only the analyzer checks (except certain C++ ones):

```
clang-tidy test.cpp -checks=-*,clang-analyzer-*,-clang-analyzer-cplusplus* -- -std=c++23
```

Enable Core Guidelines checks commonly used in enforcement:

```
clang-tidy test.cpp -checks=-*,cppcoreguidelines-* -- -std=c++23
```

126.4.2 Listing checks

`-list-checks` lists enabled checks for the current configuration. Use `-checks=*` to list all available checks in the installed toolchain.

```
clang-tidy -checks=* -list-checks
```

126.5 Configuration files: `.clang-tidy`

126.5.1 How configuration is discovered

For each source file, `clang-tidy` looks for a `.clang-tidy` file in the closest parent directory (walking up the directory tree). Command-line options override configuration-file settings.

126.5.2 Key configuration fields

A `.clang-tidy` file is written in YAML and can define:

- `Checks`
- `WarningsAsErrors`
- `HeaderFilterRegex` and `ExcludeHeaderFilterRegex`
- `CheckOptions` (per-check options)
- `ExtraArgs` / `ExtraArgsBefore`
- `FormatStyle`
- `InheritParentConfig` (merge parent config)

These correspond closely to command-line options such as `--checks`, `--warnings-as-errors`, `--header-filter`, and `--format-style`.

126.5.3 Inspecting the effective configuration

Use `--dump-config` to print the configuration that will be used for a given file (including inherited settings).

```
clang-tidy --dump-config src/file.cpp -p build
```

126.5.4 A practical Core-Guidelines oriented `.clang-tidy`

This example starts from "nothing", then enables the Core Guidelines profiles and a few high-value families:

```
---
Checks:
  - '-*'
  - 'cppcoreguidelines-pro-type-*'
  - 'cppcoreguidelines-pro-bounds-*'
  - 'cppcoreguidelines-*'
  - 'bugprone-*'
  - 'modernize-*'
  - 'readability-*'
WarningsAsErrors: ''
HeaderFilterRegex: '^(src|include)/'
FormatStyle: file
InheritParentConfig: true
...
```

Notes

- Restricting `HeaderFilterRegex` prevents noise from third-party headers.
- Start with profiles first (type and bounds) to get high safety return early.
- Add more checks gradually as the code base becomes cleaner.

126.6 Fixes and refactoring support

126.6.1 Applying fixes directly

Many checks provide "fix-it" hints. Use `--fix` to apply suggested fixes.

```
clang-tidy src/file.cpp -p build --fix
```

126.6.2 Exporting fixes for controlled application

In large projects, it is often safer to export fixes and apply them in a controlled step:

- `--export-fixes=<file>` writes suggested fixes to a YAML file,
- the fixes can be applied later using `clang-apply-replacements`.

```
clang-tidy src/file.cpp -p build --export-fixes=fixes.yaml
clang-apply-replacements fixes.yaml
```

126.6.3 Formatting around fixes

When `--fix` is used, `--format-style` controls formatting around applied fixes:

- `none` disables formatting,
- `file` uses the nearest `.clang-format`,
- named styles (e.g., `llvm`, `google`) are available.

126.7 Automation scripts shipped with `clang-tidy`

126.7.1 `run-clang-tidy.py`: run the whole database in parallel

For projects with many translation units, running `clang-tidy` file-by-file is slow. The distribution includes `run-clang-tidy.py`, which runs `clang-tidy` over all files in a compilation database and supports parallel execution. It also supports most `clang-tidy` options such as `-checks`, `-fix`, and header filtering.

```
# Run on everything in the compilation database:
run-clang-tidy.py -p build

# Run in parallel and apply fixes:
run-clang-tidy.py -p build -j 8 -fix -checks=-*,readability-*
```

126.7.2 `clang-tidy-diff.py`: focus on changed lines

`clang-tidy-diff.py` is designed for code review and CI. It reports diagnostics only on changed lines (even though the full file is analyzed and then filtered). This is helpful to prevent new issues without being blocked by old ones.

```
git diff -U0 --no-color HEAD^ | clang-tidy-diff.py -p1 -checks=-*,cppcoreguidelines-*
```

Important limitation Because the script filters output to changed lines, it can miss diagnostics that are reported outside the diff (for example, a rule reported at a function declaration when only the function body changed). Use full-project runs for release gates and major refactors.

126.8 Suppressing diagnostics safely

126.8.1 Prefer code fixes over suppressions

Suppressions should be the exception. Many warnings indicate real design problems, especially in Core Guidelines profiles. When a warning is correct, fix the code.

126.8.2 Generic suppression comments

If a warning is known to be correct-by-design for a specific line, `clang-tidy` supports generic suppression comments:

- `NOLINT` suppresses warnings on the same line,
- `NOLINTNEXTLINE` suppresses warnings on the next line,
- `NOLINTBEGIN ...NOLINTEND` suppresses warnings over a range.

These can optionally list check names (with globbing) in parentheses.

126.8.2.1 Examples

```
class Foo {
  Foo(int x); // NOLINT

  Foo(double x); // NOLINT(google-explicit-constructor, google-runtime-int)

  // NOLINTNEXTLINE(cppcoreguidelines-pro-bounds-pointer-arithmetic)
  void* p = nullptr;

  // NOLINTBEGIN(cppcoreguidelines-pro-type-const-cast)
  void legacy_api(const int* p) {
    int* q = const_cast<int*>(p);
    (void)q;
  }
  // NOLINTEND(cppcoreguidelines-pro-type-const-cast)
};
```

126.8.3 Suppression policy (recommended)

- Always add a short reason comment next to the suppression.
- Prefer suppressing a specific check name (or narrow glob) rather than `NOLINT` with no scope.
- Do not use suppressions to hide profile violations in normal production code.

126.9 Clang diagnostics inside `clang-tidy`

126.9.1 Filtering compiler warnings and errors

`clang-tidy` can surface compiler diagnostics with check names starting with `clang-diagnostic-`. For example, a Clang warning controlled by `-Wliteral-conversion` may appear as `clang-diagnostic-literal-conversion`. Some compiler errors are reported as `clang-diagnostic-error` and cannot be disabled because `clang-tidy` needs valid code to analyze.

126.9.2 Warnings-as-errors

`--warnings-as-errors=` upgrades warnings from selected checks to errors (same glob format as `--checks`). This is commonly used in CI to fail builds on specific rule violations.

```
clang-tidy src/file.cpp -p build \
  --checks=-*,cppcoreguidelines-pro-bounds-* \
  --warnings-as-errors=cppcoreguidelines-pro-bounds-*
```

126.10 Using `clang-tidy` effectively in CI

126.10.1 Baseline first, then prevent regressions

A realistic adoption strategy:

- Run `clang-tidy` once to establish a baseline of existing issues.
- Fix the most dangerous categories first (bounds, type safety, lifetime-related warnings where available).

- In CI, block *new* violations (e.g., use diff-based checks or enforce clean output for touched files).
- Expand the enabled set of checks gradually.

126.10.2 Recommended CI modes

- **Pull request / code review gate:** `clang-tidy-diff.py` (focus on new code).
- **Nightly or pre-release:** full run via `run-clang-tidy.py` over the entire compilation database.

126.11 How `clang-tidy` relates to other LLVM tooling

126.11.1 `clang-format` and formatting

`clang-tidy` is not a formatter. It can optionally format *around applied fixes* using `--format-style`. A clean workflow often uses:

- `clang-format` for layout,
- `clang-tidy` for correctness, modernization, and guideline enforcement.

126.11.2 IDE integration

Many editors and IDEs integrate `clang-tidy` to show diagnostics while typing, but CI remains the authoritative enforcement mechanism.

126.12 Limitations and what `clang-tidy` cannot guarantee

126.12.1 It depends on correct build flags

Incorrect compilation options lead to false positives, false negatives, and missing headers. A correct compilation database is essential.

126.12.2 It is not a proof system

`clang-tidy` is best at enforcing:

- rule-based patterns,
- common bug patterns,
- local data-flow properties.

Some properties require deeper interprocedural analysis or runtime checking. Use sanitizers and tests for those.

126.13 Summary

`clang-tidy` is a practical and widely used foundation for enforcing the ISO C++ Core Guidelines in real projects:

- select checks with `--checks` (globs, ordered, enable/disable),
- configure projects with `.clang-tidy` and inspect with `--dump-config`,
- use a compilation database (`-p build`) for correctness,
- automate with `run-clang-tidy.py` (parallel full runs) and `clang-tidy-diff.py` (review/CI),
- apply or export fixes with `--fix` / `--export-fixes`,

- suppress only when necessary using `NOLINT` mechanisms, and document the reason.

Chapter 127

Static analysis

127.1 Definition and scope

127.1.1 What static analysis means in C++

Static analysis is the automatic inspection of source code (and sometimes build metadata) to detect defects *without executing the program*. It ranges from simple rule-based checks (style and API misuse) to advanced analyses that approximate program behavior across control flow and function calls.
Static analysis is especially valuable for enforcing the ISO C++ Core Guidelines because many guideline violations are *structural*: once introduced, they are hard to catch reliably by tests alone (for example, lifetime and bounds issues that may only trigger under rare inputs).

127.1.2 What static analysis can and cannot guarantee

Static analysis tools are *bug-finding* tools, not proof systems:

- They can report **false positives** (warnings for code that is actually safe).

- They can miss real bugs (**false negatives**).
- Advanced analyses are often expensive and may stop exploring very complex code paths or fall back to imprecise summaries.

127.2 Main categories of static analysis

127.2.1 Compiler diagnostics (warnings)

Compilers perform a first level of static analysis:

- suspicious conversions and narrowing,
- missing returns, unreachable code,
- certain undefined-behavior patterns (limited),
- some uninitialized-variable cases (limited).

This layer is fast and should be enabled for all builds, but it is not sufficient for deeper lifetime/bounds reasoning.

127.2.2 Rule-based linters

Rule-based linters detect patterns that match specific rules. They are excellent for:

- guideline enforcement (naming, initialization rules, safe interfaces),
- modernization transformations,
- banning dangerous constructs (e.g., C-style casts, raw `new`/`delete` in most code).

127.2.3 Path-sensitive / dataflow static analyzers

These analyzers approximate program execution across branches, loops, and function calls. They can find:

- null dereferences,
- leaks and double frees,
- use-after-free patterns,
- out-of-bounds conditions (in some analyzers),
- API contract violations (when modeled).

They are more expensive and still not complete, but often catch the most dangerous classes of defects early.

127.2.4 Semantic security scanning (query-based analysis)

Some systems (notably CodeQL) build a database representation of your code and run query suites (including security-focused rules) against it. This is often used for large repositories and supply-chain security workflows.

127.3 Why static analysis is essential for the Core Guidelines

127.3.1 Profiles are designed to be checkable

The Guidelines Profiles (Type, Bounds, Lifetime) were designed so that tools can enforce meaningful subsets consistently. In practice, static analysis provides:

- repeatable enforcement in CI,

- measurable progress (baseline and regression prevention),
- early detection of high-severity memory-safety issues.

127.3.2 Where static analysis is strongest

Static analysis gives the highest return when it focuses on:

- **memory safety**: bounds, lifetime, ownership clarity,
- **type safety**: narrowing and dangerous casts,
- **resource management**: leaks, double release, missing close,
- **contract violations**: preconditions/postconditions (when modeled).

127.4 Core tools you will meet in modern C++ workflows

127.4.1 `clang-tidy` (rule-based lint + fix-its)

`clang-tidy` is a Clang-based C++ linter and static-analysis driver that provides an extensible framework for detecting and fixing typical programming errors, style violations, and interface misuse.

127.4.1.1 Typical usage with a compilation database

```
# Generate compile_commands.json (CMake)
cmake -S . -B build -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
cmake --build build

# Run clang-tidy on a file using the build flags:
clang-tidy src/file.cpp -p build
```

127.4.1.2 Enabling Core Guidelines-related checks

```
clang-tidy src/file.cpp -p build \

  ↪  --checks=-*,cppcoreguidelines-pro-type-*,cppcoreguidelines-pro-bounds-*,cppcoreguidelines-*
```

127.4.1.3 Why it matters for this book

`clang-tidy` is often the practical enforcement engine for many Core Guidelines rules and the Profiles check families (especially bounds and type-safety patterns).

127.4.2 Clang Static Analyzer (path-sensitive symbolic execution)

The Clang Static Analyzer is a source-code analysis tool that finds bugs in C/C++/Objective-C. It is path-sensitive and interprocedural, based on symbolic execution.

127.4.2.1 Whole-project runs with `scan-build`

A common workflow is to wrap your normal build:

```
scan-build cmake -S . -B build
scan-build cmake --build build
```

This produces a report of discovered issues such as leaks, null dereferences, and other modeled bug patterns.

127.4.3 MSVC code analysis and Core Guidelines checkers

Visual Studio supports a subset of the C++ Core Guidelines rules via its Microsoft C++ Code Analysis tooling. A reference list of supported checks and their mapping is maintained in the documentation.

127.4.3.1 Typical build integration

```
# Conceptually: run the MSVC analyzer as part of your build
# (Exact switches and integration depend on your build system/IDE configuration)
```

The practical goal is to run analysis regularly (IDE and CI) and treat new warnings as defects.

127.4.4 GCC static analyzer (`-fanalyzer`)

GCC provides a static analysis mode enabled by `-fanalyzer`. The GCC manual describes it as an expensive interprocedural analysis that explores "interesting" paths through the code and issues warnings for problems found. It is explicitly described as neither sound nor complete (false positives and negatives are possible).

127.4.4.1 Important note on language coverage

The GCC manual states that, in the referenced release documentation, the analyzer is suitable for use on C code. In C++ projects, teams typically use it mainly for C components or mixed-language code bases where the C portions are critical.

127.4.4.2 Example usage

```
# Run GCC static analyzer diagnostics (for C code)
gcc -fanalyzer -Wall -Wextra -O0 -g file.c
```

127.4.4.3 What it detects

When enabled, `-fanalyzer` turns on a wide set of analyzer warnings (for example: null dereference, double free, use-after-free, out-of-bounds, leaks, and many others).

127.4.5 CodeQL (query-based static analysis for security)

CodeQL-based code scanning works by running queries against CodeQL databases generated from your repository. For compiled languages such as C/C++, building the code is commonly part of the database generation process. GitHub documents the built-in C/C++ query suites used for scanning.

127.5 Static analysis workflow patterns

127.5.1 Baseline then block regressions

A realistic adoption plan:

- Run the analyzer and record current findings (baseline).
- Fix high-severity issues first (bounds, lifetime, double-free/leaks).
- In CI, fail builds on *new* violations.
- Gradually expand the enabled rule set.

127.5.2 Two CI modes: diff-focused and full

- **Diff-focused (pull requests):** analyze changed files/lines to prevent new debt.
- **Full (nightly or release gate):** analyze the whole code base to prevent drift.

127.5.3 Triage discipline

To keep static analysis useful:

- classify each diagnostic (real bug, risky pattern, false positive, acceptable exception),

- suppress only with narrow scope and a short justification comment,
- track false positives separately (tool upgrade or configuration changes may remove them).

127.6 How static analysis supports the Profiles

127.6.1 Type Safety Profile

Static analysis helps detect unsafe casts, suspicious conversions, and other type-unsafe patterns. This is primarily enforced via rule-based checks (`clang-tidy` families, MSVC Core Checkers) plus strong compiler warnings.

127.6.2 Bounds Safety Profile

Static analysis is strong at enforcing:

- "no pointer arithmetic" policy in most code,
- safe indexing patterns,
- discouraging raw array decay where spans should be used.

127.6.3 Lifetime Safety Profile

Lifetime is the hardest profile to enforce fully. Static analysis helps most when:

- the code uses safe vocabulary types (RAII owners, spans, not-null wrappers),
- interfaces express lifetime and ownership clearly,
- tools are run frequently, and warnings are treated as defects.

127.7 Limitations and best practices

127.7.1 Static analysis needs correct build information

Most modern analyzers require accurate compilation flags (include paths, defines, standard version). Without this, diagnostics become unreliable.

127.7.2 Do not confuse “no warnings” with “no bugs”

Even strong analyzers are not complete. The correct engineering approach is **layering**:

- static analysis for early detection and enforcement,
- sanitizers and tests for runtime behavior and undefined behavior manifestation,
- code review for architecture and correctness arguments.

127.7.3 Keep unsafe code small and audited

When low-level code must use pointer arithmetic or raw buffers (ABI boundaries, hardware I/O, wire formats):

- isolate it,
- document invariants,
- add focused tests,
- apply stricter analysis and runtime checking to that module.

127.8 Summary

Static analysis is the backbone of practical Core Guidelines enforcement:

- `clang-tidy` provides scalable rule-based enforcement and many fix-its.
- Clang Static Analyzer provides path-sensitive interprocedural bug finding.
- MSVC code analysis includes a supported subset of Core Guidelines checks.
- GCC `-fanalyzer` provides an expensive path exploration engine for C code, explicitly not sound/complete.
- CodeQL enables query-based scanning (notably security-focused) over a compiled-language database.

Used together with CI discipline, static analysis turns guideline intent into measurable engineering reality.

Chapter 128

CI integration

128.1 What CI integration means for the Core Guidelines

128.1.1 Goal

Continuous Integration (CI) is the practice of running automated verification on every change (pull request, merge request, or commit) so that defects are detected early and consistently.
For ISO C++ Core Guidelines enforcement, CI integration has a specific purpose:

- run the same checks the same way on every change,
- prevent regressions (no new violations),
- gradually reduce existing technical debt (baseline reduction),
- produce artifacts (logs, reports, SARIF) that reviewers can inspect.

128.1.2 Why CI matters more than local checking

Local checking is optional and inconsistent. CI is non-optional and repeatable. A rule is only truly enforced when:

- it runs automatically in CI,
- new violations fail the pipeline (or at least block merge),
- suppressions require justification and review.

128.2 Core CI building blocks

128.2.1 Build metadata: `compile_commands.json`

Most modern C++ analysis tools need the project's real compilation flags (include paths, macros, standard version, target flags). The standard way to provide that is a compilation database named `compile_commands.json`.
With CMake, it is typically generated by enabling: `CMAKE_EXPORT_COMPILE_COMMANDS`. It emits the exact compile commands for all translation units.

```
cmake -S . -B build -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
cmake --build build
```

This file becomes the foundation for:

- `clang-tidy` (`-p build`),
- Clang Static Analyzer workflows that reuse build flags,
- many third-party tools that consume compilation databases.

128.2.2 A layered verification model

A robust CI pipeline layers checks because no single tool catches everything:

1. **Build** (with strict warnings)
2. **Unit tests** (fast, deterministic)
3. **Static analysis** (Core Guidelines checks, bug patterns)
4. **Sanitizers** (ASan/UBSan/TSan where available)
5. **Security scanning** (optional, e.g., CodeQL)

128.2.3 Two run modes: PR gate vs. full gate

- **PR gate (fast):** focus on changed code and quick feedback.
- **Full gate (thorough):** run the full suite nightly or before release.

A common pattern:

- PR gate: build + tests + `clang-tidy` on touched files/lines.
- Nightly: build + tests + full `clang-tidy` + sanitizers + CodeQL.

128.3 Turning Core Guidelines Profiles into CI policy

128.3.1 Pick profiles as the enforcement baseline

For safety-focused CI, start with Profiles because they are designed to be tool-checkable:

- **Pro.safety (Type Safety)**: avoid dangerous casts, unsafe unions, uninitialized use.

- **Pro.bounds (Bounds Safety)**: avoid pointer arithmetic, unchecked access patterns.
- **Pro.lifetime (Lifetime Safety)**: reduce invalid dereferences and dangling uses (tool support varies).

In practice, `clang-tidy` is the primary cross-platform engine for enforcing many profile rules through the `cppcoreguidelines-*` check families.

128.3.2 Policy: "no new warnings"

For real code bases, you often cannot fix everything in one step. The most effective policy is:

- establish a baseline (known existing findings),
- block new findings in CI,
- shrink the baseline continuously.

128.4 Practical CI patterns for `clang-tidy`

128.4.1 Recommended `.clang-tidy` configuration

A typical configuration for Core Guidelines enforcement:

```
---
Checks:
  - '-*'
  - 'cppcoreguidelines-pro-type-*'
  - 'cppcoreguidelines-pro-bounds-*'
  - 'cppcoreguidelines-*'
  - 'bugprone-*'
WarningsAsErrors: ''
HeaderFilterRegex: '^(src|include)/'
```

```
FormatStyle: file
...
```

128.4.2 Run `clang-tidy` using the compilation database

```
clang-tidy -p build src/file.cpp
```

128.4.3 PR-mode: check only changed lines

A common PR gating technique is to analyze full files but report only on changed lines (reducing noise in legacy code). A typical workflow uses a diff-to-tidy filter script:

```
git diff -U0 --no-color origin/main...HEAD \
  | clang-tidy-diff.py -p1 -path build \
    -checks='-*,cppcoreguidelines-pro-type-*,cppcoreguidelines-pro-bounds-*'
```

Engineering note This approach is excellent for preventing new debt, but it must be complemented by periodic full runs to avoid drift and to detect issues outside the changed-line region.

128.4.4 Nightly-mode: full-project run

Full-project runs are usually done using a helper script that iterates over `compile_commands.json` and runs checks in parallel:

```
run-clang-tidy.py -p build -j 8 \
  -checks='-*,cppcoreguidelines-pro-type-*,cppcoreguidelines-pro-bounds-*'
```

128.4.5 Fail the CI job on selected checks

In CI, you often want specific rule families to be treated as errors:

```
clang-tidy -p build src/file.cpp \
  --checks='-*,cppcoreguidelines-pro-bounds-*' \
  --warnings-as-errors='cppcoreguidelines-pro-bounds-*'
```

128.5 Sanitizers in CI (runtime enforcement)

128.5.1 What sanitizers contribute

Static analysis prevents many issues, but sanitizers catch real executions of undefined behavior:

- AddressSanitizer (ASan): out-of-bounds, use-after-free.
- UndefinedBehaviorSanitizer (UBSan): many UB categories.
- ThreadSanitizer (TSan): data races (where supported).

128.5.2 CMake example: sanitizer build preset

A common CI setup defines a separate build directory for sanitizers:

```
cmake -S . -B build-asan -DCMAKE_BUILD_TYPE=RelWithDebInfo \
  -DCMAKE_CXX_FLAGS="-fsanitize=address,undefined -fno-omit-frame-pointer" \
  -DCMAKE_C_FLAGS="-fsanitize=address,undefined -fno-omit-frame-pointer"

cmake --build build-asan
ctest --test-dir build-asan --output-on-failure
```

CI policy Treat sanitizer failures as correctness defects. Add regression tests for every sanitizer-discovered bug.

128.6 Microsoft tooling in CI (Windows/MSVC)

128.6.1 MSVC Code Analysis and Core Guidelines checks

Visual Studio and MSBuild can run Microsoft C++ Code Analysis, including a supported subset of Core Guidelines checks. In Windows-heavy environments, a practical strategy is:

- run `clang-tidy` cross-platform where possible,
- additionally run MSVC code analysis in Windows CI to catch tool-specific diagnostics.

128.6.2 MSBuild property example

When building with MSBuild, analysis is typically enabled through project properties. A minimal example (project-specific details vary):

```
<PropertyGroup>
  <RunCodeAnalysis>true</RunCodeAnalysis>
  <EnableCppCoreCheck>true</EnableCppCoreCheck>
</PropertyGroup>
```

128.7 Security scanning in CI (CodeQL example)

128.7.1 What CodeQL adds

CodeQL (when used in a CI platform that supports it) is often used to detect security-sensitive patterns at scale. For compiled languages (C/C++), the scan typically needs a successful build or a configured build command so CodeQL can construct an analysis database.

128.7.2 CI placement

A common placement is:

- PR gate: optional (depends on cost and platform limits),
- nightly or scheduled: recommended for consistent coverage.

128.8 Caching, artifacts, and reproducibility

128.8.1 Caching

For large C++ projects, CI speed depends heavily on caching:

- compiler cache (`ccache` / `sccache`),
- dependency caches,
- build directory reuse (when safe),
- prebuilt toolchain images for consistent versions.

128.8.2 Artifacts

Store artifacts that help debugging and auditing:

- build logs,
- test logs,
- sanitizer reports,
- `clang-tidy` reports (and exported fixes where used),
- SARIF outputs (when the CI platform supports code scanning views).

128.8.3 Reproducibility rules

A strong CI policy enforces:

- pinned tool versions (compiler + tools),
- pinned dependency versions,
- consistent build flags across CI agents,
- a documented "how to reproduce locally" command set.

128.9 A reference CI layout

128.9.1 Recommended job matrix

A practical, high-value matrix without extreme cost:

- **Linux (Clang):** build + tests + `clang-tidy` (profiles)
- **Linux (Clang + ASan/UBSan):** build + tests
- **Windows (MSVC):** build + tests + MSVC code analysis (optional but valuable)
- **Scheduled (nightly):** full `clang-tidy` + optional CodeQL

128.9.2 Principle: treat safety checks as merge gates

If the project declares the Profiles as policy, then:

- new violations of **Pro.bounds** and **Pro.safety** should block merging,
- lifetime findings should block merging when tool confidence is high,
- suppressions should require a short justification and review.

128.10 Common failure modes (and fixes)

128.10.1 Wrong flags lead to noisy or missing diagnostics

If `clang-tidy` or analyzers do not see the real build flags, results become unreliable. Fix: always use `compile_commands.json` from the CI build, and run analysis against that database.

128.10.2 Too many warnings cause developers to ignore CI

If CI emits thousands of findings, developers learn to ignore it. Fix: adopt "baseline then regressions" and focus on profiles first.

128.10.3 Third-party headers dominate findings

Fix: set header filters to limit reporting to project code (for example, `src/` and `include/`), and keep external code out of the enforcement scope unless you own it.

128.11 Summary

CI integration is how guideline intent becomes enforceable engineering policy. A reliable Core Guidelines CI approach:

- generates `compile_commands.json` and uses it as the shared build truth,
- enforces `clang-tidy` Core Guidelines profile checks in PR and nightly modes,
- runs sanitizers to catch runtime undefined behavior,
- optionally runs MSVC code analysis on Windows builds,
- uses a baseline strategy to prevent regressions and continuously reduce existing issues,

- stores useful artifacts and pins tool versions for reproducibility.

Chapter 129

Gradual rule enforcement

129.1 Why gradual enforcement is necessary

129.1.1 Large code bases cannot be converted all at once

The ISO C++ Core Guidelines explicitly recognize that fully converting a large, existing code base in a single step is usually infeasible, and that the rules are intended to support **gradual adoption**.

129.1.2 Profiles enable incremental enforcement

The Guidelines introduce **profiles** as sets of related rules that can be applied first, while leaving other guidelines for later. This is the mechanism that makes gradual enforcement realistic, especially for safety improvements.

129.2 Principles of gradual enforcement

129.2.1 Principle 1: Establish a baseline, then stop regressions

A practical approach is:

1. Run tools on the whole repository to establish a **baseline** of existing findings.
2. Decide which findings are real bugs, risky patterns, and noise.
3. In CI, **block new violations** while you pay down the baseline over time.

This prevents quality from getting worse while allowing steady improvement.

129.2.2 Principle 2: Enforce the highest-risk rules first

Not all rules have the same safety impact. Start with rules that reduce undefined behavior and security risk:

- **Bounds** rules (out-of-bounds prevention),
- **Type-safety** rules (unsafe casts, unsafe unions, uninitialized use),
- **Lifetime** rules where tool support is strong enough (dangling dereference patterns).

Profiles are designed exactly for this targeted enforcement.

129.2.3 Principle 3: Prefer structural refactors over endless suppressions

Suppressions can hide defects, create blind spots, and increase long-term maintenance cost. A gradual strategy should aim to reduce the need for suppressions by improving design:

- replace pointer+length with `span`,
- express non-nullness explicitly,
- represent ownership with RAII owners,

- avoid storing borrowing handles across invalidating operations,
- redesign APIs rather than relying on casts.

129.2.4 Principle 4: Keep enforcement deterministic and reviewable

A rule is truly enforced only if:

- it runs automatically in CI,
- it is deterministic (same inputs $\rightarrow$ same results),
- failures are visible and reviewable (logs/artifacts),
- exceptions require explicit justification.

129.3 A staged adoption model

129.3.1 Stage 0: Measure (no gates yet)

Run the analyzers and produce reports, but do not fail builds:

- collect a baseline,
- identify the "top 20" recurring problems,
- classify by severity (safety-critical vs. maintainability).

129.3.2 Stage 1: Stop new violations in touched code

This stage blocks regressions without demanding a full cleanup. A common technique is **diff-focused analysis**: analyze files normally but report only diagnostics on changed lines.

The LLVM documentation for `clang-tidy-diff.py` states that it filters results to changed lines; it does *not* make analysis faster because `clang-tidy` still analyzes the full file and then filters. This makes it suitable for code review and CI gating.

```
# PR gate: only report diagnostics on changed lines
git diff -U0 --no-color origin/main...HEAD \
  | clang-tidy-diff.py -p1 -path build \
    -checks='-*,cppcoreguidelines-pro-type-*,cppcoreguidelines-pro-bounds-*'
```

129.3.3 Stage 2: Make a small rule set non-negotiable

Select a minimal set of high-signal rules and treat violations as build failures. For example:

- **Pro.bounds**: pointer arithmetic and unchecked access patterns,
- **Pro.safety**: unsafe casts and uninitialized use.

```
# Fail the build when new Pro.bounds diagnostics appear
clang-tidy -p build src/file.cpp \
  --checks='-*,cppcoreguidelines-pro-bounds-*' \
  --warnings-as-errors='cppcoreguidelines-pro-bounds-*'
```

129.3.4 Stage 3: Expand coverage gradually

As the code base becomes cleaner:

- widen the check set (`bugprone-*`, selected `modernize-*`),
- expand from changed-lines gating to changed-files gating,
- add periodic full-project runs (nightly or pre-release).

129.3.5 Stage 4: Enforce full profiles (release gate)

At maturity:

- run full `run-clang-tidy.py` (or equivalent) in scheduled pipelines,
- enforce stable rule sets as release criteria,
- keep exceptions rare and documented.

129.4 Baseline management strategies

129.4.1 Strategy A: "No new warnings"

This is the most common and often the most effective:

- Accept that old warnings exist temporarily.
- Block any new warnings introduced by a change.
- Reduce old warnings with dedicated cleanup work.

129.4.2 Strategy B: "Budgeted debt reduction"

Set explicit targets:

- "reduce baseline by 5% per month",
- "remove all pointer arithmetic warnings from module X by date Y",
- "remove all C-style casts from new code immediately, legacy code by refactor milestones".

This aligns cleanup with capacity and prevents endless postponement.

129.4.3 Strategy C: "Module-by-module enforcement"

If the repository has diverse components:

- enforce strict rules on new modules first,
- progressively add older modules as they are refactored,
- isolate legacy/third-party code behind wrappers and exclude it from primary enforcement.

129.5 Choosing what to enforce first

129.5.1 A practical severity ladder

A typical ladder for Core Guidelines enforcement:

1. **Memory safety and UB risk** (bounds, lifetime patterns, uninitialized use, unsafe casts)
2. **Resource safety** (leaks, ownership confusion)
3. **Concurrency safety** (race-prone patterns, misuse of atomics/locks)
4. **Maintainability** (complexity, readability, modernization)

129.5.2 Profiles-first recommendation

The Core Guidelines emphasize profiles and tool support (e.g., `clang-tidy` rules named `cppcoreguidelines-*`) as an approach to gradual adoption. Therefore, a common and effective plan is:

- start with **Pro.bounds** and **Pro.safety**,

- add lifetime checks where confidence is acceptable,
- expand to additional guideline checks as noise decreases.

129.6 Handling warnings: suppressions, exclusions, and waiver policy

129.6.1 Prefer precise suppressions with justification

If a diagnostic is correct but intentionally accepted for a specific location, suppress narrowly and explain why. This matches Microsoft guidance on managing warnings during incremental adoption of Core Guidelines checking.

```
// NOLINTNEXTLINE(cppcoreguidelines-pro-bounds-pointer-arithmetic)
// Reason: verified ABI boundary; pointer arithmetic required by external protocol layout.
auto* p = base + offset;
```

129.6.2 Exclude third-party code deliberately

Third-party headers often generate noise and can block adoption. Use header filters so tools focus on code you own:

```
HeaderFilterRegex: '^(src|include)/'
```

Keep exclusions explicit and reviewable.

129.6.3 Waivers should be time-bound

A good engineering policy is:

- each waiver has an owner,

- each waiver has a reason,
- each waiver has an expiry date or triggering refactor milestone.

129.7 CI mechanics for gradual enforcement

129.7.1 Diff-focused enforcement for pull requests

Use diff-focused checks to prevent new debt while avoiding overwhelming reviewers. LLVM documents `clang-tidy-diff.py` specifically for this use: report diagnostics on changed lines for code review and CI.

129.7.2 Full runs for scheduled pipelines

Run full-project analyses on a schedule to:

- detect drift outside the diff,
- catch cross-cutting issues introduced indirectly,
- track baseline reduction metrics.

129.7.3 Publishing results as CI artifacts (SARIF where supported)

Many CI platforms can ingest static-analysis results in SARIF format. GitHub documents SARIF support for code scanning and states that code scanning supports a subset of the SARIF 2.1.0 schema. This enables:

- clickable findings,
- consistent triage workflows,
- historical tracking of findings.

129.8 When gradual enforcement fails (and how to fix it)

129.8.1 Failure mode: too many warnings, developers ignore CI

Fix:

- narrow the initial check set (profiles first),
- enforce “no new warnings” on touched code,
- run full checks only in scheduled pipelines until the baseline shrinks.

129.8.2 Failure mode: blanket suppressions

Fix:

- require justification and ownership for each suppression,
- forbid broad “disable all” patterns in production code,
- prefer refactors that remove the root cause.

129.8.3 Failure mode: inconsistent tool versions

Fix:

- pin compiler and tool versions in CI images,
- record versions in build logs,
- upgrade tools intentionally (with baseline refresh and review).

129.9 Summary

Gradual rule enforcement is the practical bridge between guideline intent and real-world constraints:

- the Core Guidelines explicitly support gradual adoption and define **profiles** as enforceable rule sets,
- the most effective approach is **baseline then stop regressions**,
- start with safety profiles (**Pro.bounds**, **Pro.safety**), then expand,
- use diff-focused enforcement for PRs and full runs for scheduled gates,
- keep suppressions narrow, justified, and time-bound,
- publish results as CI artifacts (SARIF where supported).

Appendices

Appendix A — Complete Rule Index (Alphabetical)

Bounds

Rule ID	Official rule title
Bounds.1	Don't use pointer arithmetic; use `span` instead
Bounds.2	Only index into arrays using constant expressions or safe abstractions
Bounds.3	No array-to-pointer decay
Bounds.4	Don't use standard-library functions and types that are not bounds-checked

Constants and Immutability

Rule ID	Official rule title
Con.1	By default, make objects immutable
Con.2	By default, make member functions `const`
Con.3	By default, pass pointers and references to `const`
Con.4	Use `const` to define objects with values that do not change after construction

Rule ID	Official rule title
Con.5	Use `constexpr` for values that can be computed at compile time

Classes

Rule ID	Official rule title
C.1	Organize related data into structures (`struct` or `class`)
C.2	Use `class` if the class has an invariant; use `struct` if it does not
C.3	Represent the distinction between interface and implementation using a class
C.4	Make a function a member only if it needs direct access to the class representation
C.5	Place helper functions in the same namespace as the class
C.6	Make copy and move operations consistent
C.7	Define a destructor if a class needs an explicit action at object destruction
C.8	Use default member initializers
C.9	Minimize exposure of members
C.10	Prefer concrete types over class hierarchies
C.11	Make base classes abstract when possible
C.12	Don’t make data members `protected`
C.13	Avoid trivial getters and setters
C.14	Use meaningful names for classes and class members
C.15	Prefer immutable data members
C.16	Use inheritance to represent hierarchical concepts
C.17	Use virtual functions only when runtime polymorphism is required
C.18	Use value semantics unless you are building a hierarchy
C.19	Avoid large class hierarchies

Rule ID	Official rule title
C.20	If a class has virtual functions, define or delete a virtual destructor
C.21	If you define or delete any copy, move, or destructor, define or delete them all
C.22	Make default operations consistent
C.23	Prefer value semantics for small objects
C.24	Prefer concrete types over polymorphic types
C.25	Prefer class hierarchies only when runtime polymorphism is essential
C.26	Minimize non-public data members
C.27	Prefer protected functions to protected data
C.28	Avoid public data members
C.29	Make interfaces stable
C.30	Define a destructor if a class manages a resource
C.31	Avoid base classes with public data
C.32	Ensure base classes have virtual destructors when used polymorphically
C.33	Prefer composition over inheritance
C.34	Avoid multiple inheritance unless necessary
C.35	Avoid diamond inheritance
C.36	Use virtual inheritance only when unavoidable

C-style Programming

Rule ID	Official rule title
CPL.1	Prefer C++ to C
CPL.2	If you must use C, use the common subset of C and C++, and compile the C code as C++
CPL.3	If you must use C for interfaces, use C++ in the calling code

Rule ID	Official rule title
CPL.4	Use `const` to define objects with fixed values
CPL.5	Avoid macros
CPL.6	Use `enum class` instead of macros for integral constants

Concurrency and Parallelism

Rule ID	Official rule title
CP.1	Assume that your code will run in a multi-threaded program
CP.2	Avoid data races
CP.3	Minimize explicit sharing of writable data
CP.4	Think in terms of tasks rather than threads
CP.5	Prefer RAII for managing locks
CP.6	Use `std::lock_guard` or `std::unique_lock` to manage locks
CP.7	Avoid `volatile` for synchronization
CP.8	Don't try to use atomic operations for complex synchronization
CP.9	Prefer higher-level concurrency abstractions
CP.10	Avoid shared mutable state
CP.11	Use condition variables correctly
CP.12	Avoid deadlocks
CP.13	Avoid blocking indefinitely
CP.14	Use `std::atomic` types where appropriate
CP.15	Prefer `std::jthread` and cooperative cancellation

Error Handling

Rule ID	Official rule title
E.1	Develop an error-handling strategy early
E.2	Throw an exception to signal an error
E.3	Use exceptions for error handling only
E.4	Design error-handling around invariants
E.5	Let a constructor establish an invariant, or throw
E.6	Use RAII to prevent leaks on error paths
E.7	State preconditions using assertions
E.8	Don’t leak resources when an exception is thrown
E.9	Use `noexcept` when failure is not an option
E.10	Prefer exceptions to error codes
E.11	Don’t catch exceptions that you cannot handle
E.12	Use error codes only when exceptions are not available
E.13	Never throw while holding a resource
E.14	Use custom exception types sparingly

Enumerations

Rule ID	Official rule title
Enum.1	Prefer scoped enumerations (`enum class`)
Enum.2	Use enumerations to represent sets of related named constants
Enum.3	Avoid unscoped enumerations
Enum.4	Define operations on enumerations explicitly
Enum.5	Don’t use `enum` for unrelated values
Enum.6	Avoid unnamed enumerations
Enum.7	Specify the underlying type of an enumeration only when necessary
Enum.8	Specify enumerator values only when necessary

Expressions and Statements

Rule ID	Official rule title
ES.1	Prefer simple and conventional expressions
ES.2	Keep expressions short and clear
ES.3	Avoid complicated expressions
ES.4	Use parentheses to clarify operator precedence
ES.5	Avoid implicit narrowing conversions
ES.6	Avoid mixing signed and unsigned arithmetic
ES.7	Keep scopes small
ES.8	Use range-based `for` loops
ES.9	Avoid `goto`
ES.10	Declare variables in the smallest possible scope
ES.11	Use `auto` to avoid redundant type names
ES.12	Don't reuse variable names in nested scopes
ES.13	Avoid unnecessary temporary objects
ES.14	Minimize the use of macros
ES.15	Prefer `constexpr` expressions where possible
ES.16	Avoid macros for constants and functions
ES.17	Avoid macros for control flow
ES.18	Prefer `constexpr` and inline functions to macros
ES.19	Avoid macros that hide control flow
ES.20	Avoid macros that define or manipulate variables
ES.21	Use `constexpr` instead of macros when possible
ES.22	Avoid macros with arguments
ES.23	Avoid macros for code generation
ES.24	Prefer inline functions to macros

Rule ID	Official rule title
ES.25	Avoid macros that expand to multiple statements
ES.26	Avoid macros that introduce names into global scope
ES.27	Avoid macros that depend on argument evaluation order
ES.28	Avoid macros for type safety
ES.29	Avoid macros for debugging facilities
ES.30	Avoid macros
ES.31	Avoid macros for functions
ES.32	Avoid macros with all-capital names
ES.33	Avoid macros for symbolic constants
ES.34	Don’t define C-style variadic functions

Functions

Rule ID	Official rule title
F.1	Package meaningful operations as functions
F.2	A function should perform a single logical operation
F.3	Keep functions short and simple
F.4	If a function may have to be evaluated at compile time, declare it `constexpr`
F.5	Prefer passing arguments by value or reference, not pointer
F.6	Prefer pass-by-`const` reference for large objects
F.7	For general use, take `T*` or `T&` arguments rather than smart pointers
F.8	Prefer pure functions
F.9	Avoid `nullptr` checks inside functions
F.10	If a function is expensive to call, consider caching its result
F.11	Minimize the number of function parameters

Rule ID	Official rule title
F.12	Use default arguments sparingly
F.13	Avoid C-style variadic functions
F.14	Use `std::span` or ranges instead of pointer/length pairs
F.15	Prefer returning values to output parameters
F.16	For *in* parameters, pass cheaply-copied types by value and others by reference to `const`
F.17	For *in-out* parameters, pass by reference to non-`const`
F.18	For *out* parameters, prefer return values
F.19	Avoid returning pointers or references to local objects
F.20	For multiple return values, use a struct or tuple
F.21	To return multiple “out” values, prefer returning a struct
F.22	Use `auto` return types for generic lambdas and templates
F.23	Keep functions free of side effects when possible
F.24	Prefer non-member functions when they do not need access to representation
F.25	Use function overloading only when semantics are identical
F.26	Avoid overloading operators with surprising semantics
F.27	Prefer named functions to overloaded operators when meaning is unclear
F.28	Avoid functions with multiple responsibilities
F.29	Avoid macros for function-like behavior
F.30	Avoid functions that depend on hidden global state
F.31	Prefer lambdas to function objects for short callbacks
F.32	Use generic lambdas where appropriate
F.33	Use `std::function` only when type erasure is required
F.34	Avoid unnecessary dynamic allocation in function interfaces
F.35	Use `noexcept` to express non-throwing functions

Rule ID	Official rule title
F.36	Avoid throwing exceptions from destructors
F.37	Declare functions `noexcept` when failure is not an option
F.38	Use `[[nodiscard]]` to prevent ignored return values
F.39	Avoid returning error codes and results through the same channel
F.40	Use `std::expected` or `std::optional` to represent optional results
F.41	Prefer structured bindings with tuple-like returns
F.42	Prefer algorithms over explicit loops in functions
F.43	Avoid recursion for performance-critical paths
F.44	Mark overrides with `override`
F.45	Mark final overrides with `final`
F.46	Avoid default arguments on virtual functions
F.47	Avoid exposing implementation details through function interfaces
F.48	Prefer overload sets to default parameters when semantics differ
F.49	Avoid functions that both query and modify state
F.50	Prefer named return types for complex results
F.51	Avoid functions with ambiguous ownership semantics
F.52	Document ownership transfer in function interfaces
F.53	Prefer value-returning factory functions
F.54	Avoid raw owning pointers in interfaces
F.55	Use smart pointers to express ownership transfer
F.56	Avoid implicit ownership transfer through parameters
F.57	Use `std::move` only when ownership transfer is intended
F.58	Avoid returning smart pointers when value semantics suffice
F.59	Avoid unnecessary heap allocation in hot paths
F.60	Prefer clear, explicit interfaces over clever ones

Interfaces

Rule ID	Official rule title
I.1	Make interfaces explicit
I.2	Avoid global variables
I.3	Avoid non-`const` global data
I.4	Make interfaces resource-safe
I.5	Prefer small, stable interfaces
I.6	Avoid implicit dependencies
I.7	Use abstract base classes as interfaces
I.8	Use namespaces to express logical structure
I.9	Avoid exposing implementation details in interfaces
I.10	Prefer abstract interfaces to concrete base classes
I.11	Use pure abstract classes for interfaces
I.12	Separate interface from implementation
I.13	Avoid fat interfaces
I.14	Make interfaces easy to use correctly
I.15	Avoid requiring callers to manage resources
I.16	Prefer explicit ownership in interfaces
I.17	Avoid exposing raw pointers across interfaces
I.18	Prefer passing objects by reference or value
I.19	Avoid ambiguous ownership transfer
I.20	Document lifetime expectations in interfaces
I.21	Avoid exposing mutable global state
I.22	Use `span` to represent ranges in interfaces
I.23	Avoid exposing containers when a range suffices
I.24	Prefer value types in interfaces

Rule ID	Official rule title
I.25	Prefer empty abstract classes as interfaces
I.26	If you want a cross-compiler ABI, use a C-style subset
I.27	Avoid exposing templates across binary interfaces
I.28	Use pimpl to isolate unstable implementation details
I.29	Prefer stable interfaces for long-lived libraries
I.30	Encapsulate rule violations

Philosophy

Rule ID	Official rule title
P.1	Express ideas directly in code
P.2	Write in ISO Standard C++
P.3	Express intent
P.4	Ideally, a program should be statically type safe
P.5	Prefer compile-time checking to run-time checking
P.6	What cannot be checked at compile time should be checkable at run time
P.7	Catch run-time errors early
P.8	Don’t leak any resources
P.9	Don’t waste time or space
P.10	Prefer immutable data to mutable data
P.11	Encapsulate messy constructs
P.12	Use supporting tools as appropriate
P.13	Use support libraries as appropriate

Performance

Rule ID	Official rule title
Per.1	Don’t optimize prematurely
Per.2	Measure before you optimize
Per.3	Don’t assume that low-level code is faster
Per.4	Don’t guess about performance
Per.5	Prefer simple, clear code
Per.6	Avoid unnecessary work
Per.7	Use the right data structures
Per.8	Minimize allocations
Per.9	Avoid unnecessary copying
Per.10	Cache results when appropriate
Per.11	Avoid hidden expensive operations
Per.12	Use move semantics where appropriate
Per.13	Avoid virtual dispatch in hot paths
Per.14	Avoid unnecessary indirection
Per.15	Inline small functions when beneficial
Per.16	Prefer contiguous data layouts
Per.17	Avoid pessimizing abstractions
Per.18	Avoid unnecessary synchronization
Per.19	Minimize contention
Per.20	Prefer algorithms with better complexity
Per.21	Use parallelism judiciously

Resource Management

Rule ID	Official rule title
R.1	Manage resources automatically using RAII

Rule ID	Official rule title
R.2	Prefer scoped objects to manual resource management
R.3	Avoid raw owning pointers
R.4	Use smart pointers to represent ownership
R.5	Prefer `unique_ptr` to `shared_ptr`
R.6	Avoid `shared_ptr` cycles
R.7	Use `weak_ptr` to break cycles
R.8	Don’t leak resources
R.9	Avoid manual memory management
R.10	Avoid calling `new` and `delete` explicitly
R.11	Avoid ownership ambiguity
R.12	Prefer objects that manage their own resources
R.13	Use factory functions instead of raw allocation
R.14	Use RAII wrappers for all resources
R.15	Avoid resource leaks in error paths
R.16	Avoid throwing exceptions from destructors
R.17	Use `noexcept` destructors where appropriate

Standard Library

Rule ID	Official rule title
SL.1	Use the standard library
SL.2	Prefer standard library facilities to hand-crafted code
SL.3	Avoid reinventing standard library facilities
SL.4	Prefer algorithms to raw loops
SL.5	Prefer `std::array` and `std::vector` to C arrays
SL.6	Prefer `std::string` to C-style strings

Rule ID	Official rule title
SL.7	Prefer `std::span` for non-owning ranges
SL.8	Use `std::optional` for optional values
SL.9	Use `std::variant` for type-safe unions
SL.10	Use `std::expected` for value-or-error results

Templates and Generic Programming

Rule ID	Official rule title
T.1	Use templates to express algorithms that apply to many types
T.2	Keep templates simple and readable
T.3	Use templates only when they provide clear benefit
T.4	Use concepts to express template requirements
T.5	Prefer readable constraints to SFINAE
T.6	Avoid excessive template metaprogramming
T.7	Prefer compile-time checking in templates
T.8	Use `requires` clauses to document intent
T.9	Avoid unconstrained templates
T.10	Minimize template instantiation dependencies

Source Files

Rule ID	Official rule title
SF.1	Use `.cpp` files for implementation and `.h` files for interfaces
SF.2	A `.cpp` file must include its own header first
SF.3	Use include guards for all header files
SF.4	Avoid cyclic dependencies among source files

Rule ID	Official rule title
SF.5	Avoid dependencies on implicitly included names
SF.6	Avoid using-directives at global scope
SF.7	Don't write `using namespace` at global scope in a header file
SF.8	Use include guards for all header files
SF.9	Avoid cyclic dependencies among source files
SF.10	Avoid dependencies on implicitly included names

Profiles

Profile ID	Profile description
Pro.bounds	Bounds safety profile
Pro.lifetime	Lifetime safety profile
Pro.type	Type safety profile

Type Safety Profile

Rule ID	Official rule title
Type.1	Avoid casts
Type.2	Don't use `static_cast` to downcast
Type.3	Don't use `const_cast`
Type.4	Avoid C-style casts
Type.5	Don't use a variable before it has been initialized
Type.6	Always initialize objects
Type.7	Avoid naked unions
Type.8	Avoid variadic arguments

Lifetime Safety Profile

Rule ID	Official rule title
Lifetime.1	Don’t dereference a possibly invalid pointer
Lifetime.2	Don’t access an object after it has been destroyed
Lifetime.3	Avoid dangling references
Lifetime.4	Don’t return pointers or references to local objects

129.10 Appendix B — Rule Classification by Risk

129.10.1 What “Risk” Means in This Manual

The ISO C++ Core Guidelines are written to help developers avoid classes of defects and to enable gradual adoption using *profiles*. A profile is defined as a coherent, deterministic, and portably enforceable subset of rules designed to achieve a specific safety guarantee, such as absence of range errors (bounds), static type safety (type), or lifetime safety. This appendix classifies rules by **risk of harm when violated**.

Important: The Core Guidelines do not publish an official universal “risk ranking” for every rule. Therefore, the classification below is an *editorial framework for this book* that stays faithful to the safety goals explicitly described by the Guidelines, especially the safety profiles and their targeted guarantees.

129.10.2 Risk Levels

- **Critical** (Must-fix): likely to cause Undefined Behavior (UB), memory corruption, data races, or security vulnerabilities.
- **High**: likely to cause crashes, leaks, serious correctness failures, or hard-to-debug concurrency issues.

- **Medium**: can cause incorrect results, maintainability hazards, or performance regressions with moderate impact.
- **Low**: mainly readability, style consistency, or local quality improvements.
- **Context-dependent**: risk depends strongly on domain constraints (embedded, hard real-time, safety-certified, etc.).

129.10.3 Fast Classification Heuristics

1. If a rule is part of a safety profile intended to guarantee absence of a class of safety errors (bounds, type, lifetime), treat violations as **Critical** by default.
2. If a rule is about avoiding data races, treat violations as **Critical** because data races lead to undefined behavior in C++.
3. If a rule concerns resource ownership, leaks, or destructor behavior in failure paths, treat as **High** or **Critical** depending on whether it can lead to UB or security impact.
4. If a rule primarily affects clarity, readability, or interface stability, treat as **Medium** or **Low** unless it also affects safety.

129.10.4 Classification by Rule Family

Rule family	**Default risk**	**Reasoning aligned with the Guidelines**
Bounds.* (Bounds safety profile)	Critical	Targets absence of range errors; profile conformance is intended to make code safe by construction for that property.

Rule family	Default risk	Reasoning aligned with the Guidelines
Type.* (Type safety profile)	Critical	Targets type correctness and avoidance of unsafe casts/unions; intended safety becomes complete when combined with Bounds and Lifetime.
Lifetime.* (Lifetime safety profile)	Critical	Targets avoidance of dangling pointers/references and invalid dereferences; strongly correlated with UB.
CP.* (Concurrency and parallelism)	Critical / High	Concurrency rules emphasize avoiding data races and unsafe shared writable state; data races are explicitly highlighted as hazards.
R.* (Resource management)	High	Aims to prevent leaks and mismanaged ownership; violations commonly cause leaks, double frees, lifetime bugs, and failure-mode hazards.
E.* (Error handling)	High / Medium	Error-handling strategy and exception safety strongly influence correctness in failure paths; some violations can escalate to resource leaks or partial invariants.
I.* (Interfaces)	Medium / High	Interface rules reduce misuse and lifetime/ownership ambiguity at boundaries; risk increases when ownership/lifetime are unclear across components.
C.* (Classes)	Medium / High	Class invariants, ownership members, and special member functions affect correctness and lifetime; risk increases when ownership is unclear.

Rule family	Default risk	Reasoning aligned with the Guidelines
F.* (Functions)	Medium	Function rules improve clarity and correct parameter passing; risk rises when ownership/lifetime or concurrency concerns are involved.
ES.* (Expressions and statements)	Medium / High	Many expression rules prevent subtle bugs (narrowing, signed/unsigned, order-of-evaluation); some violations can lead to UB or security issues in low-level code.
Enum.* (Enumerations)	Low / Medium	Mostly improves type safety and clarity; risks are usually local and visible.
Per.* (Performance)	Context-dependent	Performance rules matter most in latency/throughput/real-time systems; incorrect "optimizations" can also damage correctness indirectly.
SL.* (Standard library)	Low / Medium	Promotes safe and well-tested components; risk is typically lower unless replacing unsafe raw operations.
CPL.* (C-style)	Medium / High	C-style constructs often bypass safety checks and abstraction; risk increases in low-level and boundary code.
P.* (Philosophy)	Low / Medium	Guiding principles; violations mostly create systemic design debt rather than immediate bugs.

129.10.5 Critical Risk Rules (Must-fix First)

The following rules are prioritized because they directly target memory safety, type safety, lifetime safety, and data-race freedom.

129.10.5.1 Bounds Safety (Bounds.*)

- **Bounds.1–Bounds.4** (bounds safety profile): avoid pointer arithmetic, prevent unsafe indexing and decay, and avoid unbounded library facilities.

129.10.5.2 Type Safety (Type.*)

Type safety profile rules explicitly focus on removing primary sources of type violations (unsafe casts, unions), and are intended to combine with bounds and lifetime profiles for comprehensive safety.

129.10.5.3 Lifetime Safety (Lifetime.*)

- Treat all lifetime-profile violations as Critical because they commonly produce dangling pointers/references and invalid dereferences.

129.10.5.4 Concurrency (CP.*)

- **CP.2 Avoid data races** (Critical). Data races are explicitly called out as a key hazard in the concurrency rules.

- **CP.1** is a forcing function: code can later be reused in multi-threaded contexts and shared statics can create races.

129.10.6 High Risk Rules (Fix Early)

These rules often do not *guarantee* UB by themselves, but strongly correlate with leaks, crashes, corrupted program state, or unmaintainable failure handling.

129.10.6.1 Ownership and Resource Management

A key theme is avoiding unclear ownership, especially raw owning pointers and raw resource handles. The Guidelines emphasize that ownership should be clear and that legacy raw pointers may require analysis to determine ownership.

129.10.6.2 Failure Paths and Cleanup

Resource leaks are especially dangerous in error cases; RAII is repeatedly recommended to keep cleanup reliable.

129.10.7 Examples: Why These Risks Are Classified This Way

129.10.7.1 Critical: Out-of-bounds write (Bounds safety)

```
#include <vector>
#include <cstddef>

void smash(std::vector<int>& v) {
    // If v is empty, v[0] is out-of-bounds: memory corruption / UB.
    v[0] = 42;
}
```

129.10.7.2 Critical: Data race by shared writable state (CP.2)

The Guidelines warn about races caused by shared state that can be accessed concurrently.

```
#include <thread>
#include <iostream>

static int shared_counter = 0; // shared writable state
```

```
void bump() {
    for (int i = 0; i < 1'000'000; ++i) {
        ++shared_counter; // data race => undefined behavior
    }
}

int main() {
    std::thread t1(bump);
    std::thread t2(bump);
    t1.join(); t2.join();
    std::cout << shared_counter << "\n";
}
```

129.10.7.3 High: Ambiguous ownership via raw pointer member

The Guidelines note that raw member pointers can be vague about ownership and require analysis; ownership should be made clear, typically using proper ownership expression.

```
struct Widget {
    int* p; // Is this owning? observer? unclear => high risk
    ~Widget() {
        // If we delete, we might double-delete.
        // If we don't, we might leak.
    }
};
```

129.10.8 Recommended Adoption Order (Risk-Driven)

This adoption order aligns with the profiles concept: apply coherent sets of rules to eliminate a class of errors first.

1. **Bounds safety** (Bounds.*) and **Lifetime safety** (Lifetime.*)
2. **Type safety** (Type.*)
3. **Concurrency** (CP.*) with emphasis on data-race freedom
4. **Resource management** (R.*), then **Error handling** (E.*)
5. **Interfaces / Classes / Functions / Expressions** (I.*, C.*, F.*, ES.*)
6. **Library usage and performance guidance** (SL.*, Per.*) as needed by domain constraints

129.10.9 Practical Enforcement Notes

Profiles are designed to be enforceable by tools; enabling a profile and keeping code warning-free is treated as conforming for that property. Therefore:

- Treat **Critical** rules as "no-exceptions" unless there is a documented and reviewed suppression.
- Treat **High** rules as "fix before release" in production code.
- Treat **Medium/Low** rules as "continuous improvement" with tooling and code review.

129.11 Appendix C — Comparison with MISRA and AUTOSAR

129.11.1 Scope and Purpose

This appendix compares three widely used guidance sets for C++ code quality and safety:

- **ISO C++ Core Guidelines** (the Guidelines): general-purpose rules and best practices aimed at modern C++ codebases, including safety profiles (type, bounds, lifetime) and guidance for gradual adoption. *Profiles* are defined as deterministic, portably enforceable subsets of rules designed to achieve a specific safety guarantee.
- **MISRA C++**: a safety-focused guideline set for critical systems. The latest published edition is MISRA C++:2023, targeting C++17.
- **AUTOSAR C++14**: a coding guideline set for safety-related and critical environments, specifying rules for using C++14 as defined by ISO/IEC 14882:2014. It is explicitly described as an update of MISRA C++:2008, informed by other standards and AUTOSAR analysis.

129.11.2 High-Level Comparison

Dimension	**Core Guidelines**	**MISRA C++:2023**	**AUTOSAR C++14**
Primary goal	Safer, clearer, modern C++ with practical guidance and tool-enforceable profiles	High-integrity subset and discipline for critical systems using C++17	Rules for safety-related and critical environments using C++14
Language baseline	Modern ISO C++ (Guidelines evolve; profiles emphasize enforceability)	C++17 target	C++14 as defined by ISO/IEC 14882:2014
Domain focus	General purpose; applicable across industries	Safety-critical and high-integrity systems	Automotive-centric, but stated as usable in other embedded sectors

Dimension	Core Guidelines	MISRA C++:2023	AUTOSAR C++14
Structure and enforcement	Rules plus *profiles* designed for deterministic, portable enforcement; warning-free profile conformance implies safety-by-construction for targeted properties	Typically strict compliance culture with deviation handling (process-driven)	Rule set includes compliance metadata such as required/advisory; rules are enumerated and designed for tool checking
Relationship between MISRA and AUTOSAR	Independent, but overlaps strongly in safety aims	Many AUTOSAR C++14 guidelines are reflected in MISRA C++:2023; MISRA focuses on C++17 while AUTOSAR is automotive-oriented	Explicitly described as an update of MISRA C++:2008 for C++14 in critical environments

129.11.3 How to Read Differences Correctly

Do not interpret differences as contradictions. These documents often share the same safety goals but differ in:

- **Audience and certification expectations** (general engineering vs regulated high-integrity processes).
- **Language version baseline** (C++14 vs C++17 and evolving modern C++).
- **Strictness and permitted feature set** (especially regarding dynamic polymorphism, exceptions, and dynamic allocation).

- **Governance and compliance process** (deviations, documentation, tool qualification).

129.11.4 Key Common Ground (Broad Overlap Areas)

Across all three, the following themes are consistently emphasized:

- Avoid undefined behavior and fragile low-level constructs.
- Make ownership and lifetime expectations clear.
- Restrict or tightly control casts, unions, and type punning (type safety focus).
- Prefer deterministic behavior and analyzability (tool enforcement is central in profiles and safety standards).
- Minimize global writable state and concurrency hazards (especially data races).

129.11.5 Typical Differences You Will See in Practice

129.11.5.1 Profiles versus Compliance Regimes

The Core Guidelines define **profiles** as deterministic, portably enforceable subsets of rules intended to deliver a specific guarantee, and treat warning-free profile conformance as safe-by-construction for the targeted property. This is conceptually different from the typical compliance posture in MISRA and AUTOSAR, which is often **process-driven** and relies on: policy, documentation, controlled deviations, and tool configurations.

129.11.5.2 Dynamic RTTI and `dynamic_cast`

AUTOSAR C++14 explicitly discourages `dynamic_cast` and gives a real-time suitability rationale. The Core Guidelines also tend to prefer designs that avoid unnecessary runtime type checks, but their framing is generally broader: clarity, maintainability, and cost, rather than automotive real-time constraints.

129.11.5.3 Exceptions and Failure Handling

AUTOSAR C++14 includes explicit rules about catching exceptions (including unexpected ones) to prevent immediate termination in environments where that is unacceptable. The Core Guidelines also discuss exception safety and careful use of `noexcept` (especially for low-level operations and destructors).

129.11.5.4 Language Version Pressure

MISRA C++:2023 targets C++17, while AUTOSAR C++14 is based on the C++14 language standard (ISO/IEC 14882:2014). This affects which features are available and how rules are formulated (for example, modern library types, newer attributes, and concurrency primitives).

129.11.6 Practical Mapping: When to Prefer Which Guidance

Situation	Recommended approach
Safety-certified automotive software or supplier codebase	Use AUTOSAR C++14 (or the applicable AUTOSAR/MISRA edition required by the project) as the baseline compliance standard; layer Core Guidelines as additional engineering guidance where it does not conflict with mandatory compliance.
High-integrity systems outside automotive	Use MISRA C++:2023 as the baseline discipline (language subset and compliance culture), then add Core Guidelines profiles (bounds, lifetime, type) where tooling allows.
Modern large-scale C++ products (general domain)	Use Core Guidelines as the primary rule set, and use profiles to drive measurable safety improvements first (bounds, lifetime, type).

Situation	Recommended approach
Mixed legacy code with gradual modernization	Start with a small, coherent set of enforceable constraints (profiles) to reduce disruption; this is explicitly the motivation for profiles.

129.11.7 Illustrative Examples

129.11.7.1 Example 1: RTTI and `dynamic_cast`

AUTOSAR C++14 includes an advisory rule stating that `dynamic_cast` should not be used, with a rationale tied to suitability in real-time environments.

```
struct Base { virtual ~Base() = default; };
struct Derived : Base { void f() {} };

void use(Base* b) {
    // Many safety standards discourage RTTI-based control flow in critical code.
    if (auto* d = dynamic_cast<Derived*>(b)) {
        d->f();
    }
}
```

129.11.7.2 Example 2: Exception boundaries and termination risk

AUTOSAR C++14 shows examples where uncaught unexpected exceptions can lead to immediate termination and therefore requires catching unexpected exceptions in certain contexts.

```
#include <thread>
#include <stdexcept>
```

```
void worker() {
    // If this escapes a thread entry point, the program may terminate.
    throw std::runtime_error("unexpected");
}

int main() {
    std::thread t(worker);
    t.join();
}
```

In Core Guidelines style, you would also be careful about exception guarantees and where `noexcept` is appropriate, especially for low-level operations and destructors.

129.11.7.3 Example 3: Ownership clarity versus raw owning pointers

The Core Guidelines emphasize classifying pointers as non-owners by default and making ownership explicit where relevant.

```
struct Buffer {
    int* p;      // ownership unclear: might own or might observe
    int  n;

    ~Buffer() {
        // Deleting here might double-delete.
        // Not deleting might leak.
    }
};
```

In high-integrity environments, MISRA/AUTOSAR compliance processes typically require making this ownership intent explicit, and frequently discourage or tightly control raw owning pointers and unmanaged allocation patterns.

129.11.8 Guidance for Using the Three Together

A practical, low-risk integration strategy is:

1. Use the project-mandated standard (MISRA or AUTOSAR) as the **hard compliance baseline** when required by contract, regulation, or certification.

2. Add Core Guidelines **profiles** (bounds, lifetime, type) to drive measurable safety guarantees that are designed for deterministic, portable enforcement.

3. Document deviations explicitly and keep the deviation set small and reviewed (especially for Critical-risk areas such as concurrency, bounds, and lifetime).

4. Prefer engineering designs that reduce the need for dangerous features, rather than relying only on rule exceptions (for example, avoid RTTI-based control flow where feasible in hard real-time components).

129.12 Appendix D — When and Why to Break a Rule

129.12.1 Purpose of This Appendix

The C++ Core Guidelines are intentionally ambitious: following them leads to safer, cleaner, and often faster code, but it is often *impossible* to apply all rules everywhere in large, long-lived systems. The Guidelines explicitly acknowledge that some rules may be too strict and that more exceptions may be needed to accommodate real-world needs.
This appendix provides a practical, engineering-focused method for deciding *when* breaking a rule can be justified, *why* it is done, and *how* to contain the risk.

129.12.2 Key Principles

1. **Rules are defaults, not dogma.** The default is to follow the rule; exceptions exist for compelling constraints.

2. **Break a rule only to satisfy a higher priority engineering requirement,** such as correctness, safety, portability, performance constraints, certification requirements, or compatibility with existing code.

3. **Make exceptions explicit, local, and reviewable.** If you cannot make the exception explicit and confined, you likely should not make it.

4. **Prefer rule-conforming designs first.** If a conforming alternative exists with acceptable cost, use it.

5. **Treat safety profiles as a boundary.** Profiles are defined as deterministic, portably enforceable subsets designed to provide a specific safety guarantee; warning-free code under a profile is considered safe by construction for that property.

129.12.3 A Decision Checklist (Use Before Any Rule Exception)

Answer **all** of the following:

1. **Which rule is being broken?** Identify the rule ID precisely (e.g., `CP.100`, `Bounds.1`).

2. **What is the constraint?** (e.g., ABI compatibility, hard real-time deadline, platform limitation, legacy interface.)

3. **What is the risk of violation?** If it can introduce undefined behavior, memory corruption, or data races, treat the exception as critical and require stronger justification and containment.

4. **What is the conforming alternative?** Explain why it is rejected (measured performance, unacceptable redesign cost, etc.).

5. **How is the exception contained?** (local scope, wrapper, private module boundary, narrow API surface.)

6. **How is it verified?** (tests, static analysis, fuzzing, sanitizers, formal review, tool qualification.)

7. **How will it be removed later?** If possible, add a migration plan and a tracking issue.

129.12.4 Legitimate Reasons to Break a Rule

These are common, defensible reasons in professional C++ systems.

129.12.4.1 Interoperability and ABI Constraints

- Interfacing with a C API, OS API, or third-party binary interface that cannot be changed.
- Stable ABI requirements that prevent adopting certain library types or templates across boundaries.
- A legacy codebase where a gradual adoption strategy is required.

129.12.4.2 Hard Real-Time and Resource Constraints

Some applications have constraints (bounded latency, deterministic timing, tight memory footprint) that change trade-offs. The Guidelines explicitly note that there are specialized applications where some otherwise-preferred techniques can be inappropriate, such as exceptions in hard-real-time systems without reliable cost estimates.

129.12.4.3 Performance-Driven Exceptions (With Evidence)

A rule exception for performance must be backed by:

- repeatable measurement,
- representative workload,
- and a clear explanation of why the conforming alternative fails.

Never use “performance” as a vague justification.

129.12.4.4 Low-Level Mechanisms Hidden Behind a Safe Interface

Sometimes low-level operations are necessary (raw buffers, bit manipulation, system calls). The acceptable pattern is:

- keep the unsafe code small,
- isolate it in one module,
- and expose a safe, rule-conforming API to the rest of the system.

129.12.4.5 Tooling Limitations and Transitional Periods

The Guidelines emphasize gradual adoption and the importance of applying coherent sets of rules rather than enforcing “random” rules in isolation. If tooling cannot yet enforce a rule reliably across your environment, you may temporarily defer it, but you should record this explicitly.

129.12.5 How to Break a Rule Safely

129.12.5.1 Make the Exception Explicit (Suppressions for Profile Checks)

Profiles are meant to be enabled in tools; to suppress enforcement of a *profile* check, the Guidelines show using a `[[suppress("...")]]` annotation on a language contract and state suppression should be very rare.

```
[[suppress("bounds")]] char* raw_find(char* p, int n, char x) // deliberate
↪  bounds-risk function
{
    // Low-level scan over p[0]..p[n-1]; caller must ensure invariants.
    // Keep this implementation small, tested, and isolated.
    for (int i = 0; i < n; ++i) {
        if (p[i] == x) return p + i;
    }
    return nullptr;
}
```

129.12.5.2 Keep the Violation Local and Contained

Prefer patterns like:

- one translation unit dedicated to platform code,
- a small internal namespace,
- and a narrow interface returning safe types.

129.12.5.3 Add Guardrails

When you must violate a rule, add compensating controls:

- **Preconditions** and assertions at the boundary,

- **Unit tests** including edge cases,
- **Sanitizers** in CI (AddressSanitizer/UBSan/TSan when applicable),
- **Fuzzing** for parsers and untrusted inputs,
- **Static analysis** configured consistently across builds.

129.12.6 A Concrete Example: Breaking a Concurrency Rule

The Guidelines warn that lock-free programming is error-prone and give a rule explicitly stating not to use lock-free programming unless you absolutely have to (CP.100), while also describing an exception where atomic variables can be used simply and safely under a strong memory model assumption.

129.12.6.1 When It Can Be Justified

- There is a proven contention bottleneck.
- A well-studied lock-free structure is required and cannot be replaced with standard locks.
- You have expert review and extensive testing.

129.12.6.2 How to Contain the Risk

- Prefer a well-tested library implementation rather than hand-rolled primitives (the Guidelines explicitly recommend this alternative).
- Freeze toolchain and re-test on any change in hardware, OS, compiler, or library (CP.101 enforcement guidance).

129.12.7 Documenting a Rule Exception (Template)

Use this exact record format in code review or architecture decisions:

> **Rule Exception Record**
>
> **Rule ID(s):**
> **Location (file/function):**
> **Reason (constraint):**
> **Risk (what can go wrong):**
> **Rejected conforming alternatives:**
> **Containment strategy:**
> **Verification plan:**
> **Removal plan / revisit date:**
> **Reviewer approval:**

129.12.8 When You Should *Not* Break a Rule

Breaking a rule is not justified when:

- the motivation is convenience or speed of writing code,
- you cannot articulate and measure the constraint,
- the exception spreads through the codebase (becoming "the new normal"),
- or the rule relates to safety guarantees (bounds/type/lifetime) and you cannot provide strong containment and verification.

129.12.9 Recommended Policy for This Book

1. Treat safety-profile violations (Bounds/Type/Lifetime) as **exceptional events** requiring formal justification, explicit containment, and strong verification.

2. Allow exceptions primarily at boundaries (platform/ABI/legacy interop) and in isolated low-level modules.
3. Require a Rule Exception Record for every deliberate violation.
4. Prefer making exceptions tool-visible using suppressions when profile checks are enabled, and keep suppressions rare.
5. Revisit exceptions periodically; remove them when constraints no longer apply.

129.13 Appendix E — Real-World Industrial Examples

129.13.1 What This Appendix Covers

This appendix provides realistic industrial-style scenarios and shows how to apply the ISO C++ Core Guidelines patterns to reduce risk, improve correctness, and enable tool enforcement. The Guidelines explicitly recognize that following all rules everywhere is often impossible in large, long-lived systems, and therefore recommend *gradual adoption* using coherent sets of related rules called *profiles*. A profile is defined as a deterministic and portably enforceable subset of rules designed to achieve a specific safety guarantee, and warning-free code under the profile is considered safe by construction for the targeted property.

129.13.2 Reading Guide

Each example includes:

- **Scenario:** a common industry problem.
- **Primary risks:** what tends to go wrong.
- **Guideline-aligned approach:** the safe design strategy.

- **Code sketch:** minimal code illustrating the pattern.

129.13.3 Example 1 — Buffer Parsing in Embedded / Devices (Bounds and Type Safety)

129.13.3.1 Scenario

A device receives binary frames over UART/SPI. A parser must validate the frame and extract fields.

129.13.3.2 Primary risks

Out-of-bounds reads/writes, pointer arithmetic mistakes, and implicit assumptions about lengths.

129.13.3.3 Guideline-aligned approach

Represent buffers as a span-like view rather than raw pointer arithmetic, and keep indexing guarded. The Guidelines note that pointer arithmetic is best done within spans and that a buffer pointer should be represented by a span.

129.13.3.4 Code sketch

```
#include <cstdint>
#include <span>
#include <optional>

struct Frame {
    std::uint8_t type{};
    std::uint16_t value{};
};
```

```
// returns std::nullopt if frame is invalid
std::optional<Frame> parse_frame(std::span<const std::uint8_t> bytes)
{
    // Minimal protocol: [0]=type, [1..2]=value (little-endian)
    if (bytes.size() < 3) return std::nullopt;

    Frame f;
    f.type  = bytes[0];
    f.value = static_cast<std::uint16_t>(bytes[1])
            | (static_cast<std::uint16_t>(bytes[2]) << 8);

    return f;
}
```

129.13.4 Example 2 — Preventing Hidden Data Races in a Library (Concurrency)

129.13.4.1 Scenario

A reusable library adds caching for performance. It uses function-local statics.

129.13.4.2 Primary risks

Data races and undefined behavior when the library is used from multiple threads. The Guidelines show that shared static caching can introduce data races and thus undefined behavior.

129.13.4.3 Guideline-aligned approach

Assume the code will be used in a multi-threaded program (CP.1), and avoid data races (CP.2). Move the cache into an object so the caller controls sharing.

129.13.4.4 Code sketch

```
#include <cstdint>

constexpr double COMPUTATION_OF_ZERO = 0.0;

double computation(int x);

struct ComputationCache {
    int cached_x = 0;
    double cached_result = COMPUTATION_OF_ZERO;

    double compute(int x) {
        if (cached_x != x) {
            cached_x = x;
            cached_result = computation(x);
        }
        return cached_result;
    }
};
```

129.13.5 Example 3 — Ownership at ABI Boundaries (Libraries and Plugins)

129.13.5.1 Scenario

A plugin system uses a stable C ABI. Objects are allocated inside the plugin and freed by the plugin.

129.13.5.2 Primary risks

Ownership ambiguity, leaks, double-free, and lifetime bugs when raw pointers cross a boundary.

129.13.5.3 Guideline-aligned approach

Treat raw pointers as non-owners by default and make ownership explicit when it exists. The Guidelines state that raw pointer notation is assumed to point but not own, and that owners should be converted to resource handles or marked `owner<T*>`. They also list common reasons why `owner<T*>` exists, including ABI constraints.

129.13.5.4 Code sketch

```
#include <cstddef>

extern "C" {

// Opaque handle type for ABI stability
struct PluginObject;

// Factory + destroy functions define ownership explicitly.
PluginObject* plugin_create();
void plugin_destroy(PluginObject*);

} // extern "C"
```

```
// C++ wrapper providing RAII to the rest of the program
class PluginHandle {
    PluginObject* p = nullptr;
public:
    explicit PluginHandle(PluginObject* raw) noexcept : p(raw) {}
    PluginHandle(const PluginHandle&) = delete;
    PluginHandle& operator=(const PluginHandle&) = delete;

    PluginHandle(PluginHandle&& other) noexcept : p(other.p) { other.p = nullptr; }
    PluginHandle& operator=(PluginHandle&& other) noexcept {
        if (this != &other) {
            reset();
            p = other.p;
            other.p = nullptr;
        }
        return *this;
    }

    ~PluginHandle() { reset(); }

    void reset() noexcept {
        if (p) { plugin_destroy(p); p = nullptr; }
    }

    PluginObject* get() const noexcept { return p; } // non-owning view
};
```

129.13.6 Example 4 — Smart Pointer Parameters That Communicate Intent (API Design)

129.13.6.1 Scenario

A subsystem exposes APIs that sometimes take ownership, sometimes only observe.

129.13.6.2 Primary risks

Accidentally pessimizing performance, unclear lifetime expectations, and accidental ownership transfer.

129.13.6.3 Guideline-aligned approach

Take smart pointers as parameters only to explicitly express lifetime semantics (R.30), and use `unique_ptr` parameters to express ownership transfer (R.32).

129.13.6.4 Code sketch

```
#include <memory>

struct Widget { void foo(); };

// observes; does not participate in ownership
void use(Widget& w) { w.foo(); }

// takes ownership (caller must std::move)
void sink(std::unique_ptr<Widget> w) { w->foo(); }
```

129.13.7 Example 5 — Preventing Undefined Behavior in Class Hierarchies (Virtual Destructors)

129.13.7.1 Scenario

A framework stores derived objects via base pointers.

129.13.7.2 Primary risks

Deleting through a base pointer without a virtual destructor leads to undefined behavior and resource leaks. The Guidelines explain this directly and give a representative example.

129.13.7.3 Guideline-aligned approach

A base class destructor should be either public and virtual, or protected and non-virtual (C.35).

129.13.7.4 Code sketch

```
#include <memory>
#include <string>

struct Base {
    virtual ~Base() = default; // safe polymorphic destruction
    virtual void f() = 0;
};

struct D : Base {
    std::string s{"resource needing cleanup"};
    void f() override {}
};
```

```
std::unique_ptr<Base> make()
{
    return std::make_unique<D>();
}
```

129.13.8 Example 6 — Non-Null Contracts at Call Sites (Defensive Interfaces)

129.13.8.1 Scenario

A hot path function assumes its pointer parameter must not be null.

129.13.8.2 Primary risks

Hidden null checks everywhere, inconsistent handling, and null dereference bugs.

129.13.8.3 Guideline-aligned approach

If something is not supposed to be `nullptr`, say so using `not_null<T>`. In industrial code, this is typically done with a project support library or a simple local wrapper.

129.13.8.4 Code sketch

```
#include <cassert>

template<class T>
class not_null {
    T ptr;
public:
    explicit not_null(T p) : ptr(p) { assert(ptr != nullptr); }
    operator T() const { return ptr; }
```

```
    T get() const { return ptr; }
};

struct Node { int value; };

int read_value(not_null<const Node*> n)
{
    return n.get()->value;
}
```

129.13.9 Example 7 — Deliberate Low-Level Escape Hatch (Rare Suppression)

129.13.9.1 Scenario

A performance-critical, low-level routine must operate on raw pointers (e.g., scanning a buffer) and cannot be rewritten immediately.

129.13.9.2 Primary risks

Bounds errors and memory corruption.

129.13.9.3 Guideline-aligned approach

When enforcing a safety profile, the Guidelines show that a `[[suppress("bounds")]]` annotation can suppress a bounds check, but emphasize that suppression should be very rare. If an escape hatch is required, isolate it, document it, and test it heavily.

129.13.9.4 Code sketch

```
[[suppress("bounds")]]
```

```
char* raw_find(char* p, int n, char x) // find x in p[0]..p[n-1]
{
    for (int i = 0; i < n; ++i) {
        if (p[i] == x) return p + i;
    }
    return nullptr;
}
```

129.13.10 Industrial Adoption Pattern: A Practical Playbook

The Guidelines explicitly motivate gradual adoption via profiles and warn that enforcing random rules in isolation is disruptive. A realistic adoption plan used in industry is:

1. **Start with one profile target** (Bounds, Lifetime, or Type), enable checks in the analysis tool, and keep that scope warning-free.
2. **Isolate unavoidable low-level code** behind stable, well-tested interfaces.
3. **Fix ownership confusion at boundaries** (ABI, plugins, device drivers) by explicit ownership expression (resource handles, or `owner<T*>` where conversion is too costly).
4. **Harden concurrency hotspots** by eliminating shared writable state and preventing data races.

References

129.14 How This Reference List Is Organized

This book is a technical reference manual built primarily around the *C++ Core Guidelines*. Where comparisons, standards baselines, and tooling enforcement were discussed, the book relies on:

- official ISO/IEC C++ language standards,
- official safety coding guidelines (MISRA, AUTOSAR),
- and official/static-analysis tool documentation for Core Guidelines enforcement.

129.15 Primary Rule Source

Bibliography

[1] Bjarne Stroustrup, Herb Sutter, and contributors. *C++ Core Guidelines*. Public guideline text used by this book (as provided in the project snapshot `CppCoreGuidelines.md`).

129.16 ISO/IEC C++ Language Standards (Normative Baselines)

[2] ISO/IEC. *ISO/IEC 14882:2014(E), Information technology — Programming languages — C++*. International Organization for Standardization / International Electrotechnical Commission, 2014.

[3] ISO/IEC. *ISO/IEC 14882:2017(E), Programming languages — C++*. International Organization for Standardization / International Electrotechnical Commission, 2017.

[4] ISO/IEC. *ISO/IEC 14882:2020(E), Programming languages — C++*. International Organization for Standardization / International Electrotechnical Commission, 2020.

[5] ISO/IEC. *ISO/IEC 14882:2024(E), Programming languages — C++* (C++23 publication). International Organization for Standardization / International Electrotechnical Commission, 2024.

129.17 MISRA and AUTOSAR (Safety-Critical Coding Standards)

[6] MISRA. *MISRA C++:2023* (targets ISO/IEC 14882:2017, commonly referred to as C++17). Motor Industry Software Reliability Association, 2023.

[7] AUTOSAR. *Guidelines for the use of the C++14 language in critical and safety-related systems*. AUTOSAR Adaptive Platform Release 17-10, Document ID 839.

[8] AUTOSAR. *Guidelines for the use of the C++14 language in critical and safety-related systems*. AUTOSAR Adaptive Platform Release 18-03, Document ID 839.

129.18 Guidelines Support Library (GSL) (Support Types and Concepts)

[9] Microsoft. *Guidelines Support Library (GSL): Microsoft's implementation*. Repository documentation for GSL types and contracts referenced by the C++ Core Guidelines (e.g., `not_null`, `owner`, span-like facilities).

[10] Microsoft C++ Team. *Announcing Guidelines Support Library v4.2.0*. Official release note describing GSL as an implementation of the Guidelines Support Library specified by the C++ Core Guidelines.

129.19 Tooling Documentation (Core Guidelines Enforcement)

[11] Microsoft. *Using the C++ Core Guidelines checkers* (Visual Studio / MSVC documentation). Official documentation describing how Visual Studio code analysis supports a subset of Core Guidelines rules.

[12] Microsoft. *C++ Core Guidelines Checker reference* (Visual Studio / MSVC documentation). Official list of analyzer warnings and mapping to relevant Core Guidelines rules.

[13] LLVM Project. *clang-tidy — Extra Clang Tools documentation*. Official documentation for clang-tidy, including its purpose and configuration model.

[14] LLVM Project. *clang-tidy Checks (complete check list)*. Official list of available clang-tidy checks (including `cppcoreguidelines-*` checks).

[15] LLVM Project. *clang-tidy check documentation: examples from the* `cppcoreguidelines-*` *group*. Official per-check documentation showing which Core Guideline rule a check implements (used to support statements about tool enforceability).

129.20 Standards Status and Publication Context (Official Committee/Community Sources)

[16] ISO/IEC JTC1/SC22/WG21. *C++ — Standards*. Official WG21 standards page summarizing published ISO/IEC C++ standards.

[17] Standard C++ Foundation (isocpp.org). *The Standard*. Official community site statement about the current C++ standard and its formal ISO/IEC designation.